CS342 Operating Systems
Supplemantary Notes [1]

Thursday 6th October, 2016

[1]Note that the content here can be updated at any time without any notice. The date will give you information about the version of the document (indicates the last update). *Disclaimer:* Please use these on your own risk. We do not accept any liability in anything that may result in use of the examples and text of these notes. The examples are not extensively tested.

Contents

Chapter 1

Getting Started

1.1 Debugging C Programs in Linux

There are various ways and tools for debugging a C program. In Linux, there are debuggers like gdb, xxgdb, etc. You can also use an IDE (such as Eclipse) to develop and debug your program.

A good way of debugging programs is carefully examining the program source code and using printf statements in proper places. That means, to debug various types of bugs you may not need to use a debugging tool. For debugging segmentation-fault kind of run-time errors, however, use of a debugger can be very helpful to pinpoint quickly which line in code caused the error. A segmentation fault error usually happens when you are trying to access a memory location (location in the logical memory of your program) that should not be accessed (for example because it is not in the used/valid portion of the address space).

Below we show you how you can use gdb tool to debug a program that gives you a wrong-memory access (segmentation fault) error.

First we write a simple program test.c that contains an error intentionally. The program below assigns the address 100 (logical address 100) to a pointer variable p. Probably at logical address 100 of a process there is nothing valid to access. Hence if we want to put a value to the location pointed by p (i.e., location 100), then we should get a segmentation fault error.

```
1
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
```

```
6  int main ()
7  {
8          char *p; // p is a pointer; can store an address
9          int  x;
10
11         x = 10;
12         x++;
13         p = (char *) 100; // he we make a mistake
14         *p = 100; // this should cause an error
15         x = 200;
16         return 0;
17 }
```

We compile this program with the following Makefile.

```
1  all: test
2
3  test: test.c
4          gcc -g -Wall -o test test.c
5
6  clean:
7         rm -fr test *~ *.o
```

The gcc compiler option -g is required for the compiler to generate debugging code so that the program can be used by a debugger. The -W option is used to generate all possible warnings during program compilation so that we can see and fix them.

The program executable is called test. We can now run it. If we run, we get the following error:

```
1  $ ./test
2  Segmentation fault (core dumped)
```

So, we have a run-time error: segmentation fault. How can we learn quickly at which line of the source code (test.c) we have something wrong that can cause this error. For this we run the gdb debugger as follows:

```
1  $ gdb test core
```

We have the following output generated by gdb:

```
1  $ gdb test core
2  GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
```

```
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/korpe/data/os_book/debugging/test...done.
[New LWP 2944]

warning: Can't read pathname for load map: Input/output error.
Core was generated by `./test'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483cf in main () at test.c:14
14                *p = 100; // here I expect a memory access error
(gdb) ^CQuit
(gdb)
```

The gdb output tells us that we have the segmentation error when the program tried to execute the statement *p = 100. In this way we learned with gdb the line which caused the error. Now, we have to examine that line in our program carefully and think why it can cause a memory error.

Note that while invoking gdb we used the program name (test) and a file called core file. That core file is actually a file generated when segmentation fault occurs while running the program. The kernel generates this file. It contains the memory image of the test program at the time of segmentation error. This core file is generated so that a debugger (like gdb) can read and analyze it and understand what had happened at the time of error.

Sometimes, a core file is not generated because the core file size limit might have set to 0 in the corresponding shell program (command interpreter) where we are running our test program. If a core file is not generated, we need to type the following command at the shell prompt:

```
$ ulimit -c unlimited
```

This sets the core file size limit to an infinite value. Now, core file will be generated when a segmentation fault occurs. You can put this command into your .bashrc file so that it is automatically executed when a shell is created.

1.2 An Example C Program

1.2.1 Problem Specification: Priority Queue

We will write a C program that will implement and use priority queue data structure, a data structure that is quite frequently used in operating system implementations to quickly select an element from the set of elements in a queue. We will implement the priority queue as a binary heap. But we will implement the binary heap not as an array, but as a tree (to see more pointer and tree operations).

The program will take as input a file containing a set of words (strings). The maximum length of a word can be 255 characters. The program will read the file one word at a time and insert each word into the priority queue (into the binary heap, i.e., into the binary tree) after reading it from the file. Hence we need to implement an insert() operation. Initially, the priority queue (i.e., the heap) will be an empty tree. After processing the whole input file, we will have the priority queue built up. If a word appears multiple times in the file, it should appear as a single node in the tree, but the node should have a count field indicating how many times the word has appeared. The root of the tree will be containing the minimum element (i.e., the word that is smaller than all other words in the tree). We can compare two words using the strcmp() function of the string library.

After building the tree, we will do successive delete_min() operations until tree becomes empty. Each delete_min() operation will give us the minimum element and meanwhile will delete the element from the tree. Each minimum element retrieved and deleted from the tree will be written to the output file after retrieval. At the end, when tree becomes empty, the output file will contain the words in sorted order. A word repeating x times should appear x times (consecutively) in the output file. The output file will contain one word in a line, but the input file may contain many words in a line or an empty line.

In this way, the program will be sorting the input file into an output file. There is no limit on the number of words that the input file may contain.

The program will be named as pq (priority queue) and will take two parameters as shown below.

```
1 pq infile outfile
```

The infile parameter is the name of the input text file (ASCII file) containing words. The outfile parameter is the name of the output text file containing the same words in sorted order.

1.2.2 Solution

We now provide a possible solution (a C program) for the problem specified above. Note that this will be just one implementation. The solution could be implemented in various different ways. Additonally, normally, a binary heap is implemented using an array. Here, however, we implement it using a tree structure. We do so to exercise more with C pointers and trees.

Below is the code of the program pq.c.

```
1  /*
2     $Id: pq.c,v 1.4 2015/03/03 15:36:39 korpe Exp korpe $
3  */
4
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <fcntl.h>
10 #include <sys/types.h>
11 #include <errno.h>
12 #include <sys/wait.h>
13 #include <math.h>
14
15 #define DIR_UNDEFINED 0
16 #define DIR_LEFT      1
17 #define DIR_RIGHT     2
18
19 #define FALSE 0
20 #define TRUE  1
21
22 #define MAXWORDLEN 255
23
24 struct node {
25         struct node *left;
26         struct node *right;
27         struct node *parent;
28         char *word;
29         int count;
30 };
31
32 struct heap {
33         struct node *root;        /* points to the root node of the heap */
34         int count;                /* number of nodes in the heap */
35 };
36
37 /************ PROTOTYPES OF FUNCTIONS **********/
38 void heap_sort(char *f1, char *f2);
39 void print_heap(struct node *nodep);
40 struct node *find_word (struct node*, char *);
41 struct node *find_afterlast(struct heap *, int *);
42 struct node *find_last(struct heap *);
43 int pq_delete_min(struct heap *, char *);
```

```
44  void pq_insert(struct heap *, char *);
45  /*******************************************************/
46
47  void
48  print_heap(struct node *nodep)
49  {
50          if (nodep == NULL)
51                  return;
52          else {
53                  if (nodep->parent != NULL)
54                          printf ("node=%s, parent=%s count=%d\n",
55                                  nodep->word,
56                                  nodep->parent->word,
57                                  nodep->count);
58                  else
59                          printf ("node=%s, parent=%s count=%d\n",
60                                  nodep->word,
61                                  "NULL",
62                                  nodep->count);
63                  print_heap(nodep->left);
64                  print_heap(nodep->right);
65                  fflush(stdout);
66          }
67  }
68
69
70  void
71  swap(struct node *n1, struct node *n2)
72  {
73          char tempstr[MAXWORDLEN+1];
74          int tempint;
75
76          tempint = n1->count;
77          n1->count = n2->count;
78          n2->count = tempint;
79
80          strcpy(tempstr, n1->word);
81          free(n1->word);
82          n1->word = malloc(strlen(n2->word)+1);
83          strcpy(n1->word, n2->word);
84          free(n2->word);
85          n2->word = malloc(strlen(tempstr)+1);
86          strcpy(n2->word, tempstr);
87  }
88
89  /* returns the parent of the position where the new node
90     will be inserted. returns also if the new node
91     will be inserted as the left or right child
92  */
93  struct node *
94  find_afterlast(struct heap *hp, int *dirp)
95  {
96          unsigned int n, temp, power;
97          int i, k;
98          unsigned int bit;
99          struct node *nodeptr, *child;
```

```
100
101         if (hp->root == NULL) {
102                 *dirp = DIR_UNDEFINED;
103                 return (NULL);
104         }
105
106         /* the heap has at least one element */
107         nodeptr = hp->root;
108         n = (unsigned int) (hp->count + 1);
109         temp = n;
110
111         i = 0;
112         power = 1;
113         temp = temp >> 1;
114         while (temp > 0) {
115                 i++;
116                 power = power << 1;
117                 temp = temp >> 1;
118         }
119         i--;
120         power = power >> 1;
121
122         for (k = i; k >= 1; --k) {
123                 bit = n & power;
124                 if (bit == 0)
125                         child = nodeptr->left;
126                 else
127                         child = nodeptr->right;
128
129                 power = power >> 1;
130                 nodeptr = child;
131         }
132
133         bit = n & power;
134         if (bit == 0)
135                 *dirp = DIR_LEFT;
136         else
137                 *dirp = DIR_RIGHT;
138
139         return (nodeptr);
140 }
141
142 /*
143    find the last node in the heap and return a pointer to that.
144 */
145 struct node *
146 find_last(struct heap *hp)
147 {
148         unsigned int n, temp, power;
149         int i, k;
150         unsigned int bit;
151         struct node *nodeptr, *child;
152
153         if (hp->root == NULL) {
154                 return (NULL);
155         }
```

```
156
157         /*
158            the heap has at least one element
159          */
160         nodeptr = hp->root;
161         n = (unsigned int) (hp->count);
162         temp = n;
163
164         i = 0;
165         power = 1;
166         temp = temp >> 1;
167         while (temp > 0) {
168                 i++;
169                 power = power << 1;
170                 temp = temp >> 1;
171         }
172         i--;
173         power = power >> 1;
174
175         for (k = i; k >= 0; --k) {
176                 bit = n & power;
177                 if (bit == 0)
178                         child = nodeptr->left;
179                 else
180                         child = nodeptr->right;
181
182                 power = power >> 1;
183                 nodeptr = child;
184         }
185
186         return (nodeptr);
187 }
188
189
190 /*
191   if word is already in the tree, returns a pointer to node of the
192   word, else returns NULL.
193 */
194 struct node *
195 find_word (struct node *p, char *word)
196 {
197         struct node *np;
198
199         if (p == NULL)
200                 return (NULL);
201
202         if (strcmp(p->word, word) == 0) {
203                 printf ("returning %s\n", word);
204                 fflush (stdout);
205                 return (p);
206         }
207         else {
208                 np = find_word (p->left, word);
209                 if (np != NULL)
210                         return (np);
211                 np = find_word (p->right, word);
```

```
212                 if (np != NULL)
213                         return (np);
214                 return (NULL);
215         }
216 }
217
218
219
220 /*
221    return the minimum value of heap in variable "word" and
222    delete the node containing the min value if count has reached to 0.
223    returns 1 if one element is deleted, otherwise returns 0 if
224    nothing is deleted.
225 */
226 int
227 pq_delete_min(struct heap *hp, char *word)
228 {
229
230         struct node *lastptr, *parent;
231         int mindir;
232         char minvalue[MAXWORDLEN+1];
233
234         if (hp->root == NULL)
235                 return (0);
236
237         strcpy(word, hp->root->word);
238
239         hp->root->count--;
240
241         printf ("deleting %s %d\n", hp->root->word, hp->root->count);
242         if (hp->root->count > 0)
243                 return (1);
244
245         lastptr = find_last(hp);
246
247         if (lastptr == hp->root) {
248                 free(hp->root);
249                 hp->root = NULL;
250                 hp->count = 0;
251                 return (1);
252         }
253
254         swap(hp->root, lastptr);
255
256         if (lastptr->parent->left == lastptr)
257                 lastptr->parent->left = NULL;
258         else if (lastptr->parent->right == lastptr)
259                 lastptr->parent->right = NULL;
260
261         free(lastptr->word);
262         free(lastptr);
263         hp->count--;
264
265         parent = hp->root;
266
267         while (parent->left || parent->right) {
```

```
268                 if (parent->left != NULL && parent->right == NULL) {
269                         if (strcmp(parent->left->word,
270                                    parent->word) == -1) {
271                                 swap(parent->left, parent);
272                                 parent = parent->left;
273                         } else
274                                 break;
275                 } else if (parent->right != NULL && parent->left == NULL) {
276                         if (strcmp(parent->right->word,
277                                    parent->word) == -1) {
278                                 swap(parent->right, parent);
279                                 parent = parent->right;
280                         } else
281                                 break;
282                 } else if (parent->left != NULL && parent->right != NULL) {
283                         if (strcmp(parent->left->word,
284                                    parent->right->word) == -1) {
285                                 mindir = DIR_LEFT;
286                                 strcpy(minvalue, parent->left->word);
287                         } else  {
288                                 mindir = DIR_RIGHT;
289                                 strcpy(minvalue, parent->right->word);
290                         }
291
292                         if (strcmp (minvalue, parent->word) == -1) {
293                                 if (mindir == DIR_LEFT) {
294                                         swap (parent->left, parent);
295                                         parent = parent->left;
296                                 } else {
297                                         swap(parent->right, parent);
298                                         parent = parent->right;
299                                 }
300                         } else {
301                                 break;
302                         }
303                 }
304         }
305         return (1);
306 }
307
308
309 /*
310    insert a new number (and hence a new node) into the heap.
311 */
312 void
313 pq_insert(struct heap *hp, char *wp)
314 {
315         struct node *parentptr;
316         int dir;
317         struct node *p, *child, *parent;
318
319         p = find_word(hp->root, wp);
320         if (p != NULL) {
321                 p->count++;
322                 return;
323         }
```

```
324
325         p = (struct node *) malloc(sizeof (struct node));
326         if (p == NULL) {
327                 printf("malloc failed \n");
328                 exit(1);
329         }
330         p->left = NULL;
331         p->right = NULL;
332         p->parent = NULL;
333         p->word = malloc (strlen(wp)+1);
334         if (p->word == NULL) {
335                 perror ("malloc:");
336                 exit (1);
337         }
338         strcpy(p->word, wp);
339         p->count = 1;
340
341         if (hp->root == NULL) {
342                 hp->root = p;
343                 hp->count = 1;
344                 return;
345         }
346
347         /* we have at least one element at this point in the
348            heap other than the element we want to insert
349          */
350         parentptr = find_afterlast(hp, &dir);
351
352         p->parent = parentptr;
353
354         if (dir == DIR_LEFT)
355                 parentptr->left = p;
356         else
357                 parentptr->right = p;
358
359         parent = parentptr;
360         child = p;
361
362         while (parent != NULL) {
363                 if (strcmp(child->word, parent->word) == -1) {
364                         swap(parent, child);
365                         child = parent;
366                         parent = parent->parent;
367                 } else
368                         break;
369         }
370         hp->count++;
371 }
372
373 /*
374    sort the integers in file f1 and output
375    the result into file f2
376 */
377 void
378 heap_sort(char *f1, char *f2)
379 {
```

```
380         FILE *fp1;
381         FILE *fp2;
382         struct heap *heapptr;
383         char aword[MAXWORDLEN+1];
384
385
386         heapptr = (struct heap *) malloc(sizeof (struct heap));
387         if (heapptr == NULL) {
388                 printf("malloc failed\n");
389                 exit(1);
390         }
391         heapptr->root = NULL;
392         heapptr->count = 0;
393
394         fp1 = fopen(f1, "r");
395         if (fp1 == NULL) {
396                 perror("sort:");
397                 exit(1);
398         }
399
400         while (fscanf(fp1, "%s", aword) == 1) {
401                 pq_insert(heapptr, aword);
402                 printf ("%s %d\n", aword, (int)  strlen(aword));
403         }
404
405         fclose(fp1);
406
407         print_heap (heapptr->root);
408
409         fp2 = fopen(f2, "w");
410         if (fp2 == NULL) {
411                 perror("sort:");
412                 exit(1);
413         }
414
415         while (pq_delete_min(heapptr, aword) == 1) {
416                 fprintf(fp2, "%s\n", aword);
417         }
418
419
420         fclose(fp2);
421
422         free(heapptr);
423 }
424
425
426
427 int main(int argc, char **argv)
428 {
429
430         if (argc != 3) {
431                 printf ("usage: pq infiletxt outfiletxt\n");
432                 exit (0);
433         }
434
435         heap_sort (argv[1], argv[2]);
```

```
436
437         return (0);
438 }
```

Below is the Makefile to compile the program and obtain an executable file pq.

```
1 all: pq
2
3 pq: pq.c
4         gcc -g -Wall -o pq pq.c
5
6 clean:
7         rm -fr *~ core* *.o  pq
```

Chapter 2

System Calls

2.1 What is a system call

System calls are kernel routines that can be called by applications to get some service from the operating system, for example, to open a file, to read from a file, to write into a file, etc. When a process invokes the execution of such a routine, we say the process makes a system call. A process may make a lot of system calls during its lifetime. To see this, we will examine a small program shown below, that is copying one file into another file. The program copies a file one byte at a time. This is of course not very efficient.

Below is the copy program.

```
1  /* -*- linux-c  -*-   */
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <unistd.h>
10 #include <errno.h>
11
12 #define MAXLINE 256
13
14
15 void function_copy(int fd1, int fd2)
16 {
17         int ret;
18         unsigned char abyte;
19
20         while ( (ret = read (fd1, (void *)&abyte, 1)) == 1)
21         {
```

```
22                  ret = write (fd2, (void *)&abyte, 1);
23                  if (ret != 1) {
24                          printf ("write error: should not happen.\n");
25                  }
26          }
27
28  }
29
30  int
31  main(int argc, char **argv)
32  {
33          char infilename[128];
34          char outfilename[128];
35          int in_fd, out_fd;
36
37
38          strcpy (infilename, argv[1]);
39          strcpy (outfilename, argv[2]);
40
41          in_fd = open (infilename, O_RDONLY);
42          if (in_fd == -1) {
43                  printf ("could not open input file\n");
44                  exit (1);
45          }
46
47          out_fd = open (outfilename, O_WRONLY | O_CREAT, 00754);
48          if (out_fd == -1) {
49                  perror ("could not open output file\n");
50                  exit (1);
51          }
52
53          function_copy (in_fd, out_fd);
54
55          close (in_fd);
56          close (out_fd);
57
58          printf("finished copying\n");
59
60          return (0);
61  }
```

Below we also include a Makefile that you can use to compile the copy program.

```
1  all: copy
2
3  copy: copy.c
4          gcc -static -g -Wall -o copy copy.c
5
6  clean:
7          rm -fr *~ copy
```

Note that we are compiling with `static` option of the gcc compiler. This means that the static version of the standard C library will be used in linking with our program. Hence

the executable file of our program will contain the C library routines in machine code form. They will not be loaded and linked dynamically at run time. They are linked statically at compile when static option is used.

Assume the following is the input file that we want to copy.

```
This is an input file content.
Prepared in Bilkent University.
Prepared in 2010.
```

We will use the `strace` utility in Linux to see which system calls are made while the copy program is running. To trace the system calls made by our copy program, we write the following at the command line. Assume our program name is `copy` and it takes its arguments from the command line.

```
strace -o calls_made.txt -tt -T ./copy in.txt out.txt
```

When we type this, the strace program will start running. It will start the execution of our copy program by using the execve system call. Then it will trace the execution of the copy program and will record the system calls made into an output file called `calls_made.txt` here. When copy program and strace program finishes, we have the output file `calls_made.txt` containing the information about the system calls made by the copy program. Below is the information that we get.

```
13:19:27.729250 execve("./copy", ["./copy", "in.txt", "out.txt"], [/* 35 vars */]) = 0 <0.000154>
13:19:27.729741 uname({sys="Linux", node="pckorpe", ...}) = 0 <0.000010>
13:19:27.729996 brk(0)                  = 0x9489000 <0.000009>
13:19:27.730047 brk(0x9489cd0)          = 0x9489cd0 <0.000010>
13:19:27.730105 set_thread_area({entry_number:-1 -> 6, base_addr:0x9489830, limit:1048575, seg_32bit:1, content
13:19:27.730201 brk(0x94aacd0)          = 0x94aacd0 <0.000010>
13:19:27.730251 brk(0x94ab000)          = 0x94ab000 <0.000009>
13:19:27.730319 open("in.txt", O_RDONLY) = 3 <0.000014>
13:19:27.730382 open("out.txt", O_WRONLY|O_CREAT, 0754) = 4 <0.000014>
13:19:27.730445 read(3, "T", 1)         = 1 <0.000011>
13:19:27.730498 write(4, "T", 1)        = 1 <0.000017>
13:19:27.730557 read(3, "h", 1)         = 1 <0.000010>
13:19:27.730609 write(4, "h", 1)        = 1 <0.000012>
13:19:27.730662 read(3, "i", 1)         = 1 <0.000010>
13:19:27.730713 write(4, "i", 1)        = 1 <0.000012>
13:19:27.730766 read(3, "s", 1)         = 1 <0.000010>
13:19:27.730817 write(4, "s", 1)        = 1 <0.000012>
13:19:27.730870 read(3, " ", 1)         = 1 <0.000010>
13:19:27.730921 write(4, " ", 1)        = 1 <0.000012>
13:19:27.730974 read(3, "i", 1)         = 1 <0.000010>
13:19:27.731025 write(4, "i", 1)        = 1 <0.000012>
13:19:27.731078 read(3, "s", 1)         = 1 <0.000010>
```

```
13:19:27.731129 write(4, "s", 1)        = 1 <0.000012>
13:19:27.731182 read(3, " ", 1)         = 1 <0.000010>
13:19:27.731233 write(4, " ", 1)        = 1 <0.000012>
13:19:27.731286 read(3, "a", 1)         = 1 <0.000010>
13:19:27.731337 write(4, "a", 1)        = 1 <0.000012>
13:19:27.731390 read(3, "n", 1)         = 1 <0.000010>
13:19:27.731442 write(4, "n", 1)        = 1 <0.000012>
13:19:27.731494 read(3, " ", 1)         = 1 <0.000010>
13:19:27.731557 write(4, " ", 1)        = 1 <0.000015>
13:19:27.731615 read(3, "i", 1)         = 1 <0.000010>
13:19:27.731666 write(4, "i", 1)        = 1 <0.000012>
13:19:27.731719 read(3, "n", 1)         = 1 <0.000010>
13:19:27.731770 write(4, "n", 1)        = 1 <0.000012>
13:19:27.731823 read(3, "p", 1)         = 1 <0.000010>
13:19:27.731874 write(4, "p", 1)        = 1 <0.000012>
13:19:27.731927 read(3, "u", 1)         = 1 <0.000010>
13:19:27.731978 write(4, "u", 1)        = 1 <0.000012>
13:19:27.732031 read(3, "t", 1)         = 1 <0.000011>
13:19:27.732082 write(4, "t", 1)        = 1 <0.000013>
13:19:27.732135 read(3, " ", 1)         = 1 <0.000011>
13:19:27.732186 write(4, " ", 1)        = 1 <0.000012>
13:19:27.732239 read(3, "f", 1)         = 1 <0.000011>
13:19:27.732290 write(4, "f", 1)        = 1 <0.000013>
13:19:27.732343 read(3, "i", 1)         = 1 <0.000010>
13:19:27.732394 write(4, "i", 1)        = 1 <0.000012>
13:19:27.732446 read(3, "l", 1)         = 1 <0.000010>
13:19:27.732497 write(4, "l", 1)        = 1 <0.000012>
13:19:27.732550 read(3, "e", 1)         = 1 <0.000010>
13:19:27.732601 write(4, "e", 1)        = 1 <0.000012>
13:19:27.732654 read(3, " ", 1)         = 1 <0.000010>
13:19:27.732705 write(4, " ", 1)        = 1 <0.000012>
13:19:27.732758 read(3, "c", 1)         = 1 <0.000010>
13:19:27.732809 write(4, "c", 1)        = 1 <0.000013>
13:19:27.732862 read(3, "o", 1)         = 1 <0.000010>
13:19:27.732913 write(4, "o", 1)        = 1 <0.000012>
13:19:27.732965 read(3, "n", 1)         = 1 <0.000010>
13:19:27.733017 write(4, "n", 1)        = 1 <0.000012>
13:19:27.733069 read(3, "t", 1)         = 1 <0.000010>
13:19:27.733121 write(4, "t", 1)        = 1 <0.000012>
13:19:27.733173 read(3, "e", 1)         = 1 <0.000010>
13:19:27.733224 write(4, "e", 1)        = 1 <0.000012>
13:19:27.733276 read(3, "n", 1)         = 1 <0.000010>
13:19:27.733328 write(4, "n", 1)        = 1 <0.000013>
13:19:27.733381 read(3, "t", 1)         = 1 <0.000010>
13:19:27.733432 write(4, "t", 1)        = 1 <0.000013>
13:19:27.733485 read(3, ".", 1)         = 1 <0.000011>
13:19:27.733536 write(4, ".", 1)        = 1 <0.000013>
13:19:27.733589 read(3, " ", 1)         = 1 <0.000011>
13:19:27.733649 write(4, " ", 1)        = 1 <0.000012>
13:19:27.733702 read(3, "\n", 1)        = 1 <0.000009>
13:19:27.733753 write(4, "\n", 1)       = 1 <0.000012>
13:19:27.733806 read(3, "P", 1)         = 1 <0.000010>
13:19:27.733857 write(4, "P", 1)        = 1 <0.000012>
13:19:27.733910 read(3, "r", 1)         = 1 <0.000010>
13:19:27.733961 write(4, "r", 1)        = 1 <0.000012>
13:19:27.734014 read(3, "e", 1)         = 1 <0.000010>
```

```
13:19:27.734065 write(4, "e", 1)        = 1 <0.000012>
13:19:27.734118 read(3, "p", 1)         = 1 <0.000010>
13:19:27.734168 write(4, "p", 1)        = 1 <0.000012>
13:19:27.734221 read(3, "a", 1)         = 1 <0.000010>
13:19:27.734272 write(4, "a", 1)        = 1 <0.000012>
13:19:27.734325 read(3, "r", 1)         = 1 <0.000010>
13:19:27.734377 write(4, "r", 1)        = 1 <0.000012>
13:19:27.734429 read(3, "e", 1)         = 1 <0.000010>
13:19:27.734480 write(4, "e", 1)        = 1 <0.000012>
13:19:27.734532 read(3, "d", 1)         = 1 <0.000010>
13:19:27.734584 write(4, "d", 1)        = 1 <0.000013>
13:19:27.734637 read(3, " ", 1)         = 1 <0.000010>
13:19:27.734688 write(4, " ", 1)        = 1 <0.000013>
13:19:27.734741 read(3, "i", 1)         = 1 <0.000011>
13:19:27.734792 write(4, "i", 1)        = 1 <0.000012>
13:19:27.734845 read(3, "n", 1)         = 1 <0.000011>
13:19:27.734895 write(4, "n", 1)        = 1 <0.000012>
13:19:27.734948 read(3, " ", 1)         = 1 <0.000010>
13:19:27.734999 write(4, " ", 1)        = 1 <0.000012>
13:19:27.735052 read(3, "B", 1)         = 1 <0.000010>
13:19:27.735104 write(4, "B", 1)        = 1 <0.000012>
13:19:27.735156 read(3, "i", 1)         = 1 <0.000010>
13:19:27.735208 write(4, "i", 1)        = 1 <0.000011>
13:19:27.735259 read(3, "l", 1)         = 1 <0.000010>
13:19:27.735311 write(4, "l", 1)        = 1 <0.000012>
13:19:27.735363 read(3, "k", 1)         = 1 <0.000010>
13:19:27.735414 write(4, "k", 1)        = 1 <0.000012>
13:19:27.735467 read(3, "e", 1)         = 1 <0.000088>
13:19:27.735602 write(4, "e", 1)        = 1 <0.000015>
13:19:27.735658 read(3, "n", 1)         = 1 <0.000010>
13:19:27.735710 write(4, "n", 1)        = 1 <0.000012>
13:19:27.735763 read(3, "t", 1)         = 1 <0.000011>
13:19:27.735814 write(4, "t", 1)        = 1 <0.000012>
13:19:27.735867 read(3, " ", 1)         = 1 <0.000011>
13:19:27.735918 write(4, " ", 1)        = 1 <0.000012>
13:19:27.735971 read(3, "U", 1)         = 1 <0.000011>
13:19:27.736022 write(4, "U", 1)        = 1 <0.000012>
13:19:27.736075 read(3, "n", 1)         = 1 <0.000010>
13:19:27.736127 write(4, "n", 1)        = 1 <0.000012>
13:19:27.736179 read(3, "i", 1)         = 1 <0.000010>
13:19:27.736231 write(4, "i", 1)        = 1 <0.000012>
13:19:27.736283 read(3, "v", 1)         = 1 <0.000010>
13:19:27.736335 write(4, "v", 1)        = 1 <0.000012>
13:19:27.736387 read(3, "e", 1)         = 1 <0.000010>
13:19:27.736439 write(4, "e", 1)        = 1 <0.000012>
13:19:27.736491 read(3, "r", 1)         = 1 <0.000010>
13:19:27.736543 write(4, "r", 1)        = 1 <0.000012>
13:19:27.736596 read(3, "s", 1)         = 1 <0.000011>
13:19:27.736647 write(4, "s", 1)        = 1 <0.000012>
13:19:27.736700 read(3, "i", 1)         = 1 <0.000011>
13:19:27.736751 write(4, "i", 1)        = 1 <0.000012>
13:19:27.736804 read(3, "t", 1)         = 1 <0.000011>
13:19:27.736855 write(4, "t", 1)        = 1 <0.000012>
13:19:27.736908 read(3, "y", 1)         = 1 <0.000010>
13:19:27.736959 write(4, "y", 1)        = 1 <0.000012>
13:19:27.737012 read(3, ".", 1)         = 1 <0.000010>
```

```
13:19:27.737063 write(4, ".", 1)        = 1 <0.000012>
13:19:27.737116 read(3, " ", 1)         = 1 <0.000010>
13:19:27.737167 write(4, " ", 1)        = 1 <0.000012>
13:19:27.737220 read(3, "\n", 1)        = 1 <0.000010>
13:19:27.737272 write(4, "\n", 1)       = 1 <0.000012>
13:19:27.737324 read(3, "P", 1)         = 1 <0.000010>
13:19:27.737376 write(4, "P", 1)        = 1 <0.000012>
13:19:27.737428 read(3, "r", 1)         = 1 <0.000010>
13:19:27.737480 write(4, "r", 1)        = 1 <0.000013>
13:19:27.737532 read(3, "e", 1)         = 1 <0.000010>
13:19:27.737584 write(4, "e", 1)        = 1 <0.000013>
13:19:27.737645 read(3, "p", 1)         = 1 <0.000010>
13:19:27.737697 write(4, "p", 1)        = 1 <0.000013>
13:19:27.737750 read(3, "a", 1)         = 1 <0.000011>
13:19:27.737801 write(4, "a", 1)        = 1 <0.000012>
13:19:27.737854 read(3, "r", 1)         = 1 <0.000011>
13:19:27.737905 write(4, "r", 1)        = 1 <0.000012>
13:19:27.737958 read(3, "e", 1)         = 1 <0.000010>
13:19:27.738009 write(4, "e", 1)        = 1 <0.000012>
13:19:27.738062 read(3, "d", 1)         = 1 <0.000010>
13:19:27.738113 write(4, "d", 1)        = 1 <0.000012>
13:19:27.738166 read(3, " ", 1)         = 1 <0.000010>
13:19:27.738218 write(4, " ", 1)        = 1 <0.000013>
13:19:27.738271 read(3, "i", 1)         = 1 <0.000011>
13:19:27.738322 write(4, "i", 1)        = 1 <0.000013>
13:19:27.738375 read(3, "n", 1)         = 1 <0.000011>
13:19:27.738426 write(4, "n", 1)        = 1 <0.000013>
13:19:27.738479 read(3, " ", 1)         = 1 <0.000011>
13:19:27.738530 write(4, " ", 1)        = 1 <0.000012>
13:19:27.738583 read(3, "2", 1)         = 1 <0.000010>
13:19:27.738635 write(4, "2", 1)        = 1 <0.000012>
13:19:27.738687 read(3, "0", 1)         = 1 <0.000010>
13:19:27.738739 write(4, "0", 1)        = 1 <0.000012>
13:19:27.738791 read(3, "1", 1)         = 1 <0.000010>
13:19:27.738843 write(4, "1", 1)        = 1 <0.000012>
13:19:27.738896 read(3, "0", 1)         = 1 <0.000011>
13:19:27.738947 write(4, "0", 1)        = 1 <0.000012>
13:19:27.739000 read(3, ".", 1)         = 1 <0.000010>
13:19:27.739051 write(4, ".", 1)        = 1 <0.000012>
13:19:27.739104 read(3, " ", 1)         = 1 <0.000010>
13:19:27.739155 write(4, " ", 1)        = 1 <0.000012>
13:19:27.739208 read(3, "\n", 1)        = 1 <0.000010>
13:19:27.739260 write(4, "\n", 1)       = 1 <0.000012>
13:19:27.739312 read(3, "", 1)          = 0 <0.000010>
13:19:27.739362 close(3)                = 0 <0.000011>
13:19:27.739408 close(4)                = 0 <0.000011>
13:19:27.739471 fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0 <0.000010>
13:19:27.739593 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7856000 <0.00001
13:19:27.739654 write(1, "finished copying\n", 17) = 17 <0.000016>
13:19:27.739727 exit_group(0)           = ?
```

In this output, please ignore a few system calls that are made initially. One of them is the execve system call called by the strace to start the copy program. Examine the system calls starting with the `open()` system call and ending with the `close()` system call. All

the system calls in between are the system calls made by our program as the qdirect result of the C code that we have written.

We see that our copy program calls the open and close system calls twice and read and write system calls many times. The open system call returns an integer file descriptor corresponding to the opened file. Here, the descriptor value is 3 for the input file and 4 for the output file. Those numbers could be different. They do not have to be 3 or 4. The read and write system calls use those descriptors to refer to the files to read or write.

The output shows when each system call is made and also how long each system call takes. Note that the time spent in a system call is quite short because at the time that those calls are made, the file content was already in memory (i.e., cached). Hence the system calls did not wait for disk I/O. Those read and write system calls could be immediately satisfied.

2.2 Timing a program execution

We can time the execution of programs, i.e., processes. The `time` utility of Linux can be used for this purpose. It runs a program and measures how much time the program code has spent in CPU (user mode execution time) and how much time the kernel/system code has spent in CPU on behalf of the program (i.e., while giving some services to the program). The user and system times do not include the waiting/blocking for I/O. They are pure CPU times. The real-time is the time between start of the program and end of the program. It includes the waiting time of the process for various reasons such as waiting for an I/O operation (that could not be satisfied immediately) to complete, or waiting for some other processes running in the CPU.

The following is an example for the usage of the time command to time the compilation of our simple copy program. Note that there is also a built-in shell version of the time command. We want to use the time system program located in the /usr/bin directory, not the built-in time command, because the time system program provides better information than the built-in time command.

```
1 /usr/bin/time gcc -static -g -Wall -o copy copy.c
2 real        0m0.130s
3 user        0m0.076s
4 sys        0m0.048s
```

As the output shows, while the gcc compiler was running, 76 ms of CPU time is used to run the gcc code (user mode) in the CPU and 48 ms of CPU time is used to run the kernel code in the CPU. The total time the process stayed in the system is 130 ms. It includes the waiting times for various events (I/O, etc.), if any.

To give another example, lets run the standard `cp` command to copy a large (33 MB) file and measure how long it takes. A sample file, findblocks.tar, which is 33 MB, is copied into a file x.

```
time cp  /home/korpe/data/findblocks.tar  x

real 0m0.461s user 0m0.000s sys 0m0.076s
```

The user code of the process spends nearly 0 seconds in the CPU. But the kernel (system) code spends 76 ms (system time) in the CPU to do something for the process. This does not include the waiting times. It is the time kernel code has run in the CPU. The total lifetime of the process is much larger than the kernel running time. It is 461 ms, which inludes waiting times.

The cp command is quite efficient in copying. We could use our program as well to perform the same copy operation. But our program is copying byte by byte. Hence it is very inefficient. It will take too much time to copy 33 MB. This is indeed so, as shown below.

```
time ./copy   /home/korpe/data/findblocks.tar  x
finished copying

real        1m23.661s
user        0m10.641s
sys        1m12.985s
```

It took more than 1 minute for the program to finish. The program code itself (in user mode) spent aroung 10 seconds in the CPU. The kernel code (in kernel mode) spent around 1 minute and 12 seconds in the CPU for performing system call executions requested by the program. The waiting time for I/O is not that much because the file that is copied was already in memory (cached), since this copy operation is executed after the standard cp command which had caused the kernel to cache the file content.

2.3 System call invocation mechanism

Let us now see how a system call is invoked by a program. We said that actually a library function is invoking a system call, not the program itself. We will see now how this is happening. We will use again our copy program to illustrate this. We run this program in a machine whose CPU architecture is Intel x86.

Our copy program is linked with the standard C library statically. That means the standard C library became part of our program executable file. It is not to be loaded and

linked dynamically at run time anymore. Therefore, if we look to the size of our executable file, it is quite large. It is around 580 KB. If we would have linked our program with dynamically linkable version of the C library, then the size of the executable would be around 10 KB. A big difference. Dynamic linking is more common and it is the default while you are compiling.

Static linking will enable us to examine the machine code (and assembly code) of the standard C library and our program. We can use the `objdump` utility in Linux for this purpose. If we type the following, we will obtain the machine code of our executable file.

```
objdump  -d copy > machine-code.txt
```

This will disassemble our program and will generate the assembly code and the machine code of the program into an output file, called machine-code.txt. In that file, the code of the assembly and machine code of our function `function_copy()` in our C program is as follows:

```
08048250 <function_copy>:
 8048250:       55                          push   %ebp
 8048251:       89 e5                       mov    %esp,%ebp
 8048253:       83 ec 28                    sub    $0x28,%esp
 8048256:       eb 2f                       jmp    8048287 <function_copy+0x37>
 8048258:       c7 44 24 08 01 00 00        movl   $0x1,0x8(%esp)
 804825f:       00
 8048260:       8d 45 f7                    lea    -0x9(%ebp),%eax
 8048263:       89 44 24 04                 mov    %eax,0x4(%esp)
 8048267:       8b 45 0c                    mov    0xc(%ebp),%eax
 804826a:       89 04 24                    mov    %eax,(%esp)
 804826d:       e8 1e e0 00 00              call   8056290 <__libc_write>
 8048272:       89 45 f0                    mov    %eax,-0x10(%ebp)
 8048275:       83 7d f0 01                 cmpl   $0x1,-0x10(%ebp)
 8048279:       74 0c                       je     8048287 <function_copy+0x37>
 804827b:       c7 04 24 a8 75 0a 08        movl   $0x80a75a8,(%esp)
 8048282:       e8 39 12 00 00              call   80494c0 <_IO_puts>
 8048287:       c7 44 24 08 01 00 00        movl   $0x1,0x8(%esp)
 804828e:       00
 804828f:       8d 45 f7                    lea    -0x9(%ebp),%eax
 8048292:       89 44 24 04                 mov    %eax,0x4(%esp)
 8048296:       8b 45 08                    mov    0x8(%ebp),%eax
 8048299:       89 04 24                    mov    %eax,(%esp)
 804829c:       e8 8f df 00 00              call    <__libc_read>
 80482a1:       89 45 f0                    mov    %eax,-0x10(%ebp)
 80482a4:       83 7d f0 01                 cmpl   $0x1,-0x10(%ebp)
 80482a8:       74 ae                       je     8048258 <function_copy+0x8>
 80482aa:       c9                          leave
 80482ab:       c3                          ret
```

At address 804829c (addresses are in hex), we see the call to the library function read()

(libc_read). The address of the read library function is 8056230. If we now go that address, we see the following:

```
08056230 <__libc_read>:
 8056230:       65 83 3d 0c 00 00 00        cmpl   $0x0,%gs:0xc
 8056237:       00
 8056238:       75 21                       jne    805625b <__read_nocancel+0x21>

0805623a <__read_nocancel>:
 805623a:       53                          push   %ebx
 805623b:       8b 54 24 10                 mov    0x10(%esp),%edx
 805623f:       8b 4c 24 0c                 mov    0xc(%esp),%ecx
 8056243:       8b 5c 24 08                 mov    0x8(%esp),%ebx
 8056247:       b8 03 00 00 00              mov    $0x3,%eax
 805624c:       cd 80                       int    $0x80
 805624e:       5b                          pop    %ebx
 805624f:       3d 01 f0 ff ff              cmp    $0xfffff001,%eax
 8056254:       0f 83 76 21 00 00           jae    80583d0 <__syscall_error>
 805625a:       c3                          ret
 805625b:       e8 90 0f 00 00              call   80571f0 <__libc_enable_asynccancel>
 8056260:       50                          push   %eax
 8056261:       53                          push   %ebx
 8056262:       8b 54 24 14                 mov    0x14(%esp),%edx
 8056266:       8b 4c 24 10                 mov    0x10(%esp),%ecx
 805626a:       8b 5c 24 0c                 mov    0xc(%esp),%ebx
 805626e:       b8 03 00 00 00              mov    $0x3,%eax
 8056273:       cd 80                       int    $0x80
 8056275:       5b                          pop    %ebx
 8056276:       87 04 24                    xchg   %eax,(%esp)
 8056279:       e8 e2 0e 00 00              call   8057160 <__libc_disable_asynccancel>
 805627e:       58                          pop    %eax
 805627f:       3d 01 f0 ff ff              cmp    $0xfffff001,%eax
 8056284:       0f 83 46 21 00 00           jae    80583d0 <__syscall_error>
 805628a:       c3                          ret
 805628b:       90                          nop
 805628c:       90                          nop
 805628d:       90                          nop
 805628e:       90                          nop
 805628f:       90                          nop
```

As shown, libc_read is calling read_nocancel immediately. Let us examine that function now. At address 805624c, we see the following assembly and machine code:

```
805624c: cd 80 int $0x80
```

This is a software interrupt (TRAP). This is a system call. In Intel architecture, the machine instruction corresponding to the generic trap instruction is the `int $0x80` machine instruction. The execution of this instruction causes the program to be suspended at this point and kernel code to start running. The kernel will handle the system call (the

software interrupt). It will understand exactly which system call is called by looking to the eax register of the CPU. Note that, above, the machine instruction before the TRAP (before the int) is a mov instruction. The value 3 is moved to the eax register. The is the system call number. In fact, in Linux kernel, this is the system call number assigned to the read system call (sys_read function inside the kernel is the corresponding routine implementing the read operation). Hence, after software interrupt, the kernel will use this number as an Index to the system call table to retrieve the address of the read system call. Then the read system call of the kernel will start executing in the CPU.

Now, you can find out also the call to the write system call in the machine code of the function `function_copy()` of our C program. You can find out the system call number put into the eax register that is corresponding to the write system call.

2.4 Nonblocking System Calls

Normally, system calls block the calling process until the requested operation is completed. For example, if a process would like to read something from the keyboard, the process will be blocked until the user provides the input (if not already available in a kernel buffer) by typing at the keyboard. Or, for example, if a process would like to receive from a socket or from a message queue when there is no data available, the process will be blocked until data becomes available at the socket or at the message queue to receive.

If a process has opened many sockets (network connections) or message queues, it may not be desirable to block the process on a socket or message queue when there is no data available at that socket or message queue. Maybe there is another socket or message queue where data is available. It may be better for the process to receive the available data from another socket or message queue. To be able to do this, a process should not block on a message queue for an undeterministic amount of time. We can use nonblocking calls for this purpose. Or we could also use a separate thread for each socket or message queue. Then if a thread would block on a socket/queue, another thread may receive data from another socket/queue.

Here we will show the use of nonblocking send and receive operations on message queues via a simple producer-consumer program.

In this sample application, there are 3 message queues created between a producer process and a consumer process. The producer takes an input filename as an argument. The consumer takes an output filename as an argument. The producer reads the input file (which contains a set of positive integers), one integer at a time, and puts an integer x to a message queue x mod 3. For example, integer 7 is put into queue 1. The consumer process tries to receive integers from these 3 message queues. For that it uses the mq_receive operation, but in nonblocking mode. Hence if a message queue on which we invoke mq_receive operation does not have data, the mq_receive operation does not block the calling process,

the consumer, and the consumer can try to read from another message queue.

We can set send and receive operations on a message queue to be nonblocking by providing a special option (O_NONBLOCK) to the mq_open system call, that is used to create/open a message queue. Below is an example:

```
mq_open(mq_name, O_RDWR | O_CREAT | O_NONBLOCK, 0666, NULL);
```

The call above creates and opens a message queue with name mq_name (created if it does not already exist). The queue is opened with the O_NONBLOCK option. This means mq_send and mq_receive operations on the message queue will not block the caller.

If consumer calls mq_receive on a message queue that does not have data (message) yet, the call will return immediately. The return value will indicate -1 (no success). Then we need to check what has happened. Did the call return because there was no data or because there was an error during the execution of the system call. This can be checked by looking to the value of the global variable `errno` (that is defined in the standard C library). A system call puts the error code into this variable. If the value (code) put into errno variable is EAGAIN, that means there was no error during the execution of the system and the call returned because there was not data. Hence, there is no serious error condition and we need to check data availability later again.

Below we show the code of consumer and producer programs. We also show a Makefile to compile the programs. Below is the Makefile content.

```
1  all:         consumer producer
2
3  consumer: consumer.c
4                  gcc -Wall -o consumer consumer.c -lrt
5
6  producer: producer.c
7          gcc -Wall -o producer producer.c -lrt
8
9  clean:
10         rm -fr *~ producer consumer
```

The common definitions shared by the consumer program and producer program are put into a header file called common.h. Below is the content of this header file.

```
1  #define MQNAME_PREFIX   "/a_msg_queue"
2  #define ITEMSIZE sizeof(int)
3  #define INVALID_INT -1000
```

Below is the consumer program.

```
1  /* -*- linux-c  -*-   */
2  /* $Id: consumer.c,v 1.7 2015/03/17 08:58:46 korpe Exp korpe $ */
3
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <mqueue.h>
7  #include <unistd.h>
8  #include <errno.h>
9  #include <string.h>
10 #include <fcntl.h>           /* For O_* constants */
11 #include <sys/stat.h>        /* For mode constants */
12 #include "common.h"
13
14 extern int errno; /* a system call puts the error code here */
15
16 int main(int argc, char **argv)
17 {
18         mqd_t mq[3];
19         struct mq_attr mq_attr;
20         char mq_name[128];
21         char *buffer;
22         int buflen;
23         int n, i;
24         int x;
25         FILE *fp;
26
27         fp = fopen (argv[1], "w");
28         if (fp == NULL) {
29                 printf ("can not open file\n");
30                 exit (1);
31         }
32
33         for (i = 0; i < 3; ++i) {
34                 sprintf (mq_name, "%s%d", MQNAME_PREFIX, i);
35                 mq[i] = mq_open(mq_name,
36                                 O_RDWR | O_CREAT | O_NONBLOCK,
37                                 0666,
38                                 NULL);
39                 if (mq[i] < 0) {
40                         perror("can not create/open msg queue\n");
41                         exit(1);
42                 }
43         }
44
45
46         /* get the max_size that a message queue can handle */
47         mq_getattr(mq[0], &mq_attr);
48         buflen = mq_attr.mq_msgsize;
49         buffer = (char *) malloc(buflen);
50
51         while (1) {
52                 for (i = 0; i < 3; ++i) {
53                         n = mq_receive(mq[i], buffer, buflen, NULL);
54                         if (n == -1)  {
55                                 if (errno != EAGAIN)  {
56                                         perror("unexpected error");
```

```
57                                      exit (1);
58                              }
59                              /* else, there is no data, try again */
60                      }
61                      else {
62                              x = *((int *)buffer);
63                              if (x == INVALID_INT)
64                                      break;  /* break the for loop */
65                              else {
66                                      fprintf (fp, "%d\n", x);
67                                      printf("rcvd: mq=%d item=%d\n",
68                                             i, x);
69                              }
70
71                      }
72              }
73              if (x == INVALID_INT)
74                      break; /* break the while loop */
75      }
76
77
78      /* clean up the queues. there may be some integers remaining. */
79      for (i = 0; i < 3; ++i) {
80              do {
81                      n = mq_receive(mq[i], buffer, buflen, NULL);
82                      if (n > 0) {
83                              x = *((int *)buffer);
84                              fprintf (fp, "%d\n", x);
85                              printf("rcvd: mq=%d item=%d\n",
86                                     i, x);
87                      }
88              } while (n > 0);
89      }
90
91
92      for (i = 0; i < 3; ++i)
93              mq_close(mq[i]);
94
95
96      fclose (fp);
97      return 0;
98  }
```

Finally, below is the producer program.

```
1   /* -*- linux-c  -*-   */
2
3   /* $Id: producer.c,v 1.7 2015/03/17 08:58:43 korpe Exp korpe $ */
4
5   #include <stdlib.h>
6   #include <stdio.h>
7   #include <string.h>
8   #include <mqueue.h>
9   #include <unistd.h>
```

```
10  #include <errno.h>
11  #include <fcntl.h>           /* For O_* constants */
12  #include <sys/stat.h>        /* For mode constants */
13  #include "common.h"
14
15  extern int errno;
16
17  int main(int argc, char **argv)
18  {
19          mqd_t mq[3];
20          char mq_name[128];
21          int n;
22          int x;
23          int i;
24          int ret;
25          FILE *fp;
26
27          if (argc != 2) {
28                  printf ("usage: producer <infilename>\n");
29                  exit (1);
30          }
31
32          fp = fopen (argv[1], "r");
33          if (fp == NULL) {
34                  printf ("can not open file %s \n", argv[1]);
35                  exit (1);
36          }
37
38
39          for (i = 0; i < 3; ++i) {
40                  sprintf (mq_name, "%s%d", MQNAME_PREFIX, i);
41                  mq[i] = mq_open(mq_name, O_RDWR | O_NONBLOCK);
42                  if (mq[i] < 0) {
43                          perror("can not open msg queue\n");
44                          exit(1);
45                  }
46          }
47
48
49          while (1) {
50                  ret = fscanf (fp, "%d", &x);
51                  if (ret != 1) {
52                          x = INVALID_INT; /* end of file marker */
53                          i = 0;
54                  }
55                  else {
56                          i = x % 3;
57                          // we are assuming integers are positive */
58                  }
59
60                  do  {
61                          n = mq_send(mq[i], (char *) &x, ITEMSIZE, 0);
62                          if (n == -1) {
63                                  if (errno != EAGAIN) {
64                                          perror("mq_send failed\n");
65                                          exit(1);
```

```
66                                  }
67                                  /* else, buffer full, try again */
68                          }
69                          else {
70                                  if (x != INVALID_INT)
71                                          printf ("sent: mq=%d item=%d\n",
72                                                  i, x);
73                          }
74
75                  } while (n == -1);
76                  /* here we may be stuck at the same queue;
77                     this is like a blocking call;
78                     but it is ok;
79                     we could put the number into a list and
80                     retrieve another
81                     int from file.
82                  */
83
84
85                  if (ret != 1)
86                          break;
87          }
88
89          for (i = 0; i < 3; ++i)
90                  mq_close(mq[i]);
91
92          fclose (fp);
93
94          return 0;
95  }
```

We can invoke the programs as follows. In this particular application, we need to run the consumer program first.

```
1  consumer outfile.txt
```

```
1  producer infile.txt
```

2.5 Adding a System Call to Linux Kernel

Adding a new system call to kernel is a rare occassion. We should not immediately implement a new system call whenever we need to have something new to be done by the kernel. But if a service is needed by many applications so that it worths to have a corresponding system call to access it, then we can implement that service as a system call. Otherwise, we can use modules.

Here we will describe how to implement and add a new system call to the Linux kernel. This is a good way of starting touching to the kernel. The exact steps of adding a new system call is different from OS to OS and also may be different from version of an OS to another version of the same OS. Similarly, depending on the version of the Linux kernel, the exact steps may be little bit different. But the basic steps are similar most of the time. What changes is the location of the files (and their names) to be modified, for most of the time.

Below, we describe the steps that are valid for and tested on a Linux kernel of version 2.6.28 running on an 32-bit Intel machine (x86 architecture). The kernel source had been downloaded from www.kernel.org.

While doing the steps below, we assume that you downloaded a kernel source tree and you are in its root directory. Some of the sub-directories that you should be seeing at that directory are: include, fs, drivers, kernel, net, ipc, arch, etc. Assume the name of the system call to be added is `newcall`. It will just take one integer parameter. It will add 50 to that paramater and return the sum. The return type is long. So, if we call the system call with an argument 30, the return value should be 80.

Below are the steps:

1. First we will modify the system call table. We should change into directory: `arch/x86/kernel`. There, we will modify the file `syscall_table_32.S`. Open that file and add

```
1 .long sys\_newcall
```

   to the end of the file. (In previous kernels, the file to modify was syscall_table.S located in arch/i386/kernel/syscall_table.S).

2. Then we will modify the file `unistd_32.h`. That file is located in directory arch/x86/include/asm. (In previous kernels, that file to modify was unistd.h located in /include/asm-i386/unistd.h). There we have the system call numbers defined, such as `#define __NR_read 3`. Find the last such definion, hence the last system call number assigned. You will assign one larger number than that to your system call. For example, if the last system call has a number 332, then we will assign number 333 to our system call. We add the following line to the end of such definitions in that file:

```
1 #define __NR_newcall 333
```

3. We will now add a declaration for our system call in a file called `syscalls.h`. That file is located in include/linux. Add the following line to the file as the last system call declaration:

```
1 asmlinkage long sys\_newcall(int i)
```

4. We will now modify the `Makefile`in the root directory of the source tree. Open the Makefile sitting there and add `newcall/` to `core-y` in the Makefile. After this we will have a line similar to the following in the Makefile:

```
core-y        := usr/ newcall/}
```

the above line should be in the following context:

```
# Objects we will link into vmlinux / subdirs we need to visit
init-y             := init/
drivers-y    := drivers/ sound/ firmware/
net-y              := net/
libs-y              := lib/
core-y             := usr/ newcall/
endif # KBUILD_EXTMOD
```

5. In the same directory that you have modified the Makefile, create a new sub-directory called `newcall`. Change into it. Edit a file called `newcall.c`. That will include your system call code in C. The implementation in that file may be as follows:

```
#include <linux/linkage.h>

asmlinkage long sys_newcall (int n)
{

        return ( n + 50);

}
```

6. In the same directory (in directory newcall/) create a `Makefile` that will have a single line:

```
obj-y:=newcall.o
```

So this is it, as the required kernel modifications. You now need to compile and install the kernel again. Reboot your machine with the new kernel that includes the new system call.

Now, we will write a simple program (test application) that will call our new system call. In your home directory, somewhere, create a subdirectory `test`. Change into it. There you will create a file `test.c`. Edit that file to include the following as the test code.

```
#include <sys/syscall.h>   /* including sys/syscall.h */
#include <stdio.h>         /* including stdio.h       */
#include <linux/unistd.h>  /* including linux/unistd.h */

#define __NR_newcall 333

int main ()
```

```
8  {
9          int x;
10
11         printf ("testing the new system call\n\n");
12
13
14         x = (int) syscall ( __NR_newcall, 30);
15
16         printf ("system call returned; return value is: %d\n", x);
17
18         return (0);
19 }
```

Here, in this program, the function syscall() is a function that can be used to call a system call. You can learn more about using the man page of it. Type: man syscall. It takes as parameters: the number of the system call, and its arguments. If we use this function, we do not specify the name of our new system call (the name was `newcall`), but we just use its number. Therefore, such a way of calling a system call is not very convenient for application programmers. We usually would like to use the name of the system call to call the system call, not its number. There are some macros available (like _syscall0 or _syscall1, etc.) in early versions of Linux.

We can now compile our program as:

```
1  gcc -o test test.c}
```

We can now run the program. Type `./test`. If we see 80 as the integer output, that means we could call the system call successfully and the system call is working successfully.

Here, our system call was very simple. More complex system calls may be written. Such a system call can write something into the address space of the process calling the system call. For that, we can use the kernel functions `copy_from_user()` or `copy_to_user()`. They move data between kernel and user space. For example, after obtaining some information inside the kernel, the system call can copy it to a user space variable via the copy_to_user function.

We can also have the system call to print something to the console or to a log file. For that we use the `printk` kernel function. We can not use printf since kernel is not linked with standard C library where printf is.

Note that printk will send the output to a (log) file if we are not working on a console, but working on a Windowing environment like KDE or GNOME. The name of that file is usually `/var/log/messages`. You should look to the end of it to see if something is printed. We can use the `tail` command to see the end of a file.

2.6 Some Useful Linux Commands to Search Kernel Code

You may find the following Linux commands very useful:

`grep`: This can be used to search for a keyword inside a file or a set of files. Example: `grep Bilkent */*` will search in all sub-directories and their files and all files in the current directory for a keyword `Bilkent`.

`find . -name afilename -print`: This can be used to find the location (path) of a file named `afilename` starting in the current directory and recursively looking to all possible subdirectories and printing out the pathnames of the directories that contain the file.

Chapter 3

Building a new Kernel

3.1 Building a new Linux kernel

Here we provide some information about how to compile, build and run a new Linux kernel. Note that the steps may change slightly or substantially from distribution to distribution. Additionally, steps may change from one kernel version to another kernel version as well. Therefore, most of time, we may need to search for a documentation explaining how kernel is rebuild for our specific Linux distribution and its version.

The following are the instructions to compile and install a kernel on an Ubuntu Desktop 9.10 Karmic Koala i386 32-bit system. We thank our TA Cem Mergenci for providing most of the information here.

1. Go to kernel.org where you can find kernel source codes. Locate source code for latest version, for example it was v2.6.33.2 at the time this document is written. You can download it from:

   `http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.33.2.tar.bz2}`

2. Create a directory named "kernel" on Desktop and extract the archive into this directory. Directory "kernel" should now have a sub directory "linux-2.6.33.2".

3. Open up a terminal window by following Applications Menu ¿ Accessories ¿ Terminal.

4. In order to build and install a kernel we need some libraries that are not found on standard installation. Necessary software and libraries can easily be installed by Package Manager of Ubuntu (APT). Execute following commands and enter your password, since installation requires super user privileges (thus, sudo command at the beginning):

```
1  sudo apt-get install fakeroot build-essential makedumpfile libncurses5 libncurses5-dev kernel-package
2  sudo apt-get build-dep linux
```

5. Execute following commands to copy current kernel configuration to source code directory, so that we have a base configuration.

```
1  cd ~/Desktop/kernel/linux-2.6.28
2  cp /boot/config-`uname -r` .config
```

6. Before using the configuration file, we should update it to include latest options. Execute following command and accept any default options suggested by pressing Enter. Hint: There will be lots of questions.

   make oldconfig

7. In order to change configurations to our needs, we are going to modify .config file we copied by using menuconfig. Execute following command:

   make menuconfig

8. Navigate to the "Load" option. Load the .config file you have in your root directory of the kernel source tree. Then, if you wish, navigate through titles to see what kind of options are there and which ones are included to be compiled into the kernel. You can turn off Kernel hacking > Kernel debugging > Compile Kernel with debug info. It is known to cause resulting kernel package to be much larger. "Save" and exit after you have completed your modifications, if any.

   REMARK: Due to a bug in kernel-package in Karmic we need to modify /Desktop/kernel/linux-2.6.33.2/debian/ruleset/misc/version_vars.mk Locate the line having UTS_RELEASE_HEADER and change it to look like:

```
UTS_RELEASE_HEADER=$(call doit,if [ -f include/generated/utsrelease.h ]; then
\
                        echo include/generated/utsrelease.h; \
                else \
                                echo include/generated/version.h ;
\
                fi)
```

9. Building takes long time. If you have multi-core CPU you can take advantage of parallelization. Instead of number 3, write 1 plus the number of cores you have (3=1+2 for double core). Execute following command:

   export CONCURRENCY_LEVEL=3

10. We are going to make a clean build, from scratch. Execute following commands:

```
1  make-kpkg clean
2  fakeroot make-kpkg --initrd --append-to-version=cs342 kernel-image kernel-headers
```

REMARK: Note that you do not have to make a clean build everytime you make a modification to the source. That is why we are using "make".

11. After a long building phase you can find necessary packages under "kernel" directory. Go one directory up:

    cd ..

12. Install new kernel by following commands:

```
1  sudo dpkg -i linux-image-2.6.33.2cs342_2.6.33.2cs342-10.00.Custom_i386.deb
2  sudo dpkg -i linux-headers-2.6.33.2cs342_2.6.33.2cs342-10.00.Custom_i386.deb
```

13. Those who are using Wubi have to look at [4] to overcome boot problems. If you cannot boot at all, you can use Ubuntu Live CD and get help from the Internet.

14. Congratulations, you have compiled and installed your first custom kernel! Restart the computer and now you can boot your computer with the new kernel by choosing it from boot loader menu. However, do not use it other than for experimental purposes. You can learn kernel release in execution with:

    uname -r

    REMARK: In order to make some simple changes to bootloader you can use StartUpManager:

    sudo apt-get install startupmanager

3.1.1 References

- Kernel/Compile - Community Ubuntu Documentation: "https://help.ubuntu.com/community/Kernel/Co
- KernelTeam/GitKernelBuild - Ubuntu Wiki "https://wiki.ubuntu.com/KernelTeam/GitKernelBuild"
- Ubuntu Forums - Master Kernel Thread

  `http://ubuntuforums.org/showpost.php?p=8688831&postcount=1403`

- Boot Problems:Wubi 9.10 "http://sourceforge.net/apps/mediawiki/bootinfoscript/index.php?title=Boot_
- How to compile a custom kernel for Ubuntu Intrepid: "http://blog.avirtualhome.com/2008/10/28/how-to-compile-a-custom-kernel-for-ubuntu-intrepid/"

Chapter 4

Processes and Threads

4.1 Processes

A process is a program in execution. A new process can be created using fork() or clone() system calls in Linux. If you use the fork() system call in a process, it will create another process which is called a child procss. A child process has its own address space that is duplicated initially from the parent's address space. Then, the parent and child go their own ways.

The fork() system call, when invoked in a (parent) process, will return the process id (pid) of the created child to the parent process. Process id is always positive. The fork() system call will return the value 0 to the child process. The statements after the fork call will be executed by both parent and child processes.

By checking the return value of fork, we can have some statements to be executed by the parent process and some statements to be executed by the child process and some statements to be executed by both.

Below we provide a sample program that shows how we can create a new process using the fork() system call.

```
1  /*    -*- linux-c -*- */
2  /* $Id: process.c,v 1.3 2015/02/26 13:00:13 korpe Exp korpe $ */
3
4  #include <sys/types.h>
5  #include <unistd.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <fcntl.h>
10 #include <errno.h>
```

```
#include <sys/wait.h>


int main(int argc, char *argv[])
{
        int count, i;
        pid_t apid1, apid2;
        FILE *fp;
        char filename[50];

        if (argc != 2) {
                printf("wrong number of arguments \n");
                printf("usage: process <numberofintegers> \n");
                exit(1);
        }

        count = atoi(argv[1]);

        apid1 = fork();
        if (apid1 < 0) {
                perror("main():");
                exit(1);
        }

        if (apid1 == 0) {
                printf("this is first child, pid=%d\n",
                       (int) getpid());
                fflush(stdout);

                // open a file and write something to it.

                sprintf(filename, "%d.txt", (int) getpid());
                fp = fopen(filename, "w");
                if (fp == NULL) {
                        perror("main():");
                        exit(1);
                }

                for (i = 1; i <= count; i++)
                        fprintf(fp, "%d\n", i);

                fclose(fp);

                printf("child terminating, bye...\n");

                exit(0);

        } else {
                printf("this is the parent, pid = %d\n",
                       (int) getpid());

                // create the second child
                apid2 = fork();
                if (apid2 == 0) {
                        // this is second child
                        printf ("this is second child, pid=%d\n",
```

```
67                                  (int) getpid());
68                          execlp ("/bin/ps", "ps", "aux", NULL);
69
70                  }
71                  else {
72                          // wait for the first child to terminate
73                          printf("parent is waiting for first child\n");
74                          waitpid(apid1, NULL, 0);
75
76                          printf("first child finished. parent terminating\n");
77                          exit(0);
78                  }
79
80          }
81
82          return 0;
83  }
```

The program takes one command line parameter. The parameter indicates the number of integers to be written into a file that will be created by one of the child processes that the program will create. The second child process will call the execlp() function after it is being created and in this way execute a new program ("ps aux").

Below is a Makefile that can be used to compile the program.

```
1  process: process.c
2           gcc -g -Wall -o process process.c
3  clean:
4          rm -fr process *~ *.txt core*
```

4.2 Threads

A thread is another execution/control sequence (execution flow) in a process. Every process has at least one sequence of instructions executed, hence every process has at least one thread. We can simply call this as the main thread. A multi-threaded program, that is developed in an environment supporting multi-threading, can create more threads (besides the main thread) that will run concurrently together with the main thread. It is up to the application programmer how to divide the application logic into concurrent activities (where to create threads and what to do in each thread). We may need to synchronize the threads as well.

The POSIX Pthread API (library) provides a set of thread related functions that can be used by an application programmer to create and use threads.

Below is a simple program that creates one new thread (besides the main thread that every program has).

```
1  /*    -*- linux-c -*- */
2  /* $Id: sum.c,v 1.1 2015/02/27 11:38:06 korpe Exp korpe $  */
3
4  #include <errno.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10 #include <string.h>
11
12 int sum = 0;
13
14 struct param {
15         int upper;
16         /* we can put more parameters here */
17 };
18
19
20 static void *summation(void *arg)
21 {
22         struct param *p;
23         int i;
24
25         p = (struct param *) arg;
26
27         for (i = 1; i <= p->upper; ++i)
28                 sum = sum + i;
29
30         pthread_exit(NULL);
31 }
32
33 int main(int argc, char *argv[])
34 {
35         pthread_t tid;
36         struct param par;
37         int ret;
38
39         if (argc != 2) {
40                 printf
41                     ("usage: sum <uppervalue> \n");
42                 exit(1);
43         }
44
45         par.upper  = atoi(argv[1]);        /* upper value */
46
47         ret = pthread_create(&(tid),
48                              NULL,
49                              &summation,
50                              (void *) &par);
51         if (ret != 0) {
52                 printf("thread create failed \n");
53                 exit(1);
54         }
55
56                ret = pthread_join(tid, NULL);
```

```
57          printf ("sum is: %d\n", sum);
58          return 0;
59  }
```

Below is a Makefile to compile this program.

```
1   # $Id: Makefile,v 1.2 2009/03/07 22:58:05 korpe Exp korpe $
2
3   all: thread  sum
4
5   thread: thread.c
6           gcc -g -Wall -o thread thread.c  -lpthread
7
8   sum: sum.c
9           gcc -g -Wall -o sum sum.c -lpthread
10
11  clean:
12          rm -fr *~ thread *.txt core* sum
```

Below, we show another example program that uses POSIX threads API to create many threads.

```
1   /*    -*- linux-c -*- */
2   /* $Id: thread.c,v 1.3 2015/02/27 11:38:02 korpe Exp korpe $     */
3
4   #include <errno.h>
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <pthread.h>
8   #include <sys/types.h>
9   #include <unistd.h>
10  #include <string.h>
11
12
13  #define MAXTHREADS  20                /* max number of threads */
14  #define MAXFILENAME 50                /* max length of a filename */
15
16  /*
17     thread function will take a pointer to this structure
18  */
19  struct arg {
20          int n;                        /* min value */
21          int m;                        /* max value */
22          int t_index;                /* the index of the created thread */
23  };
24
25
26  /* this is function to be executed by the threads */
27  static void *do_task(void *arg_ptr)
28  {
```

```
29          int i;
30          FILE *fp;
31          char filename[MAXFILENAME];
32
33          printf("thread %d started\n", ((struct arg *) arg_ptr)->t_index);
34
35          sprintf(filename, "output_of_thread%d.txt",
36                  ((struct arg *) arg_ptr)->t_index);
37
38          fp = fopen(filename, "w");
39          if (fp == NULL) {
40                  perror("do_task:");
41                  exit(1);
42          }
43
44          for (i = ((struct arg *) arg_ptr)->n;
45               i <= ((struct arg *) arg_ptr)->m; ++i)
46                  fprintf(fp, "integer = %d\n", i);
47
48          fclose(fp);
49
50          pthread_exit(NULL);
51  }
52
53
54
55
56  int main(int argc, char *argv[])
57  {
58          pthread_t tids[MAXTHREADS];        /*thread ids*/
59          int count;                        /*number of threads*/
60          struct arg t_args[MAXTHREADS];        /*thread function arguments*/
61
62          int i;
63          int ret;
64
65          if (argc != 4) {
66                  printf
67                      ("usage: thread <numthreads> <minvalue> <maxvalue>\n");
68                  exit(1);
69          }
70
71          count = atoi(argv[1]);        /* number of threads to create */
72
73          for (i = 0; i < count; ++i) {
74                  t_args[i].n = atoi(argv[2]);
75                  t_args[i].m = atoi(argv[3]);
76                  t_args[i].t_index = i;
77
78                  ret = pthread_create(&(tids[i]),
79                                       NULL, do_task, (void *) &(t_args[i]));
80
81                  if (ret != 0) {
82                          printf("thread create failed \n");
83                          exit(1);
84                  }
```

```
85                  printf("thread %i with tid %u created\n", i,
86                         (unsigned int) tids[i]);
87          }
88
89
90          printf("main: waiting all threads to terminate\n");
91          for (i = 0; i < count; ++i) {
92                  ret = pthread_join(tids[i], NULL);
93                  if (ret != 0) {
94                          printf("thread join failed \n");
95                          exit(0);
96                  }
97          }
98
99
100         printf("main: all threads terminated\n");
101         return 0;
102 }
```

The program first includes the `pthread.h` header file which has the declarations of a set of thread related functions that you can use. To learn more about what that interface file contains, you can use the man command as `man pthread.h`.

The program expects three command-line parameters: 1) number of threads to be created; 2) a min value; 3) a max value. Then the program (the main thread) creates those many threads. Each thread opens a file for writing and writes into that file all the integers in the range [minvalue, maxvalue]. The main thread waits until all threads terminate. Then the main thread terminates as well.

A new thread is created using the pthread_create function. When you create a new thread, a new scheduleable entity, like a process, is created. The new thread starts at the function that you specify as one of the arguments to the pthread_create function. Then, the specific function starts running (and may call other functions while it is running) together with all the threads that have been created earlier. When the function is finished, the thread terminates. We terminate a thread by calling pthread_exit() function. It can take a pointer to a variable that can store a return/exit reason as an integer value. Then that value can be passed to another thread that is waiting in a pthread_join call.

A thread may affect global variables and in this way some other functions and threads can get someting from this thread (exchange data). Threads can access global variables and share them easily. Hence, data sharing and communication between threads is very easy.

A thread start function (i.e., a function that will be executed first by a newly created thread) always expects a single parameter which is of type “void *”. For example, the following is a thread start function. It has one parameter. The return value of a thread start function is always of type “void *” as well.

```
1  static void *do\_task(void *arg\_ptr)}
```

Hence, while creating a thread, we can only pass one parameter to the thread start function. If we want to pass a lot of values, we place them into a structure and pass to the thread start function a pointer to the structure while creating the thread. In this way, the thread start function can use the passed pointer to reach to the structure and obtain the parameter values.

Below is a Makefile that can be used to compile a multi-threaded program. Note that we are linking our program with pthread library. For that we use the `-lpthread` compiler option.

```
1  # $Id: Makefile,v 1.2 2009/03/07 22:58:05 korpe Exp korpe $
2
3  all: thread  sum
4
5  thread: thread.c
6          gcc -g -Wall -o thread thread.c  -lpthread
7
8  sum: sum.c
9          gcc -g -Wall -o sum sum.c -lpthread
10
11 clean:
12         rm -fr *~ thread *.txt core* sum
```

4.3 Applications Using Multiple Process or Threads - Examples

We now show you an example application and its implementation. This application will use multiple processes or threads to do a task, which is sorting integers stored in a number of files. We will first show how we can implement this application using multiple concurrent processes. Then we will show how we can implement it using multiple concurrent threads.

Hence, we will write two programs, one using multiple processes, the other one using multiple threads.

4.3.1 Application 1 - Processes

Specification of the Application

The specification of the first application (program) is as follows. The name of our program will be proces_sort and it will sort N input files using multiple processes. The main program (parent process) will create N child processes (sorter processes) and one merger process. Each child process will read one input file and will sort the integers there into a linked list.

Then those sorted integers will be sent to a merger process via a message queue. Hence there will be one seperate message queue between each sorter process and the merger process; a total of N message queues will be created. The merger process will read the messages containing sorted integers arriving from the N different sorter proccesses. While reading the integers the merger process will merge those integers into a single sorted output. The output will go into an output file. The output file will contain the integers in sorted order; one integer per line.

Each input file will contain positive integers in any order. We can have duplicates. There will be one integer per line. Therefore, the outputfile can also have duplicates.

The program will be invoked as follows:

```
process_sort -n N ifile1 ifile2 ... ifileN -o ofile
```

For example:

```
process_sort -n 3 file1.txt file2.txt file3.txt -o out.txt}
```

In this program, a sorter process will just write (send) into its respective message queue. A merger process will just read (receive) from one of the N message queues.

If we wish, we can pack a number of integers into a single message that will be sent from a sorter process to the merger process. But in our implementation here we are sending one integers per message, altought it is not very efficient.

The main (parent process) will just be responsible from the creation of N sorter processes and one merger process. Then it has to wait until all these children do their job and terminate. Then it will write a message to the screen indicating that the sort operation is complete. It can then terminate.

We will use POSIX message queues to implement the program.

Implementation

We implement our application using a single C file: process_sort.c. The program is shown below. We also provide a makefile that can be used to compile this application and next application.

The program starts at main() function by taking the program arguments into some variables: num_files, filenames, outfilename. The num_files variable keeps number of input files

given to the program to sort. The the main() function then creates num_files (N) message queues. For each message queue created, with use the same prefix, but a different suffix. In this way, we obtain a different name for each message queue. The suffix we add is an index number corresponding to a sorter process. They are indexed from 0 to num_files-1. When a message queue created, we obtain a descriptor to refer to it. The descriptors of the message queues are stored in mq_ids variable.

The the program created the merger process (using fork()) and sorter processes. We store the process IDs of the ceated process in variables: mergerpid and sorterpid[] array. They will be used to wait for the created (child) processes to terminate using the waitpid () system call. When those processes terminate, the parent process closes and removes the message queues.

When merger process is created, it starts executing the merger() function. The merger() function takes as arguments: the name of the output file, the number of files (message queues), and an array of messase queue ids. The merger function first allocated storage for a buffer to hold an incoming message. The size of that buffer is at least the size of the largest message that can be put into a message queue. That size is obtained using the mq_getattr() message queue function. The allocated storage for the buffer is pointed by bufptr pointer.

The merger process than applies a merging algorithm to merge integers that are arriving through N message queues. Initially it tries to read one integer from each queue into an array (called head) of ingers of size N. Hence an integer at the front of queue i will be put into the array entry head[i]. Then minimum value in the head array is selected, and put into a variale min. Assume the index of that entry is q. This means queue q has provided the minimum value. That value is the absolute minimum among all integers that will arrive, since the queues will provide sorted integers.

Then another integer is read from the queue that provided the minimum value. That is the queue q in the example above. That means the value read is put into head[q] entry. Then, again, the minimum value is selected among integers in the head array and it output to the file.

This cycle of selecting the min value in the head array, writing it out, and reading a new value from the queue that had provided the min value is repeated in a while loop until all queues will no longer send any data. This will happen when the respective sorter processes have finished and indicated the finish by sending a special integer ENDOFDATA (which is -1 in this program.)

A sorter process, when created, runs the sorter() function and then terminates. The sorter function takes the respective input filename and message queue descriptor as input. Each sorter process will have its own filename and descriptor parameters. This is ensured by the parent process when the sorter() is called. The first that a sorter process does is opening its input file. Then integers are taken one by one from the input file and inserted, using the

add_sorted_list() function, into a sorted list of integers. Hence we are applying an insertion sort algorithm here, which is not very efficient (runs in $O(N^2)$ time) but that is OK. After all integers are added to the sorter list, the sorter starts sending the integers to the merger process using its message queue. Each integer is sent as a different message; this is not efficient, but that is OK for this application. After the last integers is sent, another integer (ENDOFDATA) indicating end of data stream is sent. Then sorter() function returns and the sorter child process terminates.

Below is the process_sort.c program.

```
1  /* -*- linux-c -*-  */
2  /* $Id: process_sort.c,v 1.3 2009/03/13 08:16:10 korpe Exp korpe $  */
3
4  #include <stdlib.h>
5  #include <mqueue.h>
6  #include <stdio.h>
7  #include <unistd.h>
8  #include <errno.h>
9  #include <string.h>
10 #include <sys/types.h>
11 #include <sys/wait.h>
12
13
14 #define DEBUG 0
15
16 #define ENDOFDATA  -1           /* must be a negative number  */
17 #define MAXINT 2000000000       /* assuming a file will not have this */
18 #define MQNAME "/psortmsgqueue"
19 #define MAX_FILES    6
20 #define MAX_FILENAME 128
21
22 struct list_el
23 {
24         int data;
25         struct list_el *next;
26 };
27
28 struct message {
29         int data;
30 };
31
32 void
33 print_sorted_list (struct list_el *listp)
34 {
35         struct list_el *p = listp;
36         while (p) {
37                 printf ("%d\n", p->data);
38                 p = p->next;
39         }
40 }
41
42
43 /* add a new element into the list; insertion sort;
44  not efficient sort */
```

```
45  void
46  add_sorted_list (struct list_el** listpp, int num)
47  {
48          struct list_el *new_el, *cur;
49
50          new_el = (struct list_el *) malloc (sizeof(struct list_el));
51          if (new_el == NULL) {
52                  perror ("malloc failed\n");
53                  exit (1);
54          }
55          new_el->data = num;
56          new_el->next = NULL;
57
58          if ( (*listpp) == NULL) {
59                  *listpp = new_el;
60                  return;
61          }
62          else if (num <= ( (*listpp)->data)) {
63                  new_el->next = *listpp;
64                  *listpp = new_el;
65                  return;
66          } else {
67                  cur = *listpp;
68
69                  while (cur->next)
70                  {
71                          if (num <= ((cur->next)->data))
72                                  break;
73                          else
74                                  cur = cur->next;
75                  }
76
77                  if (cur->next == NULL) {
78                          cur->next = new_el;
79                          return;
80                  }
81                  else {
82                          new_el->next = cur->next;
83                          cur->next = new_el;
84                          return;
85                  }
86          }
87  }
88
89
90
91
92  void
93  sorter(char infile[], mqd_t mq)
94  {
95
96          FILE   *fp;
97          int    number;
98          struct message msg;
99          int    n;
100
```

```
101         struct list_el *slp = NULL;  /* sorted list pointer */
102
103         fp = fopen (infile, "r");
104         while (fscanf (fp, "%d",  &number) == 1) {
105                 add_sorted_list (&slp, number);
106         }
107         if (DEBUG)
108                 print_sorted_list (slp);
109
110
111         /* send the sorted integers; one integer per message;
112            not very efficient*/
113
114         while (slp) {
115                 msg.data = slp->data;
116                 n = mq_send(mq, (char *) &msg, sizeof(struct message), 0);
117                 if (n == -1) {
118                         perror("mq_send failed\n");
119                         exit(1);
120                 }
121                 if (DEBUG)
122                         printf("mq_send success, data = %d\n", msg.data);
123                 slp = slp->next;
124         }
125
126         msg.data = ENDOFDATA;
127         n = mq_send(mq, (char *) &msg, sizeof(struct message), 0);
128         if (n == -1) {
129                 perror("mq_send failed\n");
130                 exit(1);
131         }
132         if (DEBUG)
133                 printf("mq_send success, end data = %d\n", msg.data);
134
135         fclose (fp);
136 }
137
138
139 void
140 merger(char *outfile, int file_count, mqd_t mqarray[])
141 {
142         FILE *fp;
143         struct mq_attr mq_attr;
144         struct message *msgptr;
145         char *bufptr;            /* buffer to hold a new message */
146         int buflen;
147         int head[MAX_FILES];     /* integers at the head of the queues */
148         int min;                 /* next min value to output  */
149         int q;                   /* index of queue having min value */
150         int n;                   /* number of bytes received */
151         int endedq_count = 0;    /* num queues that finished */
152         int i;
153
154         fp = fopen (outfile, "w");
155
156         mq_getattr(mqarray[0], &mq_attr);
```

```
157         if (DEBUG)
158                 printf("mq 0 maximum msgsize = %d\n", (int) mq_attr.mq_msgsize);
159
160         buflen = mq_attr.mq_msgsize;
161         bufptr = (char *) malloc(buflen);
162
163         endedq_count = 0;
164
165         /* try to read one integer from each mq */
166         for (i = 0; i < file_count; ++i) {
167                 n = mq_receive(mqarray[i], (char *)
168                                 bufptr, buflen, NULL);
169                 if (n == -1) {
170                         perror("mq_receive failed\n");
171                         exit(1);
172                 }
173                 msgptr = (struct message *) bufptr;
174                 if (DEBUG)
175                         printf("mq_receive success, msg data=%d\n",
176                                 msgptr->data);
177                 if (msgptr->data == ENDOFDATA) {
178                         head[i] = ENDOFDATA;
179                         endedq_count++;
180                 }
181                 else
182                         head[i] = msgptr->data;
183         }
184
185
186         while (endedq_count < file_count) {
187                 /* select min */
188                 q = -1;
189                 min = MAXINT;
190                 for (i = 0; i < file_count; ++i) {
191                         if (head[i] != ENDOFDATA)
192                                 if (head[i] < min) {
193                                         min = head[i];
194                                         q = i;
195                                 }
196                 }
197
198                 /* output the selected min data */
199                 fprintf (fp, "%d\n", min);
200
201                 /* read from min queue again */
202                 n = mq_receive(mqarray[q], (char *)
203                                 bufptr, buflen, NULL);
204                 if (n == -1) {
205                         perror("mq_receive failed\n");
206                         exit(1);
207                 }
208                 msgptr = (struct message *) bufptr;
209                 if (DEBUG)
210                         printf("mq_receive success, msg data=%d\n",
211                                 msgptr->data);
212                 if (msgptr->data == ENDOFDATA) {
```

```
213                         head[q] = ENDOFDATA;
214                         endedq_count++;
215                 }
216                 else
217                         head[q] = msgptr->data;
218
219         } /* while */
220
221         free (bufptr);
222         fclose (fp);
223 }
224
225
226
227 int
228 main(int argc, char **argv)
229 {
230
231         mqd_t mq;
232         pid_t pid;
233         int i;
234         int num_files;
235         char  filenames[MAX_FILES][MAX_FILENAME];
236         char  outfilename[MAX_FILENAME];
237         char  mq_names[MAX_FILES][MAX_FILENAME];
238         mqd_t mq_ids[MAX_FILES];
239
240         pid_t mergerpid;
241         pid_t sorterpid[MAX_FILES];
242
243         num_files = atoi(argv[2]);
244         for (i = 0; i < num_files; ++i)
245                 strcpy(filenames[i], argv[3+i]);
246
247         strcpy (outfilename, argv[3+num_files+1]);
248
249         /* create msg queues */
250         for (i = 0; i < num_files; ++i) {
251                 sprintf (mq_names[i], "%s%d", MQNAME, i);
252                 if (DEBUG)
253                         printf ("mq_name=%s\n", mq_names[i]);
254                 mq = mq_open(mq_names[i], O_RDWR | O_CREAT, 0666, NULL);
255                 if (mq == -1) {
256                         perror("can not create msg queue\n");
257                         exit(1);
258                 }
259                 mq_ids[i] = mq;
260                 if (DEBUG)
261                         printf("mq %d created, mq id = %d\n", i,
262                                 (int) mq_ids[i]);
263         }
264
265
266         /* create merger process */
267         pid = fork();
268         if (pid == 0) {
```

```
269                 merger(outfilename, num_files, mq_ids);
270                 exit(0);
271         }
272         mergerpid = pid;
273
274
275
276         /* create sorter processes */
277         for (i = 0; i < num_files; ++i) {
278                 pid = fork ();
279                 if (pid == 0) {
280                         sorter(filenames[i], mq_ids[i]);
281                         exit (0);
282                 }
283                 sorterpid[i] = pid;
284         }
285
286
287         /* wait for the merger and sorters to terminate */
288         for (i = 0; i < num_files; ++i) {
289                 waitpid (sorterpid[i], NULL, 0);
290         }
291         waitpid (mergerpid, NULL, 0);
292
293
294         /* remove message queues */
295         for (i = 0; i < num_files; ++i) {
296                 mq_close (mq_ids[i]);
297                 mq_unlink (mq_names[i]);
298         }
299
300
301         printf ("sorting done\n");
302
303         return 0;
304 }
305
306
307
```

Below is the Makefile to compile the program.

```
all: process_sort gen thread_sort

process_sort: process_sort.c
gcc -Wall -o  process_sort  process_sort.c -lrt


thread_sort: thread_sort.c
gcc -Wall -o  thread_sort  thread_sort.c -lrt -lpthread
```

```
gen: gen.c
gcc -Wall -o  gen gen.c -lrt


clean:
rm -fr *~ gen thread_sort process_sort
```

4.3.2 Application 2 - Threads

Specification

This application will do the same thing, this time using threads instead of child processes. There will be N (num_files) number of sorter threads, one merger thread created. They will now communicate using global variables, not message queues. Hence we don't need to use global variables in this application. The program will be invoked as before.

Implementation

When the program will started, the main() function will get the program arguments and store them in some variables. It will initialize a global array (called linkhead[]) of linked lists. There will be N entries used in the array: one entry for each thread. Each linked list will keep sorted integegers read from an input file. Then it creates the threads: first the merger thread and then than the sorter threads. The merger thread runs the merger() function and sorter threads run the sorter() function.

The merger() function takes one argument (a thread start function can only take one argument at most), which is the number of input files. The file_count local variable of merger thread is set to the value of this argument as soon as the threads starts running:

```
file_count = (int) arg;
```

Then the merger() function (hence thread) waits for all threads to terminate. This is achieved by calling the pthread_join function file_count times, each time with the thread id of a different sorter thread created. Those thread IDs are storedin a globla array (called sorters[]) at the time the threads are created.

```
for (i = 0; i < file_count; ++i)
    pthread_join (sorters[i], NULL);
```

When all sorter threads terminate, the merger thread can start executing again. At that time, the sorter threads have sorted the integers in N (file_count) different files into N different linked lists pointed by the listhead[] array. The merger thread will start merging the integers from these N linked lists with the same algorithm described in the previous application. While merging, the integers will be output to the output file. When all integers merged, the output file will contain all integers in sorted order and it will be closed.

Each sorter thread will executed the sorter() function. The sorter() function takes one argument: the index (i.e. number) of the created sorter thread. The index can be a value between 0 and num_files-1. Using this number as an index to the filenames[] and listhead[] arrays, the sorter function will open the respective input file and sort the integers into the respective linked list pointed by the respective entry of the listhead[] array. Then it will call pthread_exit() function and the corrsponding thead will terminated. The sorted data is ready in memory in a linked list pointed by the global listhead[] array entry.

Below is the code for the program `thread_sort.c`.

```
1  /* -*- linux-c -*-  */
2  /* $Id: thread_sort.c,v 1.2 2009/03/13 11:25:51 korpe Exp $  */
3
4  #include <stdlib.h>
5  #include <mqueue.h>
6  #include <stdio.h>
7  #include <unistd.h>
8  #include <errno.h>
9  #include <string.h>
10 #include <sys/types.h>
11 #include <sys/wait.h>
12 #include <pthread.h>
13
14 #define DEBUG 0
15
16 #define ENDOFDATA  -1      /* must be a negative number  */
17 #define MAXINT 2000000000  /* assuming all numbers are less than this */
18 #define MAX_FILES    6     /* max number of input files  */
19 #define MAX_FILENAME 128
20
21 struct list_el
22 {
23         int data;
24         struct list_el *next;
25 };
26
27
28 /***************** global variables ***************/
29 struct list_el*  listheads[MAX_FILES];
30 char             filenames[MAX_FILES][MAX_FILENAME];
31 pthread_t        sorters[MAX_FILES];
32 char             outfilename[MAX_FILENAME];
33 /**************************************************/
34
```

```
35
36  void
37  print_sorted_list (struct list_el *listp)
38  {
39          struct list_el *p = listp;
40          while (p) {
41                  printf ("%d\n", p->data);
42                  p = p->next;
43          }
44  }
45
46
47  void
48  add_sorted_list (struct list_el** listpp, int num)
49  {
50          struct list_el *new_el, *cur;
51
52          new_el = (struct list_el *) malloc (sizeof(struct list_el));
53          if (new_el == NULL) {
54                  perror ("malloc failed\n");
55                  exit (1);
56          }
57          new_el->data = num;
58          new_el->next = NULL;
59
60          if ( (*listpp) == NULL) {
61                  *listpp = new_el;
62                  return;
63          }
64          else if (num <= ( (*listpp)->data)) {
65                  new_el->next = *listpp;
66                  *listpp = new_el;
67                  return;
68          } else {
69                  cur = *listpp;
70
71                  while (cur->next)
72                  {
73                          if (num <= ((cur->next)->data))
74                                  break;
75                          else
76                                  cur = cur->next;
77                  }
78
79                  if (cur->next == NULL) {
80                          cur->next = new_el;
81                          return;
82                  }
83                  else {
84                          new_el->next = cur->next;
85                          cur->next = new_el;
86                          return;
87                  }
88          }
89  }
90
```

```
91
92
93  /* sorter thread function */
94  void *
95  sorter(void *arg)
96  {
97
98          FILE   *fp;
99          int    tnum;   /* index of the thread */
100         int    number; /* a number for the input file */
101
102         tnum = (int) arg;
103
104         struct list_el *slp = NULL;  /* sorted list pointer */
105
106         fp = fopen (filenames[tnum], "r");
107         if (fp == NULL) {
108                 printf ("fopen failed\n");
109                 exit (1);
110         }
111
112         while (fscanf (fp, "%d",  &number) == 1) {
113                 add_sorted_list (&slp, number);
114         }
115         if (DEBUG)
116                 print_sorted_list (slp);
117
118         listheads[tnum] = slp;
119         fclose (fp);
120
121         if (DEBUG)
122                 printf ("thread %d exiting\n", tnum);
123
124         pthread_exit (0);
125 }
126
127
128 /* merger thread function */
129 void *
130 merger(void *arg)
131 {
132         FILE *fp;              /* output filename */
133         int head[MAX_FILES];  /* first integers of eacg list */
134         int min;              /* next min value */
135         int q;                /* list index of the min value */
136         int endedq_count = 0; /* number of lists emptied */
137         int file_count;       /* number of files */
138         struct list_el *temp;
139         int i;
140
141         file_count = (int) arg;
142
143         /* wait all sorters to terminate */
144         for (i = 0; i < file_count; ++i)
145                 pthread_join (sorters[i], NULL);
146
```

```
147
148         if (DEBUG)
149                 printf ("merger: all sorters terminated\n");
150
151         fp = fopen (outfilename, "w");
152         if (fp == NULL) {
153                 perror ("fopen failed\n");
154                 exit (1);
155         }
156         endedq_count = 0;
157
158         /* try to read one integer from each mq */
159         for (i = 0; i < file_count; ++i) {
160                 if (listheads[i] == NULL) {
161                         head[i] = ENDOFDATA;
162                         endedq_count++;
163                 }
164                 else {
165                         head[i] = listheads[i]->data;
166                         temp = listheads[i];
167                         listheads[i] = listheads[i]->next;
168                         free (temp);
169                 }
170
171
172         }
173
174
175         while (endedq_count < file_count) {
176                 /* select min - not very efficient, ok*/
177                 q = -1;
178                 min = MAXINT;
179                 for (i = 0; i < file_count; ++i) {
180                         if (head[i] != ENDOFDATA)
181                                 if (head[i] < min) {
182                                         min = head[i];
183                                         q = i;
184                                 }
185                 }
186
187                 /* output the selected min data */
188                 fprintf (fp, "%d\n", min);
189
190                 /* read from min queue  again */
191
192                 if (listheads[q] == NULL) {
193                         head[q] = ENDOFDATA;
194                         endedq_count++;
195                 }
196                 else {
197                         head[q] = listheads[q]->data;
198                         temp = listheads[q];
199                         listheads[q] = listheads[q]->next;
200                         free (temp);
201                 }
202
```

```
203         } /* while */
204
205         fclose (fp);
206
207         pthread_exit (0);
208 }
209
210
211
212 int
213 main(int argc, char **argv)
214 {
215
216         int       i;
217         pthread_t mergertid;
218         int       num_files;
219
220
221         num_files = atoi(argv[2]);
222         for (i = 0; i < num_files; ++i)
223                 strcpy(filenames[i], argv[3+i]);
224
225         strcpy (outfilename, argv[3+num_files+1]);
226
227
228         /* initialize sorted lists */
229         for (i = 0; i < num_files; ++i)
230                 listheads[i] = NULL;
231
232
233         /* create merger thread */
234         int ret = pthread_create (&mergertid, NULL,
235                                   merger, (void *) num_files);
236         if (ret != 0) {
237                 perror ("thread create failed\n");
238                 exit (1);
239         }
240
241         /* create sorter threads */
242         for (i = 0; i < num_files; ++i) {
243                 int ret = pthread_create (&(sorters[i]), NULL,
244                                           sorter, (void *) i);
245                 if (ret != 0) {
246                         perror ("thread create failed\n");
247                         exit (1);
248                 }
249         }
250
251
252         /* wait for the merger to terminate */
253         pthread_join (mergertid, NULL);
254
255
256         printf ("sorting done\n");
257
258         return 0;
```

```
259 }
260
261
262
```

4.4 Signals

In Unix and Linux, a process or kernel may send a signal to another process. A signal is a notification. It indicates to the receiver process that an event has occured. There is usually a default action taken for a signal by the kernel. If, however, the process has registered a signal handler to the kernel, whenever the corresponding event occurs, the kernel no longer executes the default action, but instead the signal handler function of the process is run when the process is re-started by the kernel scheduler. Hence the signal handler is executed asynchronously.

Below we provide an example program, `signal.c` that illustrates the use of signals. This program registers a signal handler function `sigint_handler` using the signal() system call. The signal handler is to be executed whenever a SIGINT signal is generated and delivered to this process.

The SIGINT signal is sent to a process, for example, whenever the user presses the `<CTRL-C>` key sequence. It can also be sent to a process by another process. The kill program, for example, can be used to send that signal (or another signal) to a process.

In our example program, at the time the SIGINT signal is sent to the process, the process will be most probably in a while loop, looping around (this sample program is just an infinite loop program). When the SIGNAL is delivered to the process, execution will be transferred from the while loop to the signal handler function (sigint_handler()) in the program. In this example the handler function will just print out a message to the screen and call the exit() system call to terminate itself. If we would omit the exit() system call, execution would continue at the while loop after the signal handler function returns.

Below is the program (`signal.c`).

```
1  /* $Id: signal.c,v 1.2 2015/03/04 14:50:47 korpe Exp korpe $ */
2
3  #include <stdio.h>
4  #include <signal.h>
5  #include <stdlib.h>
6
7  static void sigint_handler()
8  {
9          printf("I received SIGINT signal. bye... \n");
10         fflush(stdout);
11         exit(0);
```

```
12 }
13
14
15 int main()
16 {
17         signal(SIGINT, sigint_handler);
18
19         while (1);
20 }
```

You can test this program by first compiling it with the following Makefile.

```
# $Id: Makefile,v 1.2 2009/03/07 23:01:26 korpe Exp korpe $

all: signal

signal: signal.c
gcc -o signal -Wall signal.c
clean:
rm -fr signal *~
```

This will produce an executable program called `signal`. Then you can run this program in a window. It will start an infinite loop. In the same window, you can now press the CTRL-C keys together. In this way you are sending a SIGINT signal to the process. The program execution will jump to the signal handler function and that function will be executed. It will print a message to the screen and will call exit() to ask kernel for its termination.

You can also send signal to a process using the `kill` command. To test that, first start your `signal` program in one window. Then, in another window, first learn the process id (pid) of your running program. For that you can use the `ps aux` command. It will give such an output:

```
1 ...
2 korpe    12567  0.0  0.0   ...    00:23   0:00 sshd: korpe@pts/2
3 korpe    12568  0.0  0.0   ...    00:23   0:00 -bash
4 korpe    12780  101  0.0   ...    00:27   0:07 ./signal
5 korpe    12782  0.0  0.0   ...    00:27   0:00 ps aux
6 ...
```

Here, in one line we see the pid of the `signal` process, which is 12780. Then, we can use that pid to send a signal to the process with the kill command. Type the following command on the same window and hit the return key.

```
kill -s SIGINT 12780
```

With that we sent a SIGINT signal to our process signal that was running in a while loop. The kill program sent the signal. As soon as the process signal receives the signal (via the kernel), it will run the signal handler. The signal handler will print a message to the screen and will terminate the program. We can see such an output below:

```
1 korpe@pckorpe:~$ ./signal
2 I received SIGINT signal. bye...
3 korpe@pckorpe:~$
```

This shows that we started the `signal` program and it was looping in a while loop. Then, a signal is delivered to the process and the signal handler of the process is run and printed out the message "I received SIGINT signal. bye..." to the screen and terminated the program.

If you want to write a program that is sending a signal to another program, you can use the kill() system call. The kill() system call can be used to send a signal to another process. You can learn more about kill() system call from its man page. For that just type: `man 2 kill`. This tells that the kill() system call is in section 2 of the man pages (section 2 contains the system calls). Since there is also kill program about which you may want to learn more, you should specify the section number. If you want to learn more about the kill program, you type: `man 1 kill`.

Chapter 5

Interprocess Communication

5.1 POSIX Message Queues

Processes can communicate with each other by using message queues in Linux. You can use the man pages to learn about the message queue interface (POSIX message queue API). Just type: `man mq_overview` on your shell and read the related help page.

Here, we provide an example application that uses a message queue between two processes to pass information. The application is the producer-consumer application. The producer will produce items that will be passed to the consumer which will consume the items. The items will be passed through the message queue.

The consumer program will create the message queue. Hence, it should be run first. The producer process, when started, will open the message queue that was created earlier.

The two programs can share some definitions. We put them into a header file called `shareddefs.h`. This header file is shown below. It includes the definition of a C structure that will represent an item. It also includes the name of the message queue that will be created. That name can be any valid file name (starting with "/"). Producer and consumer programs will use that name to refer to the same message queue.

```
1 /* $Id: shareddefs.h,v 1.1 2009/03/06 18:20:32 korpe Exp $ */
2
3
4 struct item {
5         int id;
6         char astr[64];
7 };
8
9
```

```
10 #define MQNAME "/justaname"
```

Below, we show the producer program. It opens the message queue by calling the `mq_open` system call (a library function in fact, which in turn calls the respective system call) Then, in a while loop, it generates and sends messages into the message queue. To send a message, we use the `mq_send` function. Between two send operations we call the sleep function to sleep for a while.

```
1  /* $Id: producer.c,v 1.2 2015/03/04 13:57:02 korpe Exp korpe $ */
2
3  #include <stdlib.h>
4  #include <mqueue.h>
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <errno.h>
8  #include <string.h>
9
10 #include "shareddefs.h"
11
12 int main()
13 {
14
15         mqd_t mq;
16         struct item item;
17         int n;
18
19
20         mq = mq_open(MQNAME, O_RDWR);
21         if (mq == -1) {
22                 perror("can not open msg queue\n");
23                 exit(1);
24         }
25         printf("mq opened, mq id = %d\n", (int) mq);
26
27
28         int i = 0;
29
30         while (1) {
31                 item.id = i;
32                 strcpy(item.astr, "operating system is good\n");
33
34                 n = mq_send(mq, (char *) &item, sizeof(struct item), 0);
35
36                 if (n == -1) {
37                         perror("mq_send failed\n");
38                         exit(1);
39                 }
40
41                 printf("mq_send success, item size = %d\n",
42                        (int) sizeof(struct item));
43                 printf("item->id   = %d\n", item.id);
44                 printf("item->astr = %s\n", item.astr);
45                 printf("\n");
```

```
46
47                  i++;
48
49                  sleep(1);        /* sleep one second */
50                  // you can remove sleep if you wish
51          }
52
53
54          mq_close(mq);
55          return 0;
56  }
```

Next, we have the program for the consumer process.

```
1   /* $Id: consumer.c,v 1.2 2015/03/04 13:57:07 korpe Exp korpe $ */
2
3   #include <stdlib.h>
4   #include <mqueue.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <errno.h>
8   #include <string.h>
9
10  #include "shareddefs.h"
11
12  int main()
13  {
14
15          mqd_t mq;
16          struct mq_attr mq_attr;
17          struct item *itemptr;
18          int n;
19          char *bufptr;
20          int buflen;
21
22          mq = mq_open(MQNAME, O_RDWR | O_CREAT, 0666, NULL);
23          if (mq == -1) {
24                  perror("can not create msg queue\n");
25                  exit(1);
26          }
27          printf("mq created, mq id = %d\n", (int) mq);
28
29          mq_getattr(mq, &mq_attr);
30          printf("mq maximum msgsize = %d\n", (int) mq_attr.mq_msgsize);
31
32          /* allocate large enough space for the buffer */
33          buflen = mq_attr.mq_msgsize;
34          bufptr = (char *) malloc(buflen);
35
36          while (1) {
37                  n = mq_receive(mq, (char *) bufptr, buflen, NULL);
38
39                  if (n == -1) {
40                          perror("mq_receive failed\n");
```

```
41                          exit(1);
42                  }
43
44                  printf("mq_receive success, message size = %d\n", n);
45
46                  itemptr = (struct item *) bufptr;
47
48                  printf("item->id = %d\n", itemptr->id);
49                  printf("item->astr = %s\n", itemptr->astr);
50                  printf("\n");
51
52          }
53
54          free(bufptr);
55          mq_close(mq);
56          return 0;
57  }
```

The consumer program first creates a message queue with the name specified in the shared-defs.h header file. This name is shared by the producer and consumer processes. We use the `mq_open` function to create a message queue as well. After creating the message queue, the consumer learns about some of the properties of the message queue by calling the mq_getattr() function. In this case we are learning the maximum size of a message that kernel can support. Then we allocate that much space for our local buffer to put an incoming item (message) from the producer. Our local buffer is just a character (byte) array pointed by bufptr. We use the malloc() function to allocate memory to be used as the buffer.

We can compile these programs and obtain the respective executable files by using the following Makefile.

```
1   # $Id: Makefile,v 1.1 2009/03/06 18:20:28 korpe Exp $
2
3   all: producer consumer
4
5   consumer: consumer.c
6           gcc -Wall -o consumer consumer.c -lrt
7
8
9   producer: producer.c
10          gcc -Wall -o producer producer.c -lrt
11
12  clean:
13          rm -fr *~ producer consumer
```

After editing such a Makefile, we just need to type `make` to compile both of the programs and obtain two executable files, i.e., programs: `producer` and `consumer`. Then, in one window, you can run the producer program and in another window you can run the

consumer program. To run the consumer, you can just type the following in one window: `consumer` or `./consumer`

The consumer will create a message queue and will wait on it for the arrival of a message.

To run the producer, we can type the following in another window: `producer` or `./producer`

Producer will start generating items every one second and will send them into message queue.

The consumer process will start running and retrieving the items from the message queue. It will print out the content of each item to the screen. Producer and consumer processes will run indefinitely until you terminate them (by a program like the `kill` command). You can type `kill -9 pid` to kill a process whose process id is pid.

Note that in the Makefile while compiling the programs we use an option `-lrt`. That means we have to link our program with the real-time library, `librt`. The is because the message queue related API functions are implemented in the rt library.

5.2 POSIX Shared Memory

POSIX shared memory API includes a set of functions that you can use to create and use a shared memory region (segment) among processes.

Note that even though all processes are in memory and sharing the RAM chip, they normally sit in non-overlapping regions of physical memory. This is because each process has its own address space and the address spaces of processes are mapped to different regions of physical memory. When a shared segment is created, however, the processes can share a portion of their address spaces and in this way can all access the portion of physical memory.

We start by showing a simple two-process application. One process creates shared memory segment of size 512 bytes and puts 100 small integers there (each integer is 1 byte). The other process attaches to the shared memory segment and reads those integers and prints them out. To keep the programs short, we are not doing any error checking, which must be done in a robust implementation.

Below is the code of the process putting integers to the shared segment.

```
1 /* -*- linux-c -*- */
2 /* $Id: simple_p.c,v 1.1 2015/03/04 12:00:03 korpe Exp korpe $ */
3
4 #include <unistd.h>
```

```
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
13
14 int
15 main(int argc, char **argv)
16 {
17         int fd, i;
18         void *sptr;
19         char *p;
20
21
22         fd = shm_open("/afilename", O_RDWR | O_CREAT, 0660);
23         ftruncate(fd, 512);
24         sptr = mmap(NULL, 512,
25                         PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
26         close(fd);
27         p = (char *)sptr;
28
29         for (i=0; i<100; ++i) {
30                 p[i] = 10 + i;
31         }
32
33
34         exit(0);
35 }
```

Below is the code of the process retrieving the integers from the shared segment and printing them out.

```
1  /* -*- linux-c -*- */
2  /* $Id: simple_c.c,v 1.1 2015/03/04 12:00:03 korpe Exp korpe $ */
3
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
13
14 int
15 main(int argc, char **argv)
16 {
17         int fd, i;
18         void *sptr;
```

```
19          char *p;
20          char c;
21          struct stat sbuf;
22
23          fd = shm_open("/afilename", O_RDWR, 0660);
24          fstat(fd, &sbuf);
25          sptr = mmap(NULL, sbuf.st_size,
26                      PROT_READ | PROT_WRITE, MAP_SHARED,
27                      fd, 0);
28          close(fd);
29          p = (char *)sptr;
30
31          for (i=0; i<100; ++i) {
32                  c = p[i];
33                  printf ("%d ", c);
34          }
35          printf ("\n");
36
37          shm_unlink ("/afilename");
38
39          exit(0);
40  }
```

Below is a Makefile to compile them. The same Makefile is used to compile the next application we show. Note the we use the -lrt compiler option to link the programs with the real-time (rt) library that implements the shared memory API for Linux.

```
1   # $Id: Makefile,v 1.1 2015/03/04 12:01:25 korpe Exp korpe $
2
3   all: producer consumer simple_p simple_c
4
5
6   consumer: consumer.c
7           gcc -Wall -g -o consumer consumer.c -lrt
8
9
10  producer: producer.c
11          gcc -Wall -g -o producer producer.c -lrt
12
13  simple_p: simple_p.c
14          gcc -Wall -g -o simple_p simple_p.c -lrt
15
16  simple_c: simple_c.c
17          gcc -Wall -g -o simple_c simple_c.c -lrt
18
19  clean:
20          rm -fr producer consumer simple_p simple_c *~ *.o   core*
```

Next, we provide the implementation of the classical producer-consumer application using POSIX shared memory. Producer and consumer processes share a memory segment in which a buffer and three variables are sitting: in, out, and count.

This solution is not free of race conditions. It can have race conditions due to simultaneous access to shared data by more than one process. But we are just ignoring this possibility at the moment. Most of the time the program will run correctly.

In this example, we need to run the producer process first, since it creates a shared segment. Then the consumer is run. Producer will start generating and sending items (integers in this case) through the shared segment to the consumer process. The consumer process will retrieve the items from the shared memory and print them out.

Below is the header file `commondefs.h` that includes the common definitions used by the producer program and consumer program.

```
1  /* -*- linux-c -*- */
2  /* $Id: commondefs.h,v 1.2 2015/03/04 12:01:17 korpe Exp korpe $   */
3
4  #define BUFFER_SIZE 10
5
6  struct shared_data
7  {
8    int in;
9    int out;
10   int buffer[BUFFER_SIZE];
11 };
12
13
14 #define SHAREDMEM_NAME "/somenamehere"
15 #define SHAREDMEM_SIZE 1024 /* bytes; large enough to hold the data */
16
17 #define NUM_ITEMS_TOPASS 50
```

In the header file, we define a structure in which we define the variables (fields) that will be sitting in the shared memory segment. Those fields are: a buffer and the variables in and out. In our program, we will define a pointer variable pointing to this structure and that pointer will be initialized to point to the beginning of the shared segment created. In this way, we will easily access the content sitting in the shared memory by easily accessing to the fields of the structure. The `in` field will point to the place (i.e., stores the index of the array entry) where the next produced item will be put into. The `out` field points to the place in the array from where the next item will be retrieved by the consumer process. The `buffer` field of the structure will keep the items.

Below is the producer program (`producer.c`).

```
1  /* -*- linux-c -*- */
2  /* $Id: producer.c,v 1.1 2015/03/04 12:00:03 korpe Exp korpe $ */
3
4  #include <unistd.h>
5  #include <stdlib.h>
```

```
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
13
14 #include "commondefs.h"
15
16 int
17 main(int argc, char **argv)
18 {
19
20         int i;
21         int fd;
22         char sharedmem_name[200];
23
24         void *shm_start;
25         struct shared_data *sdata_ptr; /* pointer to the shared data */
26         struct stat sbuf;
27
28         if (argc != 1) {
29                 printf("usage: producer\n");
30                 exit(1);
31         }
32
33         strcpy(sharedmem_name, SHAREDMEM_NAME);
34
35         /* create a shared memory segment */
36         fd = shm_open(sharedmem_name, O_RDWR | O_CREAT, 0660);
37         if (fd < 0) {
38                 perror("can not create shared memory\n");
39                 exit (1);
40
41         } else {
42                 printf
43                     ("sharedmem create success, fd = %d\n", fd);
44         }
45
46         /* set the size of the shared memory */
47         ftruncate(fd, SHAREDMEM_SIZE);
48
49         fstat(fd, &sbuf); /* get info about shared memmory */
50         /* check the size */
51         printf("size = %d\n", (int) sbuf.st_size);
52
53         // map the shared memory into the address space of the process
54         shm_start = mmap(NULL, sbuf.st_size,
55                          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
56
57         if (shm_start < 0) {
58                 perror("can not map the shared memory \n");
59                 exit (1);
60         } else
61                 printf
```

```
62                      ("mapping ok, start address = %lu\n",
63                       (unsigned long) shm_start);
64
65          close(fd);
66          /* we no longer need the file desciptor */
67
68          sdata_ptr = (struct shared_data *) shm_start;
69
70          sdata_ptr->in = 0;
71          sdata_ptr->out = 0;
72
73          i = 0;
74
75          while (1) {
76                  i++; /* we produced an integer: i */
77
78                  while ((sdata_ptr->in+1) % BUFFER_SIZE == sdata_ptr->out)
79                    ; // buffer is full - wait in a busy loop
80
81                  sdata_ptr->buffer[sdata_ptr->in] = i;
82                  sdata_ptr->in = (sdata_ptr->in + 1) % BUFFER_SIZE;
83
84                  printf ("producer put item %d into buffer\n", i);
85                  sleep (1); /* sleep 1 second */
86
87                  if (i == NUM_ITEMS_TOPASS)
88                          break;
89          }
90
91          exit(0);
92  }
```

In the producer program, we create a shared memory segment using the `shm_open` function (system call). We provide a name for the segment to be created. The name we use is defined in the common header file. The shm_open function returns a file descriptor to the calling program. In subsequent related calls we use this file descriptor. We set the size of the shared segment using the `ftruncate` function. The same function is used to set the size of a file to some value. In this case, we set the size to the values `SHAREDMEM_SIZE` defined in our common header file.

Then, the `mmap` function is used to obtain a pointer to the beginning of the shared segment. In other words, we use mmap to map the shared segment into the address space of the producer program. In this way, the producer sees it as part of its address space and accesses it by using the returned pointer value. The pointer value is stored in a local variable called `shm_start` in the producer program. We can now use this pointer to access the shared memory region. The shm_start is a void pointer. We define another pointer (`sdata_ptr` which is of type `struct shared_data` and therefore can be used to point to such a structure. We initialize the `sdata_ptr` pointer to point to the shared segment.

```
1  sdata\_ptr = (struct shared\_data *) shm\_start;
```

In this way, we can then access the shared segment by using the pointer and the fields of the structure it is pointing to. So, the access to shared memory becomes so easy afterwards. We can say, for example:

```
1         sdata_ptr->in = 0;
2         sdata_ptr->out = 0;
```

With this, we are initializing the part of the shared segment where the variables (fields of the structure) in and out are located. Hence we are accessing the shared memory by using ordinary memory access (pointers and assignment statement). No system calls are done to access the shared memory segment.

In the while loop, the producer program produces integers and puts them into the buffer in the shared memory using ordinary memory accesses. As we said, no system calls are needed during this time.

Below is the consumer program (`consumer.c`).

```
1  /* -*- linux-c -*- */
2  /* $Id: consumer.c,v 1.1 2015/03/04 12:00:03 korpe Exp korpe $ */
3
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
13
14 #include "commondefs.h"
15
16 int
17 main(int argc, char **argv)
18 {
19
20         int i;
21         char sharedmem_name[200];
22         int fd;
23
24         void *shm_start;
25         struct shared_data *sdata_ptr;
26
27         struct stat sbuf;
28         // attributes of the file referring to the segment
29
30         if (argc != 1) {
31                 printf("usage: consumer \n");
32                 exit(1);
```

```
33          }
34
35          strcpy(sharedmem_name, SHAREDMEM_NAME);
36
37          fd = shm_open(sharedmem_name, O_RDWR, 0660);
38
39          if (fd < 0) {
40                  perror("can not open shared memory\n");
41                  exit (1);
42          } else {
43                  printf
44                      ("sharedmem open success, fd = %d  \n", fd);
45          }
46
47          fstat(fd, &sbuf);
48          printf("size of sharedmem = %d\n", (int) sbuf.st_size);
49
50          // map the shared segment into your address space
51          shm_start =
52              mmap(NULL, sbuf.st_size,
53                   PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
54
55          if (shm_start < 0) {
56                  perror("can not map shared memory \n");
57                  exit (1);
58          } else
59                  printf("mapping ok, start address = %lu\n",
60                         (unsigned long) shm_start);
61
62          close(fd);
63
64
65          sdata_ptr = (struct shared_data *)  shm_start;
66
67          while (1) {
68
69                  while (sdata_ptr->in == sdata_ptr->out)
70                    ; // buffer is empty -- wait in a busy loop
71
72                  i = sdata_ptr->buffer[sdata_ptr->out];
73                  sdata_ptr->out = (sdata_ptr->out + 1) % BUFFER_SIZE;
74
75                  printf ("consumer retrieved item %d from buffer\n", i);
76
77                  sleep (1);
78
79                  if (i == NUM_ITEMS_TOPASS)
80                          break;
81          }
82
83          shm_unlink(sharedmem_name);
84          // remove the shared segment
85
86          printf("removed the shared memory segment!. bye...\n");
87          exit(0);
88  }
```

The consumer program opens the shared memory segment by using again the `shm_open` system call. Then it uses mmap to map the stared memory into its address space (i.e., gets a pointer to point to the beginning of the shared memory). The pointer is stored in the local variable `shm_start`. Another pointer (sdata_ptr) of type `struct shared_data` is initialized to this pointer. Hence sdata_pointer can be now be used to access the shared memory in an easy manner.

Below is a Makefile that can be used to compile the programs.

```
1  # $Id: Makefile,v 1.1 2015/03/04 12:01:25 korpe Exp korpe $
2
3  all: producer consumer simple_p simple_c
4
5
6  consumer: consumer.c
7          gcc -Wall -g -o consumer consumer.c -lrt
8
9
10 producer: producer.c
11         gcc -Wall -g -o producer producer.c -lrt
12
13 simple_p: simple_p.c
14         gcc -Wall -g -o simple_p simple_p.c -lrt
15
16 simple_c: simple_c.c
17         gcc -Wall -g -o simple_c simple_c.c -lrt
18
19 clean:
20         rm -fr producer consumer simple_p simple_c *~ *.o   core*
```

We type `make` to compile the programs and obtain two executable files: producer and consumer. We run the producer first since it creates the shared memory (consumer could create the shared memory as well, instead of producer). Then we run the consumer. We can see what is going on in the screen.

5.3 System V Shared Memory API

In a Unix/Linux system, we have another set of functions (interface) that can be used to create and use shared memory. This interface comes from an early version of the Unix operating system, called System V, developed at AT&T. Therefore, it is called System V shared memory interface. Now it is also part of the POSIX standard interface, hence we can call it also as POSIX shared memory API. That means we have now two POSIX APIs for shared memory. In the previous section we described one of them, which is the newer one. In this section, we describe and give an example for the second one, which is historically developed with System V Unix. We will refer to it as System V API here.

Below, we provide an example application showing how to use System V shared memory API. It involves a server program and a client program. The server program and client program take a filename as a command line parameter. That filename is used to refer to the same shared memory segment that will be shared by the two processes to pass information to each other. The filename can be the name of any existing file. We can create, for example, a temporary file in our current directory with some name, say `file-x` (for creatring a new file you can simply use the `touch` command), and then use it in both programs to refer to the same shared memory segment.

We run the programs as follows. We run the server first:

```
./server file-x
```

and then the client:

```
./client file-x
```

The client is just putting some numbers into the shared memory and the server is just reading and outputting them to the screen. We could do the reserve.

Below is the code of the server program.

```
/*
   $Id: server.c,v 1.1 2015/03/04 13:47:22 korpe Exp korpe $
*/

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SHMEMSIZE 16   // size of shm to create
#define MAXPATHNAME 256

int
main(int argc, char **argv)
{
        int i;
        int tick;
        int ret;
```

```
25          int fd;
26          struct shmid_ds buf;        /* stores info about shared mem */
27          char pathname[MAXPATHNAME];  /* pathname to derive a key */
28          key_t key;                /* key for shared mem*/
29          int mem_id;                /* id of shared mem */
30          char *ptr;                /* pointer to shared mem*/
31
32          if (argc != 2) {
33                  printf("usage: server <pathname> \n");
34                  exit(1);
35          }
36
37          strcpy(pathname, argv[1]);
38
39          printf("pathname = %s \n", pathname);
40
41          /*
42             just test if the file exists or not. if not
43             create the file.
44           */
45          fd = open(pathname, O_RDWR | O_CREAT, 0660);
46          if (fd == -1) {
47                  printf("can not open or create the file: %s\n", pathname);
48                  exit(1);
49          }
50          close(fd);
51
52          /*
53             obtain a key from pathname.
54           */
55          key = ftok(pathname, 2);
56          if (key == -1) {
57                  printf("can not generate the key \n");
58                  exit(1);
59          }
60          printf("key = %d \n", key);
61
62          // shmget creates or opens a shm with given key.
63          // if there is already one, just open it.
64
65          mem_id = shmget(key, (size_t) SHMEMSIZE, 0600 | IPC_CREAT);
66          if (mem_id == -1) {
67                  perror("can not create shm\n");
68                  exit(1);
69          } else {
70                  printf("shm, shmid = %d \n", mem_id);
71          }
72
73
74          // with shmctl(),  we can get info about the shm */
75
76          ret = shmctl(mem_id, IPC_STAT, &buf);
77          if (ret == -1) {
78                  perror("shmget failed");
79                  exit(0);
80          }
```

```
81
82          printf("shmem->size = %d\n", (int) buf.shm_segsz);
83
84          //   attach to the shared memory.
85          //   obtain a pointer.
86
87          ptr = (char *) shmat(mem_id, NULL, 0);
88          if (ptr != NULL) {
89                  printf("attached to the shared memory \n");
90          } else {
91                  printf("can not attach to the shared memory \n");
92                  exit(1);
93          }
94
95          tick = 0;
96          while (1) {
97
98                  printf("shared memory content (in hex) at tick %d:\n",
99                         tick);
100                 for (i = 0; i < buf.shm_segsz; ++i) {
101                         printf("%02x ", (unsigned char) ptr[i]);
102                 }
103                 printf("\n\n");
104                 sleep(1);
105                 tick++;
106         }
107 }
```

The server program takes the command line parameter, i.e., the filename (pathname), and obtains a unique key from it using the ftok() function. That key is then used to create a shared memory by calling the shmget() system call. The return value is an integer ID for the created shared memory. Subsequent operations on the shared memory can use this ID. The program attaches to the shared memory by calling the `shmat` system call.

Then, in an endless loop, the server reads the content of the shared memory (which is just 16 bytes long as indicated with the SHMEMSIZE macro) and displays the content to the screen.

Below is the client program.

```
1   /*
2      $Id: client.c,v 1.1 2015/03/04 13:47:17 korpe Exp korpe $
3   */
4
5   #include <unistd.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8   #include <sys/ipc.h>
9   #include <sys/shm.h>
10  #include <sys/types.h>
11  #include <string.h>
```

```
12  #include <errno.h>
13
14  #define MAXPATHNAME 256
15
16  char pathname[MAXPATHNAME];
17  int mem_id;
18
19  int
20  main(int argc, char **argv)
21  {
22          int ret;
23          struct shmid_ds buf;
24          char pathname[MAXPATHNAME];
25          key_t key;
26          int mem_id;
27          char *ptr;
28
29          if (argc != 2) {
30                  printf("usage: client <pathname> \n");
31                  exit(1);
32          }
33
34          strcpy(pathname, argv[1]);
35          printf("pathname = %s \n", pathname);
36
37          /*
38             obtain a key from pathname.
39           */
40          key = ftok(pathname, 2);
41          if (key == -1) {
42                  printf("can not generate the key \n");
43                  exit(1);
44          }
45          printf("key = %d \n", key);
46
47
48          //create or open a shared memory region (segment)
49          mem_id = shmget(key, 0, 0600 | IPC_CREAT);
50          if (mem_id == -1) {
51                  perror("shmget failed");
52                  exit(1);
53          } else {
54                  printf("shmget success, shmid = %d \n", mem_id);
55          }
56
57          //obtain info about created shared mem segment
58          ret = shmctl(mem_id, IPC_STAT, &buf);
59          if (ret == -1) {
60                  perror("shmctl failed");
61                  exit(0);
62          }
63          printf("shmem->size = %d\n", (int) buf.shm_segsz);
64
65          ptr = (char *) shmat(mem_id, NULL, 0);
66          if (ptr != NULL)
67                  printf("attached to shm\n");
```

```
68          else {
69                  printf("can not attach to shm\n");
70                  exit(1);
71          }
72
73          int k = 0;
74          while (1) {
75                  printf("writing to shm a byte value = %d\n", k);
76
77                  // ptr points to the beginning to shm
78                  int i;
79                  for (i = 0; i < buf.shm_segsz; ++i) {
80                          ptr[i] = (char) k;
81                  }
82                  sleep(3);
83                  k++;
84          }
85  }
```

It is opening the shared memory by using the pathname that is provided as command line parameter. First it converts the pathname to a key. Then it uses the key to open the shared memory. Then it attaches to the shared memory by using the `shmat` system call. The return value is a pointer which can be used by the program to access the shared memory.

The client then write some data into the shared memory in an endless loop. Every 3 seconds, it write a sequence of bytes with some value. The server can read them. Hence the client passes data to the server using the created shared memory segment. Note that in this program we are not applying any locking operation in accessing the shared segment. Therefore, race conditions may occur. They are not tried to be prevented in this sample program.

A created shared memory segment (i.e., shared memory object) has a long term (kernel lifetime) persistence. It is not automatically deleted when the creating process terminates, unless the process explicitly deletes it. Therefore, you can get information about the shared memory from command line (another window that you open) by use of the `ipcs` command. This command will show the ID and some other info about the created shared memory region. You can check the man page of ipcs. A related command is `ipcrm`, which is used to remove a shared memory segment.

Below, we also include a Makefile to compile the server and client program.

```
1
2
3   all: server client
4
5   server: server.c
6           gcc -Wall -g -o server server.c
7
```

```
8  client: client.c
9          gcc -Wall -g -o client client.c
10
11 clean:
12         rm -fr client server  *~ core*
```

5.4 Pipes

A pipe can be used by two processes created by the same program (for example a parent and child) to exchange a stream of bytes. It is a uni-directional communication mechanism. It is very easy to use provided that the processes are created from the same program.

Below, we provide a sample application using pipes.

```
1  /* $Id: pipe.c,v 1.2 2015/02/26 12:51:24 korpe Exp korpe $ **/
2
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <stdlib.h>
7
8  int
9  main(int argc, char *argv[])
10 {
11         int fd[2];
12         int pid;
13         int i;
14
15         if (pipe(fd) < 0) {
16                 printf ("could not create pipe\n");
17                 exit (1);
18         }
19
20
21         pid = fork();
22         if (pid < 0) {
23                 printf ("could not create child\n");
24                 exit (1);
25         }
26
27         if (pid == 0) {
28                 close (fd[1]);
29
30                 printf ("This is the child process.\n");
31
32                 unsigned char recv_byte;
33                 while (read(fd[0], &recv_byte, 1) > 0)
34                 {
35                         printf ("%d ", recv_byte);
36                 }
```

```
37
38                printf ("\n");
39                fflush (stdout);
40                close(fd[0]);
41                printf ("child terminating\n");
42        }
43        else {
44                close(fd[0]);
45
46                // sent some number of bytes
47                unsigned char sent_byte;
48                for (i=0; i < 100; ++i)  {
49                        sent_byte = (unsigned char) i;
50                        write (fd[1], &sent_byte, 1);
51                }
52
53                close(fd[1]);
54                printf ("parent terminating\n");
55                exit (0);
56        }
57
58        return 0;
59 }
```

It is creating a child process and uses a pipe to enable communication between parent and child. A pipe is created before the child process is created. Two descriptors fd[0] and fd[1] are set by the pipe() call. Those descriptors are passed to the child process. Hence, one descriptor can be used by the parent (for sending for example) and the other one can be used by the child (for receiving for example) to communicate with each other.

Below is a Makefile to compile the program.

```
1 # $Id: Makefile,v 1.1 2015/02/26 12:53:46 korpe Exp korpe $
2
3 process: pipe.c
4          gcc -g -Wall -o pipe pipe.c
5 clean:
6         rm -fr pipe *~ *.txt core*
```

We can run the program with:

```
1 ./pipe
```

Pipes are used quite heavily by Unix programs. The shell program, for example, is using pipes to re-direct output among several programs that are run together as part of the same shell command. For example, we can write the following at the shell prompt:

```
1 > ps aux | sort | more
```

As a result, the shell process will create three processes: ps, sort, and more. The output of ps process is given to the sort process by use of a pipe in between (whatever ps writes to stdout is redirected to the pipe created between ps and sort). Similarly, the output of the sort is given as an input to the more program by use of another pipe in between.

Chapter 6

Over Network Communication

6.1 Sockets and Network Programming

Two processes running in different machines can talk to each other using sockets. We need to learn the socket API to program with sockets and to communicate processes in different machines. Most operating systems, including Linux and Windows, provide a socket interface (API) for such network programming, i.e. writing network applications involving processes running in different machines. We need to have, of course, these machines connected together using a network like the Internet (the TCP/IP network). Internet is using TCP (Transmission Control Protocol) and IP (Internet Protocol) protocols to govern the transportation of messages (packets) between machines and processes. TCP is a transport protocol ensuring reliability over the Internet which can loose packets from time time. TCP also provides congestion control. There is another transport protocol called UDP (User Datagram Protocol) which is not providing any mechanism for reliable delivery and congestion control, and therefore causing less delay on the average.

TCP, UDP and IP protocols are usually implemented inside the kernel. The kernel also implements the socket interface so that we make socket related calls in our applications to create and use sockets to exchange data over the Internet. The kernel also implements the network driver driving the network card in our machine. The networking software in our machine is layered, following the networking layering model (networking stack) defined in the standards. The driver and card implements the layer 2 (link layer). The IP protocol implementation in the kernel is the implementation of a layer 3 protocol. The IP protocol implementation is responsible to assemble and emit and receive and parse IP packets. Each TCP/IP or UDP/IP packet includes a source IP address, a source port number, a destination IP address and a destination port number. TCP and UDP are transport layer protocols (layer 4). Hence the TCP implementation in the kernel is the layer 4 implementation of the networking stack. Similarly, the UDP implementation in our kernel is again a layer 4 protocol implementation. From top to bottom we have the following

layering: Application, Socket Interface, TCP or UDP, IP, and finally Driver/NetworkCard.

A socket can either use TCP or UDP. If we want reliable communication for our application, then we need to use UDP. Some applications like file transfer, web page transfer, etc., are requiring absolute reliability and therefore need to be using TCP sockets (STREAM sockets). Some applications like multimedia audio/video applications (such as a Skype audio communication) may tolerate some data losses, and therefore can use UDP sockets (DATAGRAM sockets).

TCP is connection oriented. This means two sockets (one in one process, one in the other process) need to be connected together using the TCP's three-way handshake before processes can use those sockets to exchange data. This connection establishment is requested by applications and done by the kernels (TCP protocols running in the kernels). After a TCP connection is established, processes can use the related sockets to send/receive data. TCP is also a byte-stream oriented protocol. A sequence of bytes are sent from one side to the other side. We can consider that a pipe is created from one socket to the other with a TCP connection establishment. Then we can drop data into this pipe and it will find its way. We don't need to specify the destination (destination IP address and port number) with every data send operation.

UDP is a connectionless protocol. We just create a UDP socket and we can start sending and receiving data from it immediately. Since the socket is not connected to some other socket (there is no virtual pipe created), we need to specify the destination (destination IP address and port number) for each data we are sending. This also means that through the same socket we can send data to various destinations. But a TCP socket can be used to send to only one other end, i.e., the socket to which the connection is established.

UDP is a packet-oriented datagram protocol. It preserves packet boundaries (message boundaries). That means, when we send a message of size x, the receiver receives exactly a message of size x or nothing. It never receives a message partially with a receive operation. This is different than TCP. In TCP, when a sender sends x bytes, the receiver may not receive x bytes immediately with one system call, but can receive these x bytes (in the same order and without loss) with several system calls.

6.1.1 Developing TCP Server and Client Programs

A TCP server process uses two types of sockets: listening socket and connection socket. A listening socket must be used to listen and accept incoming connection requests from clients. For each connection request accepted, a new socket (connection socket) is created and connected to the respective client. If there are, for example, 5 connections established, then there must be 5 connection sockets created at the server. But one listening socket can be used to listen for all connection requests.

When a packet is received by a machine that may be running one or more processes

using TCP sockets, the incoming packet must be demultiplexed (directed) to the correct TCP socket. This can be done by looking to 4 parameters: sender IP address, sender port number, receiver IP address, receiver port number. This is because each TCP socket created in the machine is identified by a 4-tuple: (remote IP address, remote port number, local IP address, local port number). Such a 4-tuple uniquely identifies a TCP socket created in the machine. This is the case even though the TCP sockets in the machine have the same local port numbers. Because, either their remote IP addresses or port numbers will be different and each will have a different 4-tuple value.

Below we show you the code for a TCP server and a TCP client program. The server uses an IP address SERVER_IP that is defined as a constant (macro) in the header file given below. The server uses also a SERVER_PORT defined in the same header file as the port number to which connection requests can be sent from clients. You should choose an unused port number for your server. The header file is included by both the server and client programs. Make sure to chance the SERVER_IP and SERVER_PORT depending on your machine.

```
1
2
3  #define SERVER_PORT 13422
4  #define SERVER_IP "127.0.0.1"
5
6
7  #define IPSTRLEN 64
```

Below we show you the code for the TCP server program. A TCP server uses socket() and listen() system calls to create a listening socket and set it to the passive mode (listening mode). None of these two calls are blocking. They return immediately after doing their job. Then the accept() system call (function) is called on the listening socket. This will put the caller (server process) into waiting state until somebody (some client process) makes a connection request using the connect() system call. Then the connection is accepted by the accept() system call at the server side, and a new socket is created and returned by accept(). When accept() returns, the server can continue running. The new socket is called a connection socket and it is connected to the socket of the client requesting connection. Then the server and client are ready to send/receive (exchange) data. After serving the client, the server process can wait for another incoming connection request from the same or different client by calling the accept() function again.

A client request can be handled by the server process itself or the server can create a new child process (or thread) to handle the request. Then the request is handled concurrently by the child process (or thread), while the server is waiting for other requests. In this way, a new child process or thread can be created to handle a new request. This way we can see many child processes or threads running and handling requests concurrently in the system. When a child finishes handling a request, it can terminate.

```
1  /*    -*- linux-c -*- */
```

```
2  /* $Id: tcp_server.c,v 1.4 2012/02/26 10:37:31 korpe Exp korpe $ */
3
4  #include <sys/socket.h>
5  #include <sys/types.h>
6  #include <stdio.h>
7  #include <arpa/inet.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 #include <netinet/in.h>
12
13 #include "shared.h"
14
15 int main()
16 {
17
18         int list_sock;
19         int conn_sock;
20         struct sockaddr_in sa, ca;
21         socklen_t ca_len;
22         char buf[1024];
23         int i;
24         char ipaddrstr[IPSTRLEN];
25
26         list_sock = socket (AF_INET, SOCK_STREAM, 0);
27
28         bzero (&sa, sizeof(sa));
29         sa.sin_family = AF_INET;
30         sa.sin_addr.s_addr = htonl(INADDR_ANY);
31         sa.sin_port = htons(SERVER_PORT);
32         bind (list_sock,
33               (struct sockaddr *) &sa,
34               sizeof(sa));
35
36         listen (list_sock, 5);
37
38         while (1)
39         {
40
41                 bzero (&ca, sizeof(ca));
42                 ca_len = sizeof(ca); // important to initialize
43                 conn_sock = accept (list_sock,
44                                     (struct sockaddr *) &ca,
45                                     &ca_len);
46
47                 printf ("connection from: ip=%s port=%d \n",
48                         inet_ntop(AF_INET, &(ca.sin_addr),
49                                   ipaddrstr, IPSTRLEN),
50                         ntohs(ca.sin_port));
51
52                 for (i=0; i<100; ++i)
53                 {
54                         *((int *)buf) = htonl(i+20);
55                         // we using converting to network byte order
56
57                         write (conn_sock, buf, sizeof(int));
```

```
58                 }
59
60                 * ((int *)buf) = htonl(-1);
61                 write (conn_sock, buf, sizeof(int));
62
63                 close (conn_sock);
64
65                 printf ("server closed connection to client\n");
66         }
67 }
```

Below is the client code. After creating a socket, it invokes the connect system call to connect to a server.

TCP Client:

```
1  /*    -*- linux-c -*- */
2  /* $Id: tcp_client.c,v 1.3 2012/02/25 21:22:43 korpe Exp $ */
3
4  #include <sys/socket.h>
5  #include <sys/types.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <stdio.h>
9  #include <sys/socket.h>
10 #include <arpa/inet.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <unistd.h>
14 #include <netinet/in.h>
15
16 #include "shared.h"
17
18 int main()
19 {
20
21         int  sock;
22         struct sockaddr_in sa;
23         int ret;
24         char buf[1024];
25         int x;
26
27         sock = socket (AF_INET, SOCK_STREAM, 0);
28
29         bzero (&sa, sizeof(sa));
30         sa.sin_family = AF_INET;
31         sa.sin_port = htons(SERVER_PORT);
32         inet_pton (AF_INET, SERVER_IP, &sa.sin_addr);
33
34         ret = connect (sock,
35                        (const struct sockaddr *) &sa,
36                        sizeof (sa));
37
```

```
38          if (ret != 0) {
39                  printf ("connect failed\n");
40                  exit (0);
41          }
42
43
44          x = 0;
45          while (x != -1) {
46                  read (sock, buf , sizeof(int));
47                  x = ntohl(*((int *)buf));
48                  if (x != -1)
49                          printf ("int rcvd = %d\n", x);
50          }
51
52          close (sock);
53
54          exit (0);
55  }
```

Below we have the UDP server program.

UDP Server:

```
1   /*    -*- linux-c -*- */
2   /* $Id: udp_server.c,v 1.3 2012/02/26 10:37:31 korpe Exp korpe $ */
3
4   #include <sys/socket.h>
5   #include <sys/types.h>
6   #include <stdio.h>
7   #include <arpa/inet.h>
8   #include <stdlib.h>
9   #include <string.h>
10  #include <unistd.h>
11  #include <netinet/in.h>
12
13  #include "shared.h"
14
15  int main()
16  {
17          int sock;
18          struct sockaddr_in sa, ca;
19          int ca_len;
20          char message[1024];
21          int n;
22
23          sock = socket (AF_INET, SOCK_DGRAM, 0);
24
25          bzero (&sa, sizeof(sa));
26          sa.sin_family = AF_INET;
27          sa.sin_addr.s_addr = htonl(INADDR_ANY);
28          sa.sin_port = htons(SERVER_PORT);
29
30          bind (sock, (struct sockaddr *) &sa,
31                sizeof(sa));
```

```
32
33          while (1)
34          {
35                  ca_len = sizeof(ca); // important to init
36                  n = recvfrom (sock, message, 1024, 0,
37                                (struct sockaddr *) &ca,
38                                (socklen_t *) &ca_len);
39
40                  printf ("received: %s\n", message);
41
42                  sendto (sock, message, n, 0,
43                          (struct sockaddr *) &ca,
44                          ca_len);
45          }
46  }
```

Below we have the UDP client program.

UDP Client:

```
1   /*    -*- linux-c -*- */
2   /* $Id: udp_client.c,v 1.3 2012/02/25 21:22:43 korpe Exp $ */
3
4   #include <sys/socket.h>
5   #include <sys/types.h>
6   #include <stdio.h>
7   #include <arpa/inet.h>
8   #include <stdlib.h>
9   #include <string.h>
10  #include <unistd.h>
11  #include <netinet/in.h>
12
13  #include "shared.h"
14
15  int main()
16  {
17          int sock;
18          struct sockaddr_in sa;
19          char astring[1024];
20          char message[1024];
21          int n;
22
23          sock = socket (AF_INET, SOCK_DGRAM, 0);
24
25          bzero (&sa, sizeof(sa));
26            sa.sin_family = AF_INET;
27          sa.sin_port = htons(SERVER_PORT);
28          inet_pton (AF_INET, SERVER_IP, &sa.sin_addr);
29
30          strcpy (astring, "hello server");
31          sendto (sock, astring, strlen(astring)+1, 0,
32                  (struct sockaddr *) &sa, sizeof(sa));
33
34          n = recvfrom (sock, message, 1024, 0, NULL, NULL);
```

```
35          if (n == -1) {
36                  printf ("recvfrom failed\n");
37                  exit (1);
38          }
39
40          printf ("received back: %s\n", message);
41          return 0;
42  }
```

Below we have a Makefile to compile all these programs. Run a server first before running a client.

Makefile

```
1
2
3
4   all: tcp_server tcp_client udp_server udp_client s c
5
6   s: s.c
7                   gcc -g -Wall -o s s.c
8
9   c: c.c
10                  gcc -g -Wall -o c c.c
11
12  tcp_server: tcp_server.c
13              gcc -g -Wall -o tcp_server tcp_server.c
14
15  tcp_client: tcp_client.c
16              gcc -g -Wall -o tcp_client tcp_client.c
17
18  udp_server:
19              gcc -g -Wall -o udp_server udp_server.c
20
21  udp_client:
22              gcc -g -Wall -o udp_client udp_client.c
23
24  clean:
25          rm -fr core tcp_client tcp_server udp_client udp_server  *~
26
27
```

Chapter 7

Synchronization

7.1 POSIX Named Semaphores

POSIX API provides a set of functions to declare and use semaphores. There are two types of POSIX semaphores, named semaphores and unnamed semaphores. You can obtain more information about semaphores by reading the man page sem_overview. For that, just type:

```
man sem_overview
```

Here we provide an example application that uses named semaphores. A named semaphore can be accessed by many processes using the same name to open it. The application is again the classical producer-consumer bounded-buffer problem. There is a bounded buffer between a producer process and a consumer process. It has BUFSIZE slots, hence can hold at most BUFSIZE items.

The bounded buffer sits on a shared memory segment between the producer and consumer. A shared variable that keeps the number of items in the buffer is also sitting in the shared memory. We pack those data (buffer and count variable) into a structure defined below. The structure also contains two additional fields (variables): in and out. The in variable is accessed by the producer and the out variable is accessed by the consumer. These variables do not have to be sitting in the shared memory, but we put them into the shared data structure since they are related with accessing the buffer.

```
1 struct shared_data {
2       int buf[BUFSIZE];    /* shared buffer */
3       int count;           /* current number of items in buffer */
4       int in;  /* points to the first empty slot */
5       int out; /* points to the first full slot */
6 };
```

This structure is defined in a header file, called `common.h` that is included by both the producer program and consumer program. It is shown below. It also includes the declarations of some macros such as the BUFSIZE, NUM_ITEMS, etc.

```
1  /* -*- linux-c  -*-  */
2  /*  $Id: common.h,v 1.3 2009/03/15 20:15:35 korpe Exp korpe $ */
3
4  #ifndef COMMON_H
5  #define COMMON_H
6
7  #define SEMNAME_MUTEX       "/name_sem_mutex"
8  #define SEMNAME_FULL        "/name_sem_fullcount"
9  #define SEMNAME_EMPTY       "/name_sem_emptycount"
10
11 #define SHM_NAME  "/name_shm_sharedsegment1"
12
13 #define BUFSIZE  10               /*  bounded buffer size */
14
15 #define NUM_ITEMS 10000               /*  total items to produce  */
16
17 /* set to 1 to synchronize;
18    otherwise set to 0 and see race condition */
19 #define SYNCHRONIZED 0
20
21 struct shared_data {
22         int buf[BUFSIZE];    /* shared buffer */
23         int count;           /* current number of items in buffer */
24         int in; // this field is only accessed by the producer
25         int out; // this field is only accessed by the consumer
26 };
27
28 #endif
```

The header file also contains the macro definitions (constants) of the names for 3 semaphores. Different processes use the same name to acess the same semaphore. One of the processes creates a semaphore by using the sem_open function with O_CREAT flag, and other processes can open the semaphore using the same sem_open function.

```
1  #define SEMNAME_MUTEX       "/name_sem_mutex"
2  #define SEMNAME_FULL        "/name_sem_fullcount"
3  #define SEMNAME_EMPTY       "/name_sem_emptycount"
```

The producer program first creates a shared memory segment that is just big enough to hold the structure defined above. The shared memory is created and attached using shm_open() and mmap functions(). As a result, the producer process gets the start address (an address in its own address space; this is a logical address) pointing to the start of the shared segment. That address is stored in the pointer variable `shm_start`.

We also defined a pointer variable to point to the shared data. The variable name is `sdp`. It is initialized to point to the start of the shared memory segment. Hence the structure sits in the beginning of the shared segment. Then we can access the shared data structure by using this pointer. For example, the following lines of code initializes the buffer array and the count variable.

```
1         sdp = (struct shared_data *) shm_start;
2         for (i = 0; i < BUFSIZE; ++i)
3             sdp->buf[i] = 0;
4         sdp->count = 0;
```

Those intialization statements are writing into the shared memory segment. Note that we could also use the shm_start variable to access the buffer and count. In this example, shm_start and sdp are pointing to the same place.

The producer program then creates 3 semaphores and initializes them using the `shm_open` function. There is one semaphore for mutual exclusion (`sem_mutex`) and two semaphores for synchronization (`sem_full` and `sem_empty`). The `sem_mutex` semaphore is initialized to 1, the sem_full is initiazed to 0 (indicates the number of full slots), and sem_empty is initialized to BUFSIZE (indicates the number of empty slots).

Then in a while loop, the producer tries to put a new item into buffer if there is an empty slot (i.e., if the empty semaphore did not reach to zero; otherwise it blocks). If there is empty slot available, the producer goes and puts an item into buffer. But to prevent the consumer also accessing the buffer, the producer (and also the consumer) has to do a wait() operation on the sem_mutex semaphore. This ensures that only one of them will be updating the buffer at a time.

Inside the while loop, we see two pieces of code conditioned on the SYNCHRONIZED macro. This macro is defined in file common.h. If it is set to 1, it means we would like to synchronize the producer and consumer using semaphores. Hence the code should work correctly. If it is set to 0, it means that we don't use any synchronization tool (primitive) and this may cause race conditions. In this case, busy waiting is used to check whether the buffer has at least one empty slot or not.

Below is the producer program (`producer.c`).

```
1 /* -*- linux-c -*- */
2 /* $Id: producer.c,v 1.2 2009/03/15 19:48:55 korpe Exp $ */
3
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <stdio.h>
7 #include <sys/types.h>
8 #include <string.h>
9 #include <errno.h>
```

```
10  #include <sys/stat.h>
11  #include <fcntl.h>
12  #include <sys/mman.h>
13  #include <semaphore.h>
14  #include "common.h"
15
16  #define TRACE 1
17  #define SHM_SIZE sizeof (struct shared_data)
18
19  int
20  main(int argc, char **argv)
21  {
22
23          sem_t *sem_mutex;       /* protects the buffer */
24          sem_t *sem_full;        /* counts the number of items */
25          sem_t *sem_empty;       /* counts the number of empty buffer slots */
26
27
28          int fd;                     /* shm file descriptor */
29          struct stat sbuf;           /* info about shm */
30          void *shm_start;            /* pointer to the start of shared memory  */
31          struct shared_data * sdp;   /* pointer to shared data structure */
32          int item;    /* an integer item to pass */
33          int i;
34
35
36          /* first clean up a shm with same name */
37          shm_unlink (SHM_NAME);
38
39          /* create a shared memory segment */
40          fd = shm_open(SHM_NAME, O_RDWR | O_CREAT, 0660);
41          if (fd < 0) {
42                  perror("can not create shared memory\n");
43                  exit (1);
44          }
45          printf("shm created, fd = %d\n", fd);
46
47          ftruncate(fd, SHM_SIZE);    /* set size of shared memmory */
48          fstat(fd, &sbuf);
49          printf("shm_size=%d\n", (int) sbuf.st_size);
50
51          shm_start = mmap(NULL, sbuf.st_size, PROT_READ | PROT_WRITE,
52                           MAP_SHARED, fd, 0);
53          if (shm_start < 0) {
54                  perror("can not map shm\n");
55                  exit (1);
56          }
57          printf ("mapped shm; start_address=%u\n", (unsigned int) shm_start);
58          close(fd); /* no longer need the descriptor */
59
60          sdp = (struct shared_data *) shm_start;
61          for (i = 0; i < BUFSIZE; ++i)
62                  sdp->buf[i] = 0;
63          sdp->count = 0;
64          sdp->in = 0;
65
```

```
66
67          /* first clean up semaphores with same names */
68          sem_unlink (SEMNAME_MUTEX);
69          sem_unlink (SEMNAME_FULL);
70          sem_unlink (SEMNAME_EMPTY);
71
72          /* create and initialize the semaphores */
73          sem_mutex = sem_open(SEMNAME_MUTEX, O_RDWR | O_CREAT, 0660, 1);
74          if (sem_mutex < 0) {
75                  perror("can not create semaphore\n");
76                  exit (1);
77          }
78          printf("sem %s created\n", SEMNAME_MUTEX);
79
80          sem_full = sem_open(SEMNAME_FULL, O_RDWR | O_CREAT, 0660, 0);
81          if (sem_full < 0) {
82                  perror("can not create semaphore\n");
83                  exit (1);
84          }
85          printf("sem %s created\n", SEMNAME_FULL);
86
87          sem_empty =
88                  sem_open(SEMNAME_EMPTY, O_RDWR | O_CREAT, 0660, BUFSIZE);
89          if (sem_empty < 0) {
90                  perror("can not create semaphore\n");
91                  exit (1);
92          }
93          printf("sem %s create\n", SEMNAME_EMPTY);
94
95
96          item  = 0;
97          while (item < NUM_ITEMS) {
98                  if (SYNCHRONIZED) {
99                          sem_wait(sem_empty);
100                         sem_wait(sem_mutex);
101
102                         sdp->buf[sdp->in] = item;
103                         sdp->in = (sdp->in + 1) % BUFSIZE;
104
105                         sem_post(sem_mutex);
106                         sem_post(sem_full);
107
108                 } else {
109                         while (sdp->count == BUFSIZE)
110                                 ; /* busy wait */
111                         sdp->count++;
112                         sdp->buf[sdp->in] = item;
113                         sdp->in = (sdp->in + 1) % BUFSIZE;
114                 }
115
116                 if (TRACE)
117                         printf("producer put item=%d\n", item);
118
119                 item++;
120         }
121
```

```
122         sem_close(sem_mutex);
123         sem_close(sem_full);
124         sem_close(sem_empty);
125
126         printf("producer ended; bye...\n");
127         exit(0);
128 }
```

The consumer program is similar. It opens the shared memory segment (does not create it), and then also opens the semaphores that were created by the producer. Then the consumer tries to retrieve an item from the buffer if it has one. It checks this by doing a wait operation on the `sem_full` semaphore. If there is no item in the buffer, that semaphore will have a value 0 and a wait() operation on it will cause the consumer to sleep, until the producer puts an item into the buffer and issues a post() operation on the same semaphore. As said earlier, the consumer has to do a wait operation on `sem_mutex` as well before updating the buffer. This is because, if $1 < \text{count} < \text{BUFSIZE}$, both producer and consumer may be in a situation to update the buffer. Producer will not sleep on `sem_empty` and consumer will not sleep on `sem_full`. So they can be around accessing the buffer for putting or retrieving an item nearly at the same time. Therefore we need `sem_mutex` to allow only one of them to update the buffer at a time.

Below is the consumer code (`consumer.c`).

```
1  /* -*- linux-c -*- */
2  /* $Id: consumer.c,v 1.1 2009/03/15 19:48:55 korpe Exp $ */
3
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
13 #include <semaphore.h>
14 #include "common.h"
15
16 #define TRACE 1
17
18 int
19 main(int argc, char **argv)
20 {
21         int fd;
22         struct stat sbuf;
23         void *shm_start;
24         struct shared_data *sdp;
25         int item;     /* retrieved item */
26         int expected; /* expected item */
27
28         /* semaphores */
```

```
29          sem_t *sem_mutex;
30          sem_t *sem_full;
31          sem_t *sem_empty;
32
33          /* open the shared memory segment */
34          fd = shm_open(SHM_NAME, O_RDWR, 0600);
35          if (fd < 0) {
36                  perror("can not open shm\n");
37                  exit (1);
38          }
39          printf("shm open success, fd = %d\n", fd);
40
41          fstat(fd, &sbuf);
42          printf("shm size = %d\n", (int) sbuf.st_size);
43
44          shm_start = mmap(NULL, sbuf.st_size, PROT_READ | PROT_WRITE,
45                          MAP_SHARED, fd, 0);
46          if (shm_start < 0) {
47                  perror("can not map the shm \n");
48                  exit (1);
49          }
50          printf ("mapped shm; start_address=%u\n", (unsigned int) shm_start);
51          close(fd);
52
53          sdp = (struct shared_data *) shm_start;
54
55          /* open the semaphores;
56             they should already be created and initialized  */
57          sem_mutex = sem_open(SEMNAME_MUTEX, O_RDWR);
58          if (sem_mutex < 0) {
59                  perror("can not open semaphore\n");
60                  exit (1);
61          }
62          printf("sem %s opened\n", SEMNAME_MUTEX);
63
64
65          sem_full = sem_open(SEMNAME_FULL, O_RDWR);
66          if (sem_full < 0) {
67                  perror("can not open semaphore\n");
68                  exit (1);
69          }
70          printf("sem %s opened\n", SEMNAME_FULL);
71
72          sem_empty = sem_open(SEMNAME_EMPTY, O_RDWR);
73          if (sem_empty < 0) {
74                  perror("can not open semaphore\n");
75
76          }
77          printf("sem %s opened\n", SEMNAME_EMPTY);
78
79          sdp->out = 0;
80          item = -1;
81          expected = 0;
82          while (expected  < NUM_ITEMS) {
83                  if (SYNCHRONIZED) {
84                          sem_wait(sem_full);
```

```
85                          sem_wait(sem_mutex);
86
87                          item = sdp->buf[sdp->out];
88                          sdp->out = (sdp->out + 1) % BUFSIZE;
89
90                          sem_post(sem_mutex);
91                          sem_post(sem_empty);
92                  } else {
93                          while (sdp->count == 0)
94                                  ; /* busy wait */
95                          sdp->count--;
96                          item  = sdp->buf[sdp->out];
97                          sdp->out = (sdp->out + 1) % BUFSIZE;
98                  }
99
100                 if (TRACE)
101                         printf("consumer retrieved item=%d\n", item);
102
103                 if (item != expected) {
104                         printf("race condition occured; expected=%d, item=%d\n",
105                                expected, item);
106                         exit(1);
107                 }
108                 expected++;
109         }
110
111         sem_close(sem_mutex);
112         sem_close(sem_full);
113         sem_close(sem_empty);
114
115
116         /* remove the semaphores */
117         sem_unlink(SEMNAME_MUTEX);
118         sem_unlink(SEMNAME_FULL);
119         sem_unlink(SEMNAME_EMPTY);
120
121         /* remove the shared memory */
122         shm_unlink(SHM_NAME);
123
124         printf("consumer ended; bye...\n");
125         exit(0);
126 }
```

Below is a Makefile that can be used to compile these programs. As the result, two executable files will be obtained: `producer` and `consumer`. We need to run the producer first. Then the consumer.

```
1  # $Id: Makefile,v 1.2 2009/03/15 19:49:00 korpe Exp $
2
3  all: producer consumer
4
5  producer: producer.c
6                  gcc -Wall -g -o producer producer.c -lrt -lpthread
```

```
7
8  consumer:
9                      gcc -Wall -g -o consumer consumer.c -lrt -lpthread
10
11
12 clean:
13                 rm -fr producer consumer *~ *.o   core*
```

7.1.1 Pthread Mutex and Condition Variables

POSIX Pthread library has functions to create mutex variables and condition variables to be used in multi-threaded applications to synchronize threads and to protect critical sections.

The following are some of the most important Pthread API functions related to mutex and condition variables:

```
1  int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
2  int pthread_mutex_destroy(pthread_mutex_t *mutex);
3  int pthread_mutex_lock(pthread_mutex_t *mutex);
4  int pthread_mutex_unlock(pthread_mutex_t *mutex);
5  int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
6  int pthread_cond_destroy(pthread_cond_t *cond);
7  int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *restrict mutex);
8  int pthread_cond_broadcast(pthread_cond_t *cond);
9  int pthread_cond_signal(pthread_cond_t *cond);
```

We now present a multi-threaded application that makes use of Pthread mutex and condition variables. The application takes two filenames as input. It copies the integers from first file to the second file. The first file must contain positive integers, one integer per line. The application creates two threads, a producer thread and consumer thread. The producer thread opens the input file, reads one integer at a time and put the integer as item into the bounded buffer implemented as a queue (linked list). The consumer retrieves the items from the queue one by one and writes them into the output file.

The solution uses one mutex variable, `th_mutex_queue`, to provide exclusive access to the shared queue by the producer and consumer threads. It also uses two condition variables, `th_cond_hasspace`, and `th_cond_hasitem`. The first condition variable is used by the producer thread to sleep/block/wait until there is space in the buffer to put an item. It is signaled by the consumer thread. The second condition variable is used by the consumer thread to wait until there is an item in the buffer to retrieve. It is signalled by the producer thread.

Below is the program (`mutexcond.c`).

```
1  /* -*- linux-c -*-  */
2  /* $Id: mutexcond.c,v 1.3 2009/03/24 14:02:11 korpe Exp korpe $ */
3
4  #include <stdlib.h>
5  #include <mqueue.h>
6  #include <stdio.h>
7  #include <unistd.h>
8  #include <errno.h>
9  #include <string.h>
10 #include <sys/types.h>
11 #include <sys/wait.h>
12 #include <pthread.h>
13
14 #define TRACE 0
15
16 #define BUFSIZE     100
17 #define MAXFILENAME 128
18 #define ENDOFDATA   -1 /* marks the end of data stream from producer */
19
20 /*
21    Bounded buffer has a queue of items. It is a FIFO queue.  There can
22    be at most BUFSIZE items. Below we have structures related to the
23    queue and buffer.
24 */
25
26 struct bb_qelem {
27         struct bb_qelem *next;
28         int data;  /* an item - an integer */
29 };
30
31 struct bb_queue {
32         struct bb_qelem *head;
33         struct bb_qelem *tail;
34         int count; /* number of items in the buffer */
35 };
36
37 struct bounded_buffer {
38         struct bb_queue *q;               /* bounded buffer queue  */
39         pthread_mutex_t th_mutex_queue;   /* mutex  to protect queue */
40         pthread_cond_t  th_cond_hasspace;  /* will cause producer to wait */
41         pthread_cond_t  th_cond_hasitem;   /* will cause consumer to wait */
42 };
43
44
45 /************************************* global variables ***********/
46 char infilename[MAXFILENAME];
47 char outfilename[MAXFILENAME];
48 struct bounded_buffer *bbuffer;  /* bounded buffer pointer */
49
50 /******************************************************************/
51
52 void
53 bb_queue_init(struct bb_queue *q)
54 {
55         q->count = 0;
56         q->head = NULL;
```

```
57          q->tail = NULL;
58  }
59
60
61  // this function assumes that space for item is already allocated
62  void
63  bb_queue_insert(struct bb_queue *q, struct bb_qelem *qe)
64  {
65
66          if (q->count == 0) {
67                  q->head = qe;
68                  q->tail = qe;
69          } else {
70                  q->tail->next = qe;
71                  q->tail = qe;
72          }
73
74          q->count++;
75  }
76
77  // this function does not free the item
78  struct bb_qelem *
79  bb_queue_retrieve(struct bb_queue *q)
80  {
81          struct bb_qelem *qe;
82
83          if (q->count == 0)
84                  return NULL;
85
86          qe = q->head;
87          q->head = q->head->next;
88          q->count--;
89
90          return (qe);
91  }
92
93
94
95  /* producer thread start function */
96  void *
97  producer (void * arg)
98  {
99          FILE *fp;
100         int number;
101         struct bb_qelem *qe;
102
103         fp = fopen (infilename, "r");
104
105         while ( fscanf (fp, "%d", &number) == 1) {
106                 /*  insert item into buffer */
107
108                 qe = (struct bb_qelem *) malloc (sizeof (struct bb_qelem));
109                 if (qe == NULL) {
110                         perror ("malloc failed\n");
111                         exit (1);
112                 }
```

```
113                 qe->next = NULL;
114                 qe->data = number;
115
116                 pthread_mutex_lock(&bbuffer->th_mutex_queue);
117
118                 /* critical section begin */
119                 while (bbuffer->q->count == BUFSIZE)
120                         pthread_cond_wait(&bbuffer->th_cond_hasspace,
121                                           &bbuffer->th_mutex_queue);
122
123                 bb_queue_insert(bbuffer->q, qe); //
124
125                 if (TRACE)
126                         printf ("producer insert item = %d\n", qe->data);
127
128                 if (bbuffer->q->count == 1)
129                         pthread_cond_signal(&bbuffer->th_cond_hasitem);
130
131                 /* critical section end */
132
133                 pthread_mutex_unlock(&bbuffer->th_mutex_queue);
134
135         }
136
137         fclose (fp);
138
139
140         /* put and end-of-data marker to the queue */
141         qe = (struct bb_qelem *) malloc (sizeof (struct bb_qelem));
142         if (qe == NULL) {
143                 perror ("malloc failed\n");
144                 exit (1);
145         }
146         qe->next = NULL;
147         qe->data = ENDOFDATA;
148
149         pthread_mutex_lock(&bbuffer->th_mutex_queue);
150
151         /* critical section begin */
152
153         while (bbuffer->q->count == BUFSIZE)
154                 pthread_cond_wait(&bbuffer->th_cond_hasspace,
155                                   &bbuffer->th_mutex_queue);
156
157         bb_queue_insert(bbuffer->q, qe);
158
159         if (bbuffer->q->count == 1)
160                 pthread_cond_signal(&bbuffer->th_cond_hasitem);
161
162         /* critical section end */
163
164         pthread_mutex_unlock(&bbuffer->th_mutex_queue);
165
166
167         printf ("producer terminating\n");  fflush (stdout);
168
```

```
169         pthread_exit (NULL);
170 }
171
172
173
174 /* consumer thread start function */
175 void *
176 consumer (void * arg)
177 {
178         FILE *fp;
179         struct bb_qelem *qe;
180
181         fp = fopen (outfilename, "w");
182
183
184         while (1) {
185
186                 pthread_mutex_lock(&bbuffer->th_mutex_queue);
187
188                 /* critical section begin */
189
190                 while (bbuffer->q->count == 0) {
191                         pthread_cond_wait(&bbuffer->th_cond_hasitem,
192                                           &bbuffer->th_mutex_queue);
193                 }
194
195                 qe = bb_queue_retrieve(bbuffer->q);
196
197                 if (qe == NULL) {
198                         printf("can not retrieve; should not happen\n");
199                         exit(1);
200                 }
201
202                 if (TRACE)
203                         printf ("consumer retrieved item = %d\n", qe->data);
204
205                 if (bbuffer->q->count == (BUFSIZE - 1))
206                         pthread_cond_signal(&bbuffer->th_cond_hasspace);
207
208                 /* critical section end */
209
210                 pthread_mutex_unlock(&bbuffer->th_mutex_queue);
211
212                 if (qe->data != ENDOFDATA) {
213                         fprintf (fp, "%d\n", qe->data);
214                         fflush (fp);
215                         free (qe);
216                 }
217                 else {
218                         free (qe);
219                         break;
220                 }
221         }
222
223         fclose (fp);
224
```

```
225         printf ("consumer terminating\n"); fflush (stdout);
226                pthread_exit (NULL);
227 }
228
229
230 int
231 main(int argc, char **argv)
232 {
233
234         pthread_t prodtid, constid;
235         int ret;
236
237
238         if (argc != 3) {
239                 printf ("usage: mutexcond <infile> <outfile>\n");
240                 exit (1);
241         }
242
243         strcpy(infilename, argv[1]);
244         strcpy(outfilename, argv[2]);
245
246
247         /* init buffer and mutex/condition variables  */
248         bbuffer =
249                 (struct bounded_buffer *) malloc(sizeof (struct bounded_buffer));
250         bbuffer->q = (struct bb_queue *) malloc(sizeof (struct bb_queue));
251         bb_queue_init(bbuffer->q);
252         pthread_mutex_init(&bbuffer->th_mutex_queue, NULL);
253         pthread_cond_init(&bbuffer->th_cond_hasspace, NULL);
254         pthread_cond_init(&bbuffer->th_cond_hasitem, NULL);
255
256
257         ret = pthread_create (&prodtid, NULL,
258                                 producer, NULL);
259         if (ret != 0) {
260                 perror ("thread create failed\n");
261                 exit (1);
262         }
263
264
265         ret = pthread_create (&constid, NULL,
266                                 consumer, NULL);
267         if (ret != 0) {
268                 perror ("thread create failed\n");
269                 exit (1);
270         }
271
272
273         /* wait for threads to terminate */
274         pthread_join (prodtid, NULL);
275         pthread_join (constid, NULL);
276
277
278         /* destroy buffer and mutex/condition variables */
279         free(bbuffer->q);
280         free(bbuffer);
```

```
281         pthread_mutex_destroy(&bbuffer->th_mutex_queue);
282         pthread_cond_destroy(&bbuffer->th_cond_hasspace);
283         pthread_cond_destroy(&bbuffer->th_cond_hasitem);
284
285         printf ("closing...\n");
286         return 0;
287 }
```

In this application example, an issue needs a little bit more elaboration. In the producer thread, the buffer is checked if it is full or not before putting a new item. Then, if it is full, pthread_cond_wait function is called on the condition variable th_cond_hasspace. This will put the producer thread into sleep until the buffer has space again. It will happen when the consumer retrieves an item from the full buffer.

```
1                 while (bbuffer->q->count == BUFSIZE)
2                             pthread_cond_wait(&bbuffer->th_cond_hasspace,
3                                                 &bbuffer->th_mutex_queue);
4
```

This piece of code that causes the producer thread to wait until buffer again has space is shown above. We put the producer into sleep (into wait) in a while loop. This is a *pattern* that we need to use. The reason for this looping is: in case the producer thread is waken up when the condition is still not TRUE (i.e. count is still equal to BUFSIZE), we should again put the producer thread into sleep. To ensure this we use a while loop. This is not busy waiting; the thread is sleeping most of the time until count is not equal to BUFSIZE anymore. But if is waken up earlier, by checking the condition again, we prevent the consumer to try to put an item into buffer even though the count == BUFSIZE. This check is done from time to time (not all time), when the producer is waken up earlier that it is supposed to be waken up. Therefore this is not busy waiting; hence it is efficient.

7.1.2 Condition Variables: Another Example

We also give an example for the use the broadcast operation on condition variables. The program below creates many producer threads and one consumer thread. Each producer thread produces integers and puts them into the shared buffer. The consumer thread retrieves them from the buffer. If there are 10 producer threads and if each thread produces 10000 integers, then the consumer thread should retrieve 100000 integers from the buffer.

```
1  /* -*- linux-c -*- */
2  /* $Id $ */
3
4  #include <unistd.h>
5  #include <stdlib.h>
```

```
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <string.h>
9  #include <errno.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <sys/mman.h>
13 #include <semaphore.h>
14 #include <sys/signal.h>
15 #include <signal.h>
16 #include <dirent.h>
17 #include <sys/stat.h>
18 #include <sys/wait.h>
19 #include <pthread.h>
20
21 #define BUFSIZE 10
22 #define TCOUNT 10
23 #define ITEMS 10000
24
25
26 struct item {
27         int value;   // produced value
28         int source;  // index of producer
29 };
30
31 struct buffer {
32   int count;
33   int in;
34   int out;
35   struct item data[BUFSIZE];
36   pthread_mutex_t mutex;
37   pthread_cond_t xp; // producer will wait for a slot using this
38   pthread_cond_t xc; // consumer fill wait for an item using this
39 };
40
41
42 struct buffer *buf;         // shared bounded buffer
43
44 static void *
45 producer_thread(void *arg)
46 {
47
48         int index;  // thread index
49         int i = 0;
50
51         index = (int) arg;
52
53         for (i = 0; i < ITEMS; ++i)  {
54
55                 // value of i is the item produced
56
57                 pthread_mutex_lock(&buf->mutex);
58                 while (buf->count == BUFSIZE)
59                         pthread_cond_wait(&buf->xp, &buf->mutex);
60                 // sleeps in this loop as long as buffer is full
61
```

```
62
63                  // put the item into buffer
64                  buf->data[buf->in].value = i;
65                  buf->data[buf->in].source = index;
66                  buf->in = (buf->in + 1) % BUFSIZE;
67                  buf->count++;
68
69                  if (buf->count == 1)
70                          pthread_cond_signal(&buf->xc);
71
72                  pthread_mutex_unlock(&buf->mutex);
73
74          }
75
76          pthread_exit (NULL);
77  }
78
79
80
81  static void *
82  consumer_thread(void *arg)
83  {
84          int n = 0;
85          int x;
86          int s;
87
88          while (1) {
89                  pthread_mutex_lock(&buf->mutex);
90
91                  while (buf->count == 0)
92                          pthread_cond_wait(&buf->xc, &buf->mutex);
93                  // sleeps in this loop as long as buffer is empty
94
95
96
97                  // retrive an item from buffer
98                  x = buf->data[buf->out].value;
99                  s = buf->data[buf->out].source;
100                 buf->out = (buf->out + 1) % BUFSIZE;
101                 buf->count--;
102
103                 if (buf->count == (BUFSIZE - 1)) {
104                         // wake up all possible workers;
105                         // otherwise some workers will always sleep
106                         pthread_cond_broadcast (&buf->xp);
107                 }
108
109                 pthread_mutex_unlock(&buf->mutex);
110
111                 n++;
112
113                 if (n == (TCOUNT * ITEMS))
114                     break;
115         }
116
117         printf ("retrieved %d items\n", n);
```

```
118
119         printf ("consumer finished successfully\n");
120
121         pthread_exit (NULL);
122 }
123
124
125 int
126 main(int argc, char **argv)
127 {
128
129         pthread_t pids[TCOUNT]; // producer tids
130         pthread_t ctid;         // consumer tid
131         int i;
132         int ret;
133
134
135         buf = (struct buffer *) malloc(sizeof (struct buffer));
136         buf->count = 0;
137         buf->in = 0;
138         buf->out = 0;
139         pthread_mutex_init(&buf->mutex, NULL);
140         pthread_cond_init(&buf->xp, NULL);
141         pthread_cond_init(&buf->xc, NULL);
142
143
144         // create producer threads
145         for (i = 0; i < TCOUNT; ++i) {
146                 ret = pthread_create(&pids[i], NULL, producer_thread,
147                                      (void *) i);
148                 if (ret < 0) {
149                         perror("thread create failed\n");
150                         exit(1);
151                 }
152         }
153
154
155         // create consumer thread
156         ret = pthread_create(&ctid, NULL, consumer_thread, (void *) NULL);
157         if (ret < 0) {
158                 perror("thread create failed\n");
159                 exit(1);
160         }
161
162
163         // wait for the producer threads to terminate
164         for (i = 0; i < TCOUNT; ++i)
165                 pthread_join(pids[i], NULL);
166
167         // wait for the consumer thread to terminate
168         pthread_join(ctid, NULL);
169
170         free(buf);
171
172         pthread_mutex_destroy(&buf->mutex);
173         pthread_cond_destroy(&buf->xp);
```

```
174         pthread_cond_destroy(&buf->xc);
175
176         printf("program ending...bye..\n");
177         exit (0);
178 }
```

Here is a Makefile to compile the programs.

```
1  # $Id: Makefile,v 1.1 2009/03/16 22:05:52 korpe Exp korpe $
2
3  all: mutexcond bcast
4
5  mutexcond: mutexcond.c
6          gcc -Wall -o  mutexcond mutexcond.c -lrt -lpthread
7
8  bcast: bcast.c
9          gcc -Wall -o  bcast bcast.c -lrt -lpthread
10
11
12 clean:
13         rm -fr *~  mutexcond bcast
```

7.2 Peterson's Solution to Critical Region Problem

Peterson's solution is a pure software solution for the critical section problem. It works, however, only for two processes. Lets us identify these processes as i and 1-i, where i can be either 0 or 1. A process i that would like to execute a critical section code has to execute the following entry_section code before executing the critical section code.

```
1                 flag[i] = 1;
2                 turn = 1-i;
3                 while (flag[(1-i)] && turn == (1-i));
```

And after the critical section is executed, the process i has to execute the following exit_section code:

```
1                 flag[i] = 1;
```

There are two processes. The variable i represents the ID of a process. The process ID i can be either 0 or 1. Hence if i is the process ID of one process, then 1-i is the process ID of the other process.

We now show a sample application that uses this solution to synchronize (provide mutual exclusion) two threads while accessing a shared data item. Here the application uses two threads sharing an global integer variable (as the shared data item), but the same solution applies if we would use two processes sharing an integer variable sitting in a shared memory created and used by those processes.

Our program is peterson.c shown below. It declares a global integer variable `balance`, that is to be shared by two threads that will be created. One thread will increment the value of balance N times, and the other thread will decrement the value of the balance again N times. The balance will start with value 0. Hence when those threads executed and finished, the balance should be again 0, if processes are sycnhronized properly and there were no race conditions. If, however, there were race conditions, the result may not be 0.

The global variable N is the number of increments/decrements that will be performed by the threads. Its will be taken as a command-line parameter into the program.

The global variable `use_sync` indicates if we will use synchronization or not, while updating the shared data. If its value is 0 (taken from comand line), then no synchronization will be used (i.e. Peterson's solution will not be used; threads will directly access the shared data and race conditions may occur). If its value if 1, then sychronization is used (before updating the shared data, threads will call entry_section code of Peterson's solution).

The global variables `turn` and `flag` are defined to implement Peterson's solution. Since they are global, they can be shared by threads.

We also defined some additonal structures and global variables to trace the execution of the threads and see later what has happened. The `vtime` (virtual time) variable and `logp` (log pointer) variables are used for this purpose. These are not essential for our program, but are only used for tracing what is happing behind.

Below is the program code (`peterson.c`).

```
1  /* -*- linux-c  -*- */
2  /*    $Id: peterson.c,v 1.7 2009/03/20 16:27:56 korpe Exp korpe $  */
3
4  #include <errno.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10 #include <string.h>
11 #include <malloc.h>
12
13
14 int balance = 0;
15
```

```
16 int N = 3;               /* number of updates*/
17 int use_sync = 0;        /* to synchronize or not */
18 int cs_length = 500000;  /* to prolong the update time */
19
20
21 /* peterson solution shared variables */
22 int turn;
23 int flag[2];
24
25
26 /* logging structures and data */
27 struct  logel {
28         char event[128];
29         struct logel *next;
30 };
31
32 struct log {
33         struct logel *head;
34         struct logel *tail;
35 };
36
37
38 /* these logging variables are also shared;
39    we ignore race conditions on them */
40 int vtime = 0;     /* virtual time */
41 struct log *logp;  /* log list */
42
43
44 /* this can function can be called by two threads at the same time;
45    this may result in a race condtion. we ignore this issue.
46 */
47 void add_log (struct log *l, struct logel *e, int tid, int bal, char *str)
48 {
49         e = (struct logel *) malloc (sizeof(struct logel));
50         sprintf (e->event, "%9d:%5d: balance=%d, mesg=%s\n",
51                  vtime, tid, balance, str);
52         if (l->tail != NULL) {
53                 l->tail->next = e;
54                 l->tail = e;
55         }
56         else {
57                 l->head = e;
58                 l->tail = e;
59         }
60 }
61
62 void print_log (struct log *l)
63 {
64         struct logel *e, *p;
65
66         printf ("    vtime   tid\n");
67         e = l->head;
68         while (e) {
69                 p = e;
70                 printf ("%s", e->event); fflush (stdout);
71                 e = e->next;
```

```
72                  free (p);
73          }
74  }
75
76  /* entry section code */
77  void cs_entry (int i)
78  {
79          flag[i] = 1;
80          turn = 1-i;
81          while (flag[(1-i)] && turn == (1-i)) {
82                  vtime++;
83                  ;
84          }
85  }
86
87
88
89  /* exit section code */
90  void cs_exit (int i)
91  {
92          if (use_sync) {
93                  flag[i] = 0;
94          }
95  }
96
97
98  void * writer (void *arg)
99  {
100         int i;            /* id of the thread: 0 or 1 */
101         int temp, k, c;   /* temporary and loop variables */
102         struct logel *le; /* a log element */
103
104         i = (int) arg;
105
106         for (k = 0; k < N; ++k) {
107                 vtime++;
108
109                 add_log (logp, le, i, balance,  "ready to cs");
110
111                 if (use_sync)
112                         cs_entry(i);
113
114                 /* critical section starts */
115
116                 vtime++;
117
118                 add_log (logp, le, i, balance, "entered cs");
119
120                 temp = balance;
121                 temp = i == 0 ? temp + 1 : temp -1;
122
123                 for (c = 0; c < cs_length; ++c)
124                         ;
125
126                 balance = temp;
127
```

```
128                 vtime++;
129
130                 add_log (logp, le, i, balance, "leaving cs");
131
132                 /* critical section ends */
133
134                 if (use_sync)
135                         cs_exit (i);
136         }
137         pthread_exit (0);
138 }
139
140
141 int
142 main(int argc, char *argv[])
143 {
144         pthread_t tids[2];        /* thread ids */
145         int ret;
146
147
148         if (argc != 4) {
149                 printf ("usage: peterson <num_update> <sync_flag> <duration> \n");
150                 exit (1);
151         }
152
153         N = atoi (argv[1]);
154         use_sync = atoi (argv[2]);
155         cs_length = atoi (argv[3]);
156
157         vtime = 0;
158
159         logp = (struct log *) malloc (sizeof (struct log));
160         logp->head = NULL;
161         logp->tail = NULL;
162
163         ret = pthread_create(&(tids[0]), NULL, writer, (void *) 0);
164         if (ret != 0) {
165                 printf("thread create failed \n");
166                 exit(1);
167         }
168         printf("thread %d with tid %u created\n", 0,
169                         (unsigned int) tids[0]);
170         fflush (stdout);
171
172         ret = pthread_create(&(tids[1]), NULL, writer, (void *) 1);
173         if (ret != 0) {
174                 printf("thread create failed \n");
175                 exit(1);
176         }
177         printf("thread %d with tid %u created\n", 1,
178                (unsigned int) tids[1]);
179         fflush (stdout);
180
181         ret = pthread_join(tids[0], NULL);
182         ret = pthread_join(tids[1], NULL);
183
```

```
184         print_log (logp);
185
186         printf("balance=%d\n", balance); fflush (stdout);
187
188         return 0;
189 }
```

The functions `cs_entry(int i)` and `cs_exit(int i)` implement the entry_section code and exit_section code of Peterson's solution. They need to be called before and after a critical region. The main() function starts by getting program parameters from the command line into some variables: N, use_sync, and cs_length. The cs_length parameter is used to prolong the critical section as much as we want to observe a context switch while a thread is executing inside the critical region. The cs_length value indicates the number of iterations that a for loop performs to prolong the critical section. The main() function creates two threads with IDs 0 and 1 (these are not the thread IDs used by the kernel; these are thread IDs that just make sense in our application). Both threads run the writer() thread start routine. Hence the writer() function is executed in a thread with ID 0 and in a thread with ID 1. The local variable `i` of the writer() function keeps the ID of the thread.

The writer() function is to be executed by two threads concurently. If the thread ID is 0, then writer() increments the balance inside a critical region. The increment is done N times (i.e., critical region is to be entered and left N times). If the thread ID is 1 (i.e. i == 1), then writer() function decrements the shared balance variable in the critical region. If we look more closely to the critical region code:

```
1                   temp = balance;
2                   temp = i == 0 ? temp + 1 : temp -1;
3                   for (c = 0; c < cs_length; ++c) ;
4                   balance = temp;
```

We see that balance is not simply incremented or decremented. It is first copied to a temp variable and temp is incremented/decremented. And then temp is written back to balance. Before that, however, a for loop is executed to prolong the critical region further. How long we prolong the duration of the critical section is determined by the value of the cs_length variable: the number of for loop operations that are not doing anything useful. In this way we can control how long will the critical region last. If it is too short, we may not observe a context switch (from one thread execution in CPU to another thread execution in CPU). Hence we may not observe that race conditions may be happening and appreciate the value of a synchronization tool. If we prolong the critical section duration, we have a good change of having context switching happening inside the critical region and we can have race conditions, if mutual exclusion is not provided.

We also record the running history of two threads into our logp list. The logp is a linked list of events. Each event is indicating what is happening at a time instant (virtual time

is used). For example, at virtual time 517, thread 0 might have entered the critical region and at virtual time 3462, thread 1 might have entered the critical region. These kind of events are recorded there. At the end, when threads have terminated, the log records are printed to screen, so that we can see what has happened during the execution of the threads: which one is run at which time and how balance is updated.

We also include a Makefile that can be used to compile the program and obtain an executable file called `peterson`.

```
1
2  all: peterson
3
4  peterson: peterson.c
5                  gcc -g -Wall -o peterson  peterson.c -lpthread
6
7  clean:
8          rm -fr *~ peterson *.txt core* *.o
9
10
```

Next, we show some sample runs and results of the program. We can run the program with various values of the command line parameters. The program can be started as follows, for example:

```
./peterson 3 1 500000
```

This says that 3 increments and 3 decrements will be performed on a balance that is initially zero; synchronization will be used; and each update will take at least 500000 for-loop-iterations time. If we run the program like this, the following is the output we can have on a PC.

```
1  korpe@pckorpe:$ ./peterson 3 1 500000
2  thread 0 with tid 3085314960 created
3  thread 1 with tid 3076922256 created
4      vtime   tid
5          1:    0: balance=0, mesg=ready to cs
6          2:    0: balance=0, mesg=entered cs
7          3:    0: balance=1, mesg=leaving cs
8          4:    0: balance=1, mesg=ready to cs
9          5:    0: balance=1, mesg=entered cs
10         6:    1: balance=1, mesg=ready to cs
11   9505879:    0: balance=2, mesg=leaving cs
12   9506457:    0: balance=2, mesg=ready to cs
13   9506457:    1: balance=2, mesg=entered cs
14  31677962:    1: balance=1, mesg=leaving cs
15  31678676:    1: balance=1, mesg=ready to cs
16  31678676:    0: balance=1, mesg=entered cs
17  48379549:    0: balance=2, mesg=leaving cs
18  48380253:    1: balance=2, mesg=entered cs
```

```
 48380254:    1: balance=1, mesg=leaving cs
 48380255:    1: balance=1, mesg=ready to cs
 48380256:    1: balance=1, mesg=entered cs
 48380257:    1: balance=0, mesg=leaving cs
balance=0
korpe@pckorpe:$
```

The output shows us what happened during the concurrent execution of two threads. The balance is 0 again at the end. That means we did not have race conditions, and computation is correct. This is achieved by the synchronization tool: Peterson's solution.

First, thread 0 starts running. At virtual time 1 it is about to enter the critical section. At virtual time 2, it entered the critical section. It is allowed to do that since there is no thread in the critical section at that time. Tread 0 executes the critical section and updates the balance to 1. Then it again wants to enter the critical section and enters. But before being able to finish its critical section (to finish the update), a context switch happens and thread 1 runs. Thread 1 wants to enter the critical section. But it can not do that since thread 0 is in the critical section. So it is busy waited at that point by the Peterson's solution. Then after a while later again thread 0 is run and it starts running from where it left off. It was in the critical region. It runs and completes the update and leaves the critical region. The balance is updated to 2. Then it again would like enter the critical section, but can not, since thread 1 is waiting to enter. So thread 1 is allowed to enter the critical region. It updates the balance and decrements it from 2 to 1. Then it continues and would like to enter the critical region again. But it can not do that since thread 0 is waiting to enter the critical section. So thread 1 loops in the entry section until it expires its time slice (time quantum). Then thread 0 is run and allowed to enter the critical region. It decrements the balance from 1 to 0.

And this goes on similarly.

At the end balance reaches to 0, after 3 increments by thread 0 and 3 decrements by thread 1, that are done in a mutually exclusive manner. We did not have race conditions and the result is correct.

Lets see what the output would be if we did not use synchronization. For that we run the program as:

```
./peterson 3 0 500000
```

Namely, synchronization is turned off. Below is the result we get:

```
korpe@pckorpe:$ ./peterson 3 0 500000
thread 0 with tid 3084696464 created
thread 1 with tid 3076303760 created
    vtime   tid
        1:    0: balance=0, mesg=ready to cs
```

```
6          2:    0: balance=0, mesg=entered cs
7          3:    0: balance=1, mesg=leaving cs
8          4:    0: balance=1, mesg=ready to cs
9          5:    0: balance=1, mesg=entered cs
10         6:    1: balance=1, mesg=ready to cs
11         7:    1: balance=1, mesg=entered cs
12         8:    0: balance=2, mesg=leaving cs
13         9:    0: balance=2, mesg=ready to cs
14        10:    0: balance=2, mesg=entered cs
15        11:    1: balance=0, mesg=leaving cs
16        12:    1: balance=0, mesg=ready to cs
17        13:    1: balance=0, mesg=entered cs
18        14:    0: balance=3, mesg=leaving cs
19        15:    1: balance=-1, mesg=leaving cs
20        16:    1: balance=-1, mesg=ready to cs
21        17:    1: balance=-1, mesg=entered cs
22        18:    1: balance=-2, mesg=leaving cs
23 balance=-2
24 korpe@pckorpe:$
25
```

The program took shorter amount of time to execute, but the result is incorrect. It is -2. This is because we had some race conditions. Initially, thread 0 is started running and increments the balance from 0 to 1. While it is in the middle of updating the balance from 1 to 2, a context switched happens and thread 1 is run at virtual time 6. At that time, thread 0 is in the critical section. But since we don't have anything used to protect the critical region, thread 1 starts running at time 6 and directly enters the critical region as well. It now sees the value of balance as 1 and tries to decrement it. But again, before it could complete, thread 0 is run again. It now completed its increment from 1 to 2. Then thread 1 is run again, and completes its decrement. Now balance becomes suddenly 0. We had race condition. We have some more race conditions and finally the result is -2, which is not the expected and correct result. The correct result should be 0.

If we run the program with synchronization turned off, it is possible that we can see a different result each time. Hence the result is inconsistent we say. But with synchronization turned on, however, the result is always the same, hence it is consistent.

Chapter 8

Memory Management

8.1 Building Static Libraries

We will now show how we can develop, build and use a static library. A static library can contain a set of functions that can be called by other applications. The implementation details of the library is hidden from the applications. Only a header file is visible to the applications. An applicaton should incude that header file, and also should link with the libray object file, to use the library. The final executable file contains the object code of the static library functions that are called from the application.

Below we show a very simple sample library and its header file. The file mylib.h is the header file that must be included in an application that would like to use the library. The header file just contains the function prototypes and other definitions/declerations that should be visible to other applications. In effect, the header file is the interface of the library to other applications.

Below is `mylib.h`.

```
1  /* -*- linux-c  -*-   */
2  /* $Id: mylib.h,v 1.3 2009/03/22 21:18:12 korpe Exp korpe $    */
3
4  void mylib_init ();
5
6  void mylib_func1 (char * b);
7
8  void mylib_func2 ();
9
10 void mylib_close ();
```

The mylib.c file implements the library. The program statements implementing the func-

tions should be here. The library can contain some global variables that may not be visible to the application, but can be called from any function of the library. The implementation includes the functions whose prototype is shown in the header file. The implementation code (mylib.c file) may also include some other internal functions that are required to implement the visible functions.

Below is the file `mylib.c`.

```
1  /* -*- linux-c  -*-   */
2  /* $Id: mylib.c,v 1.2 2009/03/22 21:17:09 korpe Exp $    */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7
8
9  int idata; /* mylib integer data; can be anything */
10
11
12 void
13 mylib_init ()
14 {
15         idata = 0;
16
17         printf ("mylib_init called\n");
18         return;
19 }
20
21 void
22 mylib_func1 (char * b)
23 {
24         /* do something with idata */
25
26         /* do something with b */
27
28         printf("mylib_func1 called\n");
29 }
30
31 void
32 mylib_func2 ()
33 {
34         printf("mylib_func2 called\n");
35 }
36
37
38 void
39 mylib_close ()
40 {
41   printf("mylib_close called \n");
42 }
43
```

Next, we show an example simple application (app.c) that is using the library. The

application includes the header file of the library, which is mylib.h. Then it calls the functions implemented in that library.

```
/*   -*-  linux-c   -*-   */
/* $Id: app.c,v 1.2 2009/03/22 21:17:09 korpe Exp $   */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "mylib.h"

int
main()
{
       int i;
       char buf[1024];

       mylib_init();

       mylib_func1 (buf);

       for (i  = 0; i < 10; ++i) {
         mylib_func2 ();
       }

       mylib_close ();

}
```

Below is a Makefile that can be used to compile, build the library and also build the application. A library is nothing, but an object file with an extension .a. Here, our library will be libmylib.a. It is compiled from the source file mylib.c. First, mylib.c is compiled using gcc -c to obtain mylib.o object file. Then, `ar` utility (from archieve) is used to obtain/create the library. After that `ranlib` utility is used to index the library so that finding of library functions happens quickly.

```
# $Id: Makefile,v 1.2 2009/03/22 21:17:09 korpe Exp $


all: libmylib.a app

#rule to obtain the library object file
mylib.o: mylib.c
        gcc -Wall -g -c  mylib.c

#rule to obtain the library
libmylib.a: mylib.o
        ar cr libmylib.a   mylib.o
        ranlib libmylib.a

```

```
15 #rule to obtain the application object file
16 app.o: app.c
17         gcc -Wall -g -c -I. app.c
18
19 #rule to obtain the application executable file
20 app: app.o libmylib.a
21         gcc -Wall -g -o app app.o -I. -L. -lmylib
22
23 clean:
24         rm -fr *.a *.o *~  app libmylib.a
25
```

The library (libmylib.a) is then linked with an application that would like to use it. In this case, it is app.c. The object code (not executable code - not including the library object) of the application can be obtained using the gcc compiler and using the -c option. Then we can link the object code app.o with the library libmylib.a (which is also object code) and obtain the executable file app. The linking is again done with gcc and by using -l option. After -l option we specify the library: like -lmylib.

8.2 Building Shared Libraries

A shared library is a library that is loaded into memory when the program is started to run (process is created). It is not part of the program executable while the program is sitting on the disk. When the program is loaded into memory, then the shared library that the program is referencing to is also loaded and ready to be used by the program. If several programs that are started would like to use the shared library, only one copy of each function implemented in the shared library is loaded into memory. Therefore we save disk and memory space.

Below we show with an example how we can create a shared library. The library code is in mylib.c again. The interface is in mylib.h header file. The header file should be included by an application that would like to use the shared library. In this case it is app.c.

Below is the header file (`mylib.h`).

```
1  /* -*- linux-c  -*-   */
2  /* $Id: mylib.h,v 1.1 2009/03/25 23:34:06 korpe Exp korpe $    */
3
4  void mylib_func1 ();
5
6  void mylib_func2 (char * b);
7
8  void mylib_func3 ();
9
10 void mylib_func4 ();
```

Below is the library source (`mylib.c`).

```
1  /* -*- linux-c  -*-   */
2  /* $Id: mylib.c,v 1.1 2009/03/25 23:34:02 korpe Exp korpe $    */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7
8  void
9  mylib_func1 ()
10 {
11         printf ("mylib_func1 called\n");
12         return;
13 }
14
15 void
16 mylib_func2 (char * b)
17 {
18         /* do something with b */
19
20         printf("mylib_func2 called\n");
21 }
22
23 void
24 mylib_func3 ()
25 {
26         printf("mylib_func3 called\n");
27 }
28
29
30 void
31 mylib_func4 ()
32 {
33   printf("mylib_func4 called \n");
34 }
35
```

Below is an application (`app.c`) that is using the library.

```
1  /*   -*-  linux-c   -*-   */
2  /* $Id: app.c,v 1.1 2009/03/25 23:34:13 korpe Exp korpe $   */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7
8  #include "mylib.h"
9
10 int
11 main()
12 {
```

```
13        int i;
14        char buf[1024];
15
16        mylib_func1();
17
18        mylib_func2 (buf);
19
20        for (i  = 0; i < 10; ++i) {
21          mylib_func3 ();
22        }
23
24        mylib_func4 ();
25
26  }
```

We can now obtain the shared library and link it with our application as follows.

We have to compile the library source to obtain the corresponding object file. This can be done with "gcc -c". But we have to have an additonal option which is "-fPIC". PIC stands for position independent code. So we type:

```
gcc -fPIC -Wall -g -c mylib.c -o libmylib.o
```

This will produce the library object file: libmylib.o. We can now obtain the shared library again using gcc, but using the -shared option. We type:

```
gcc -shared -o libmylib.so libmylib.o
```

and obtain the shared library module: libmylib.so. Our library is ready now.

We can now build our application using the library. We can obtain the object code of the application by:

```
gcc -Wall -g -c -I. app.c
```

And then we can link the application with the shared library using:

```
gcc -Wall -g -o app app.o -I. -L. -lmylib
```

Now, application executable wil be generated. The name of the executable is `app`. It is not containing the object codes of the shared library functions. But the executable has information in it that some shared library functions are called and the application depends on a shared library to execute. When the application is started, that shared library (libmylib.so) has to be loaded. In order to do that, shell has to know the path where the library is sitting. Assume it is in the current directory (which is denoted with .). We have to add that path to the list of paths that are searched when a shared library is to be loaded when an application is started. The list of paths is stored in an environment

variable of the shell, called LD_LIBRARY_PATH. We have to set it so that it also includes the current directory. We can do this by typing the following at the bash shell:

```
export LD_LIBRARY_PATH=./
```

Now, shell knows from where to load the library. If we want to learn to which shared library an application depends on, we can use the ldd command like the following:

```
ldd app
```

The output that we will get is:

```
korpe@pckorpe:$ ldd app
                linux-gate.so.1 =>  (0xffffe000)
                libmylib.so => ./libmylib.so (0xb7f7e000)
                libc.so.6 => /lib/libc.so.6 (0xb
korpe@pckorpe:$
```

The output says that the program ¡tt¿app¡/tt¿ depends on the standard C library (libc.so.6) and on our new library (libmylib.so).

We can now test our application. To do that, we just type:

```
./app
```

And the output will be:

```
korpe@pckorpe:$ ./app
mylib_func1 called
mylib_func2 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func3 called
mylib_func4 called
korpe@pckorpe:$
```

This shows that the shared library is succesfully loaded and its functions are called from the application app while it is executing.

If we look to the object code of our app file, we can see that it is not containing the object code of the functions implemented in the shared library. For that we use the objdump utility. The -S option gives the corresponding 80x86 32 bit assembly code. We type:

```
objdump -S app
```

The following is the output for the section about the functions of shared library:

```
Disassembly of section .plt:

...

x08048474 <mylib_func2@plt>:
 8048474:       ff 25 14 a0 04 08       jmp    *0x804a014
 804847a:       68 28 00 00 00          push   $0x28
 804847f:       e9 90 ff ff ff          jmp    8048414 <_init+0x18>
...
```

As we can see in the output of objdump, the object code of the function implementations are not included. For example the object code of function mylib_func1, are not included. If we would link our library statically, however, we could see the following output showing, for example, the object code for function mylib_func2().

```
08048436 <mylib_func2>:

void
mylib_func2 ()
{
 8048436:       55                      push   %ebp
 8048437:       89 e5                   mov    %esp,%ebp
 8048439:       83 ec 08                sub    $0x8,%esp
        printf("mylib_func2 called\n");
 804843c:       c7 04 24 55 85 04 08    movl   $0x8048555,(%esp)
 8048443:       e8 ac fe ff ff          call   80482f4 <puts@plt>
}
 8048448:       c9                      leave
 8048449:       c3                      ret
```

8.3 Memory Management

8.3.1 Logical Memory Addresses

In Unix or Linux, ee can analyze an executable file or an object file using tools such as `nm`, `objdump`, and `readelf`. Detailed information about their options can be obtained from the respective man pages.

We will now do a simple C-program and object-code analysis which will investigate the logical addresses used by the simple program. The program and a Makefile to compile it are given below. The program is not meant to do any useful task.

```
1  /*   -*-  linux-c   -*-   */
2  /* $Id: app.c,v 1.1 2009/03/31 22:04:21 korpe Exp korpe $   */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7
8  int glob_k;  /* unitialized data - will be placed in .bss section */
9
10 int glob_m = 10;  /* initialized data */
11 int glob_n = 5;   /* initialized data */
12
13 struct node {
14         struct node *left, *right;
15         int value;
16 };
17
18 void
19 func_double ()
20 {
21         glob_m = glob_m + glob_m;
22         glob_n = glob_n + glob_n;
23 }
24
25
26 int
27 main()
28 {
29         int local_i;
30         struct node *np;
31         char *cp;
32         int j;
33
34         printf ("adress of glob_m  = 0x%x\n", (unsigned int) &glob_m);
35         printf ("adress of glob_n  = 0x%x\n", (unsigned int) &glob_n);
36         printf ("adress of glob_k  = 0x%x\n", (unsigned int) &glob_k);
37
38
39         glob_k = 260;
40         cp = (char *)&glob_k;
41         for (j = 0; j < sizeof (unsigned int); ++j) {
42                 printf ("byte[%d]:  value=%d, address=%x\n", j, (unsigned int) (*cp),
43                         (unsigned int)cp);
44                 cp++;
45         }
46
47
48
49         np = (struct node *) malloc (sizeof(struct node));
50
51         printf ("adress of local_i = 0x%x\n", (unsigned int) &local_i);
```

```
52          printf ("address of node pointer np    = 0x%x\n", (unsigned int) &np);
53          printf ("address of node pointed by np = 0x%x\n", (unsigned int) np);
54
55          printf ("heap segment end address      = 0x%x\n", (unsigned int) sbrk(0));
56
57
58          printf ("adress of func1() = 0x%x\n", (unsigned int) &func_double);
59          printf ("adress of main()  = 0x%x\n", (unsigned int) &main);
60
61          for (local_i = 0; local_i < 100; ++local_i) {
62                  glob_m = glob_m + local_i;
63          }
64
65          func_double();
66
67          while (1)
68                  ;
69
70          return 0;
71  }
```

```
1   # $Id: Makefile,v 1.1 2009/03/31 22:04:24 korpe Exp korpe $
2
3
4   all: app
5
6   #rule to obtain the application object file
7   app.o: app.c
8           gcc -Wall  -c -I. app.c
9
10  #rule to obtain the application executable file
11  app: app.o
12          gcc -Wall  -o app app.o
13
14  clean:
15          rm -fr *.a *.o *~  app
16
```

The simple program above just uses some global variables that are initialized, some not initialized, and some local variables. It also uses some pointer variables. It uses malloc to allocate memory from heap. Then it prints out the addresses of various program elements: variables, functions, addresses of dynamically allocated structures, and addresses of pointers. When we compile the program, we obtain an executable file called `app`. Below is the output that we get if we run this program `app`. The addresses are hexadecimal numbers.

```
1   adress of glob_m  = 0x804a020
2   adress of glob_n  = 0x804a024
3   adress of glob_k  = 0x804a02c
4   byte[0]:  value=4, address=804a02c
5   byte[1]:  value=1, address=804a02d
6   byte[2]:  value=0, address=804a02e
```

```
byte[3]:  value=0, address=804a02f
adress of local_i = 0xbfb0ade8
address of node pointer np    = 0xbfb0ade4
address of node pointed by np = 0x804b008
heap segment end address      = 0x806c000
adress of func1() = 0x8048414
adress of main()  = 0x804843f
```

We will now compare this output against the output of `objdump` utility which reads the executable file `app` and gives information about its content. The objdump utility has many options. Some options (-d and -D) can be used even to list the corresponding assembly code of the program including the machine/binary code. For example, the output that we can get using `objdump -D app` will give the assembly code together with the machine code. For example, the machine and assembly code of the function func_double() of the program will look like the following:

```
08048414 <func_double>:
 8048414:       55                      push   %ebp
 8048415:       89 e5                   mov    %esp,%ebp
 8048417:       8b 15 20 a0 04 08       mov    0x804a020,%edx
 804841d:       a1 20 a0 04 08          mov    0x804a020,%eax
 8048422:       8d 04 02                lea    (%edx,%eax,1),%eax
 8048425:       a3 20 a0 04 08          mov    %eax,0x804a020
 804842a:       8b 15 24 a0 04 08       mov    0x804a024,%edx
 8048430:       a1 24 a0 04 08          mov    0x804a024,%eax
 8048435:       8d 04 02                lea    (%edx,%eax,1),%eax
 8048438:       a3 24 a0 04 08          mov    %eax,0x804a024
 804843d:       5d                      pop    %ebp
 804843e:       c3                      ret
```

The corresponding C code is:

```
void
func_double ()
{
        glob_m = glob_m + glob_m;
        glob_n = glob_n + glob_n;
}
```

Basically, the function is just doubling the two global variables. For that, for example, it moves the content of memory address 0x804a020 (a logical address) into a CPU register with the instruction

`mov 0x804a020,%edx`

The (logical) address of this instruction is 0x8048417. At address 0x804a020, we have the global variable glob_m, as can be seen from the program output and in the analysis of the executable file below.

We can now look at the content of the executable file `app` with a different option of objdump: the option `-x`. That option allows us to get some summary information about the sections of the program (like the text section that includes instructions), such as their start (logical) addresses. The output with option also provides information about the symbols used by the program (i.e., variable and function names) and their logical addresses. Hence, if we run `objdump -x app`, we get the following output:

```
app:     file format elf32-i386
app
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048370

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x00000120 memsz 0x00000120 flags r-x
  INTERP off    0x00000154 vaddr 0x08048154 paddr 0x08048154 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000007bc memsz 0x000007bc flags r-x
    LOAD off    0x00000f14 vaddr 0x08049f14 paddr 0x08049f14 align 2**12
         filesz 0x00000114 memsz 0x0000011c flags rw-
 DYNAMIC off    0x00000f28 vaddr 0x08049f28 paddr 0x08049f28 align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000168 vaddr 0x08048168 paddr 0x08048168 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
    NOTE off    0x00000188 vaddr 0x08048188 paddr 0x08048188 align 2**2
         filesz 0x00000018 memsz 0x00000018 flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-
   RELRO off    0x00000f14 vaddr 0x08049f14 paddr 0x08049f14 align 2**0
         filesz 0x000000ec memsz 0x000000ec flags r--

Dynamic Section:
  NEEDED      libc.so.6
  INIT        0x80482f8
  FINI        0x804865c
  HASH        0x80481a0
  STRTAB      0x8048240
  SYMTAB      0x80481d0
  STRSZ       0x58
  SYMENT      0x10
  DEBUG       0x0
  PLTGOT      0x8049ff4
  PLTRELSZ    0x28
  PLTREL      0x11
  JMPREL      0x80482d0
  REL         0x80482c8
  RELSZ       0x8
```

```
  RELENT      0x8
  VERNEED     0x80482a8
  VERNEEDNUM  0x1
  VERSYM      0x8048298

Version References:
  required from libc.so.6:
    0x0d696910 0x00 02 GLIBC_2.0

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048154  08048154  00000154  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048168  08048168  00000168  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.SuSE    00000018  08048188  08048188  00000188  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .hash         00000030  080481a0  080481a0  000001a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynsym       00000070  080481d0  080481d0  000001d0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynstr       00000058  08048240  08048240  00000240  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version  0000000e  08048298  08048298  00000298  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version_r 00000020  080482a8  080482a8  000002a8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.dyn      00000008  080482c8  080482c8  000002c8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rel.plt      00000028  080482d0  080482d0  000002d0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         00000017  080482f8  080482f8  000002f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000060  08048310  08048310  00000310  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         000002ec  08048370  08048370  00000370  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .fini         0000001c  0804865c  0804865c  0000065c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .rodata       0000013e  08048678  08048678  00000678  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .eh_frame     00000004  080487b8  080487b8  000007b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .init_array   00000000  08049f14  0804a028  00001028  2**0
                  CONTENTS, ALLOC, LOAD, DATA
 17 .ctors        00000008  08049f14  08049f14  00000f14  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 18 .dtors        00000008  08049f1c  08049f1c  00000f1c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 19 .jcr          00000004  08049f24  08049f24  00000f24  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 20 .dynamic      000000c8  08049f28  08049f28  00000f28  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 21 .got          00000004  08049ff0  08049ff0  00000ff0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 22 .got.plt      00000020  08049ff4  08049ff4  00000ff4  2**2
```

```
                  CONTENTS, ALLOC, LOAD, DATA
 23 .data         00000014  0804a014  0804a014  00001014  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          00000008  0804a028  0804a028  00001028  2**2
                  ALLOC
 25 .comment      00000173  00000000  00000000  00001028  2**0
                  CONTENTS, READONLY
 26 .debug_aranges 00000058  00000000  00000000  000011a0  2**3
                  CONTENTS, READONLY, DEBUGGING
 27 .debug_pubnames 00000025  00000000  00000000  000011f8  2**0
                  CONTENTS, READONLY, DEBUGGING
 28 .debug_info   00000199  00000000  00000000  0000121d  2**0
                  CONTENTS, READONLY, DEBUGGING
 29 .debug_abbrev 00000066  00000000  00000000  000013b6  2**0
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_line   00000137  00000000  00000000  0000141c  2**0
                  CONTENTS, READONLY, DEBUGGING
 31 .debug_str    000000bb  00000000  00000000  00001553  2**0
                  CONTENTS, READONLY, DEBUGGING
 32 .debug_ranges 00000048  00000000  00000000  00001610  2**3
                  CONTENTS, READONLY, DEBUGGING
SYMBOL TABLE:
08048154 l    d  .interp        00000000              .interp
08048168 l    d  .note.ABI-tag        00000000              .note.ABI-tag
08048188 l    d  .note.SuSE        00000000              .note.SuSE
080481a0 l    d  .hash        00000000              .hash
080481d0 l    d  .dynsym        00000000              .dynsym
08048240 l    d  .dynstr        00000000              .dynstr
08048298 l    d  .gnu.version        00000000              .gnu.version
080482a8 l    d  .gnu.version_r        00000000              .gnu.version_r
080482c8 l    d  .rel.dyn        00000000              .rel.dyn
080482d0 l    d  .rel.plt        00000000              .rel.plt
080482f8 l    d  .init        00000000              .init
08048310 l    d  .plt        00000000              .plt
08048370 l    d  .text        00000000              .text
0804865c l    d  .fini        00000000              .fini
08048678 l    d  .rodata        00000000              .rodata
080487b8 l    d  .eh_frame        00000000              .eh_frame
08049f14 l    d  .init_array        00000000              .init_array
08049f14 l    d  .ctors        00000000              .ctors
08049f1c l    d  .dtors        00000000              .dtors
08049f24 l    d  .jcr        00000000              .jcr
08049f28 l    d  .dynamic        00000000              .dynamic
08049ff0 l    d  .got        00000000              .got
08049ff4 l    d  .got.plt        00000000              .got.plt
0804a014 l    d  .data        00000000              .data
0804a028 l    d  .bss        00000000              .bss
00000000 l    d  .comment        00000000              .comment
00000000 l    d  .debug_aranges        00000000              .debug_aranges
00000000 l    d  .debug_pubnames        00000000              .debug_pubnames
00000000 l    d  .debug_info        00000000              .debug_info
00000000 l    d  .debug_abbrev        00000000              .debug_abbrev
00000000 l    d  .debug_line        00000000              .debug_line
00000000 l    d  .debug_str        00000000              .debug_str
00000000 l    d  .debug_ranges        00000000              .debug_ranges
00000000 l    df *ABS*        00000000              abi-note.S
```

```
00000000 l    df *ABS*        00000000              suse-note.S
00000000 l    df *ABS*        00000000              ../sysdeps/i386/elf/start.S
00000000 l    df *ABS*        00000000              init.c
00000000 l    df *ABS*        00000000              initfini.c
00000000 l    df *ABS*        00000000              /usr/src/packages/BUILD/glibc-2.5/cc-nptl/csu/crti.S
08048394 l     F .text        00000000              call_gmon_start
00000000 l    df *ABS*        00000000              crtstuff.c
08049f14 l     O .ctors        00000000              __CTOR_LIST__
08049f1c l     O .dtors        00000000              __DTOR_LIST__
08049f24 l     O .jcr        00000000              __JCR_LIST__
0804a028 l     O .bss        00000001              completed.5752
0804a01c l     O .data        00000000              p.5750
080483c0 l     F .text        00000000              __do_global_dtors_aux
080483f0 l     F .text        00000000              frame_dummy
00000000 l    df *ABS*        00000000              crtstuff.c
08049f18 l     O .ctors        00000000              __CTOR_END__
08049f20 l     O .dtors        00000000              __DTOR_END__
080487b8 l     O .eh_frame        00000000              __FRAME_END__
08049f24 l     O .jcr        00000000              __JCR_END__
08048630 l     F .text        00000000              __do_global_ctors_aux
00000000 l    df *ABS*        00000000              initfini.c
00000000 l    df *ABS*        00000000              /usr/src/packages/BUILD/glibc-2.5/cc-nptl/csu/crtn.S
00000000 l    df *ABS*        00000000              app.c
08049ff4 l     O .got.plt        00000000              .hidden _GLOBAL_OFFSET_TABLE_
08049f14 l       .init_array        00000000              .hidden __init_array_end
08049f14 l       .init_array        00000000              .hidden __init_array_start
08049f28 l     O .dynamic        00000000              .hidden _DYNAMIC
0804a014  w      .data        00000000              data_start
080485b0 g     F .text        00000005              __libc_csu_fini
08048370 g     F .text        00000000              _start
0804a020 g     O .data        00000004              glob_m
00000000  w      *UND*        00000000              __gmon_start__
00000000  w      *UND*        00000000              _Jv_RegisterClasses
08048678 g     O .rodata        00000004              _fp_hw
0804865c g     F .fini        00000000              _fini
08048414 g     F .text        0000002b              func_double
00000000       F *UND*        0000019f              __libc_start_main@@GLIBC_2.0
0804867c g     O .rodata        00000004              _IO_stdin_used
0804a024 g     O .data        00000004              glob_n
0804a014 g       .data        00000000              __data_start
00000000       F *UND*        00000073              sbrk@@GLIBC_2.0
0804a018 g     O .data        00000000              .hidden __dso_handle
080485c0 g     F .text        00000069              __libc_csu_init
00000000       F *UND*        00000039              printf@@GLIBC_2.0
0804a02c g     O .bss        00000004              glob_k
0804a028 g       *ABS*        00000000              __bss_start
00000000       F *UND*        00000181              malloc@@GLIBC_2.0
0804a030 g       *ABS*        00000000              _end
0804a028 g       *ABS*        00000000              _edata
08048629 g     F .text        00000000              .hidden __i686.get_pc_thunk.bx
0804843f g     F .text        0000016f              main
080482f8 g     F .init        00000000              _init
```

The output shows that the .text section starts at (logical) address 0x08048310 (hexadecimal number). The .data section starts at logical address 0x0804a014, and the uninitialized

data section (.bss) starts at logical address 0x0804a028. The `readelf -S app` will also give information about the sections, including their sizes in bytes. If we execute that command, we see the size of .text section is 0x2ec bytes (492 bytes), the size of .data section is 0x14 (20) bytes and the size of .bss section is 8 bytes.

```
Section Headers:
  [Nr] Name Type      Addr     Off    Size   ES Flg Lk Inf Al
  ...
  ...
  [13] .text PROGBITS 08048370 000370 0002ec 00  AX 0   0 16
  ...
  [24] .data PROGBITS 0804a014 001014 000014 00  WA 0   0  4
  [25] .bss  NOBITS   0804a028 001028 000008 00  WA 0   0  4
  ...
  ...
```

Hence the program's partial layout in its logical address space seems to be like this:

```
Address         : What is included there
-----------------------------------------------------------------
...
0x08048370      : .text starts (instructions start)

                  text section includes main() and func_double()

0x08048370+0x2ec: .text ends
-----------------------------------------------------------------
0x0804a014      : .data starts
                  global variables (glob_m and glob_n) are here
0804a014+0x14   : .data ends
0804a028        : .bss (uninitialized data) starts
                  the unitialized global variable (glob_k) is here
0804a028+8      : .bss ends
-----------------------------------------------------------------
....
                : dynamically allocated variables sit here
                  (variables allocated space with malloc)
....
0x806c000       : end of heap section of the program
-----------------------------------------------------------------
...

...
....
-----------------------------------------------------------------
                : stack
-----------------------------------------------------------------
```

We can see the output of `objdump -x` above and find out the symbol values there for various symbols of the program. That means we can see the the logical addresses (symbol

values) that are assigned to various variables and functions of the program. For example, the addres of the main() functions is 0x0804843f. The program output (i.e. output when we run program app) also shows the same address for the main function. The address of the func_double() is 0x08048414, as it is the case also in the program output.

The addresses of the global variables glob_m and glob_n are 804a020 and 0x804a024, respectively. This can be verified by looking to the output of `objdump -x app` and `app`. The address of the glob_k variable is 0x804a02c. It is an uninitialized global variable. Hence it is located in the .bss segement. As can be verified, that address falls into the address range assigned to the .bss segment (range is from 0804a028 to 0804a030).

In the program app, the integer global variable glob_k is set to the value 260 before being printed out. The output of the program app shows this fact. The value of glob_k is 0x00000104 in hexadecimal. Hence the byte[0] of that integer contains vaue 0x04. Byte[1] contains value 0x01. The other bytes contain value zero. The address of those bytes are also printed. The least significant byte of the integer has the smallest address value. This implies that the byte order of this machine is litte-endian. In fact, x86 architecture is using little-endian byte order. This is verified with this output.

The address of the pointer variable np and the address of the structure pointed by that variable are worth to investigate. The pointer variable np has address 0xbfb0ade4, which is an address from the stack section that is generated at run time. No such section needs to be existing in the executable file, which is not running. Stack is only needed at run time. Since the variables np, local_i, j, and cp are defined as local variables in the main() function, they are allocated space from the stack section, not from the .data section. In the stack, np variable comes just after the variable local_i. This can be verified by looking to their addresses. The variable local_i has address 0xbfb0ade8, and the variable np has address 0xbfb0ade4. Since local_i comes first in the program, it is pushed earlier, hence has a bigger address, since the stack grows downward (from bigger to smaller addresses). Hence the address of np is 4 bytes smaller than the address of local_i.

The address of the structure that is pointed by the pointer np, however, does not seem to be a stack address. The address of the structure (the value of the pointer variable) is 0x804b008. It is a much smallar address than the addresses of the stack variables. That address sits somewhere between the .bss section and stack section. It sits in a section that we call heap section. The end of the heap section can be obtained by using the sbrk(0) function. The program actually calls that and prints out the value. It is 0x806c000. In fact, the address of the structure, which is 0x804b008, is less than the address 0x806c000 (end of heap section). sometimes the .data, .bss and heap sections together are called the data segment that can be grown on demand, when more heap memory is allocated using malloc. Hence we can say that sbrk(0) indicates the end of the data segment of the program.

8.3.2 Logical Memory Regions of a Process

In Linux, we can use the /proc file system to obtain information about the currently started processes. If we go to the `/proc` directory, we will see lots of numbers. These are the process IDs of the currently started processes. We can change to a directory with a pid as a name. There we can look to the content of a file called `maps`. It will tell us the start addresses of various used regions (sections) of the logical memory of the process.

8.3.3 Kernel Memory Allocation

Kernel allocates memory using slab allocator scheme. In that, objects of the same size (type) are put into spaces called caches. A cache consists of one or more contigous physical memory areas called slabs. A slab consists of one or more page frames that are contigous. Each cache stores a different type of object. For example, we have a cache that stores process descriptors (called PCBs; `struct task_struct` structures); we have a cache that stores directrory entry objects, and so on.

All caches that are used at a given in the kernel can be seen by examining the file `/proc/slabinfo`. You can view the content of the file by ordinary `cat` command. Type `cat /proc/slabinfo`.

An output like the following will be dispayed:

```
# name            "active_objs" "num_objs" "objsize" "objperslab"
  "pagesperslab" : tunables "limit" "batchcount" "sharedfactor" : slabdata "active_slabs" "num_slabs" "sharedav
ip_fib_alias          15    113     32  113    1 : tunables  120   60
  8 : slabdata      1      1      0
ip_fib_hash           15    113     32  113    1 : tunables  120   60
  8 : slabdata      1      1      0
dm_tio                 0      0     16  203    1 : tunables  120   60
  8 : slabdata      0      0      0
dm_io                  0      0     20  169    1 : tunables  120   60
  8 : slabdata      0      0      0

...

inode_cache         1198   1628    352   11    1 : tunables   54   27
8 : slabdata    148    148      0
dentry_cache      558056 562020    132   29    1 : tunables  120   60
8 : slabdata  19380  19380      0
filp               3120   3120    192   20    1 : tunables  120   60
8 : slabdata    156    156      0
names_cache           25     25   4096    1    1 : tunables   24   12
8 : slabdata     25     25      0
key_jar               16     30    128   30    1 : tunables  120   60
8 : slabdata      1      1      0
idr_layer_cache      359    377    136   29    1 : tunables  120   60
8 : slabdata     13     13      0
```

```
buffer_head       152992 203976     52   72    1 : tunables  120   60
8 : slabdata   2833   2833      0
mm_struct            123    135    448    9    1 : tunables   54   27
8 : slabdata     15     15      0
vm_area_struct      9611   9752     84   46    1 : tunables  120   60
8 : slabdata    212    212      0
fs_cache              98    177     64   59    1 : tunables  120   60
8 : slabdata      3      3      0
files_cache           97    170    384   10    1 : tunables   54   27
8 : slabdata     17     17      0
signal_cache         127    144    448    9    1 : tunables   54   27
8 : slabdata     16     16      0
sighand_cache        132    132   1344    3    1 : tunables   24   12
8 : slabdata     44     44      0
task_struct          176    195   1392    5    2 : tunables   24   12
8 : slabdata     39     39      0
anon_vma            2540   2540     12  254    1 : tunables  120   60
8 : slabdata     10     10      0
pgd                   97     97   4096    1    1 : tunables   24   12
8 : slabdata     97     97      0
pid                  180    303     36  101    1 : tunables  120   60
8 : slabdata      3      3      0


....

size-64             3386   3953     64   59    1 : tunables  120   60
8 : slabdata     67     67     15
size-32             6893   6893     32  113    1 : tunables  120   60
8 : slabdata     61     61      0
```

In this output we see information about the caches: their names, the objects size of the objects stored in each cache, the number of objects that can be stored in a cache, the number of active objects, etc. For example, the cache for storing PCB structures is called `task_struct`, and the size of each object that can be stored in that cache is 1392 bytes. The cache can hold 195 objects. Currently there are 176 objects that are stored (active objects). The cache contains 39 slabs. Each slab can contain 5 objects.

Similarly, there are other caches. There is a cache to store inodes, a cache to store directory entries, a cache to store page global directories (top level page tables), a cache to store 64-byte objects, a cache to store 32 byte objects, and so on.

8.4 Page Tables

Virtual addresses are translated into physical addresses using page tables. We will now describe a Linux module that dumps the page table of a process into a kernel log file. The module is written for 32 bit Intel architecture (i386, x86, or x86_64 architecture running in 32 bit mode). Those architectures are using two level page tables for a process.

Below is the module code.

```
1  /* -*- linux-c  -*- */
2  /* $Id: page.c,v 1.11 2009/05/06 08:26:30 korpe Exp korpe $ */
3
4  #include <linux/highmem.h>
5  #include <linux/module.h>
6  #include <linux/kernel.h>
7  #include <linux/init.h>
8  #include <linux/sched.h>
9  #include <linux/mm.h>
10 #include <asm/page.h>
11
12
13
14 /* PD: page table dump module */
15
16 #define PD_PAGESIZE 4096
17
18 #define RESULT_SPACE   8192 /* bytes */
19
20 struct pf_map {
21         unsigned int outer_index;
22         unsigned int inner_index;
23         unsigned long page_num;
24         unsigned long frame_num;
25 };
26
27 static struct pf_map *map;
28
29 int map_count = 0;
30
31 #define RESULT_SIZE ((RESULT_SPACE / sizeof(struct pf_map)))
32
33
34 static int my_pid = 1000;
35 module_param (my_pid, int, 0);
36
37
38
39 static void print_result (void)
40 {
41         int i;
42
43         for (i = 0; i < map_count; ++i) {
44                 printk (KERN_INFO "(%u,%u)p#=%lu,va=%p f#=%lu,pa=%p\n",
45                         map[i].outer_index,
46                         map[i].inner_index,
47                         map[i].page_num,
48                         (void*)(map[i].page_num * 4096),
49                         map[i].frame_num,
50                         (void*)(map[i].frame_num * 4096) );
51         }
52
53         printk (KERN_INFO "number of page frames allocated = %u\n",
54                 map_count);
```

```
55
56          if (map != 0)
57                  kfree ( (void*) map);
58  }
59
60
61  static void dump_inner_table (unsigned long va, unsigned int pgd_index)
62  {
63          int i;
64          pte_t *p; /* pointer to second level page table */
65          unsigned  long  pt_entry;
66          unsigned  long  framenum;
67          unsigned  long  pagenum;
68
69          p = (pte_t *) va;
70
71
72          for (i = 0; i < 1024; ++i) {
73
74                  pt_entry = pte_val(p[i]);
75
76                  if ((pt_entry % 2) == 1)  { /* checking Present flag */
77                          framenum = pt_entry >> 12;
78                          pagenum  = pgd_index * 1024 + i;
79
80                          if (map_count < RESULT_SIZE) {
81                                  map[map_count].outer_index = pgd_index;
82                                  map[map_count].inner_index = i;
83                                  map[map_count].page_num = pagenum;
84                                  map[map_count].frame_num = framenum;
85
86                                  map_count++;
87                          }
88                          else {
89                                  printk (KERN_INFO "result table full \n");
90                                  break;
91                          }
92
93                  }
94
95          }
96  }
97
98
99
100 static void dump_pgd (struct task_struct *task)
101 {
102         unsigned int i;
103         unsigned long pgd_entry;
104         unsigned long framenum;
105         unsigned long va;      /* virtual address */
106
107         for (i = 0; i  < 768; ++i) {
108
109                 pgd_entry = pgd_val(task->mm->pgd[i]);
110
```

```
111                 if ( (pgd_entry % 2) == 1 ) { /* checking Present flag */
112
113                         framenum = pgd_entry >> 12;
114
115                         va  = (unsigned int) kmap ( mem_map + framenum);
116
117                         dump_inner_table (va, i);
118
119                         kunmap ( mem_map + framenum);
120                 }
121         }
122 }
123
124
125 int init_module (void)
126 {
127         struct task_struct *task;
128
129         printk (KERN_INFO "module started: page table dump\n");
130         printk (KERN_INFO "process pid = %u\n",  my_pid);
131
132
133         map = (struct pf_map *)  kmalloc (RESULT_SPACE, GFP_KERNEL);
134
135         if (map == 0) {
136                 printk (KERN_INFO "can not allocate memory \n");
137                 return;
138         }
139
140         task = &init_task;
141
142         do {
143                 if (task->pid == my_pid) {
144
145                         dump_pgd(task);
146
147                         break;
148                 }
149
150         } while ( (task = next_task(task)) != &init_task );
151
152
153
154         print_result ();
155
156         return 0;
157 }
158
159 void cleanup_module (void)
160 {
161
162         printk (KERN_INFO "module ending:  page table dump\n");
163 }
164
165
166
```

```
167
168
```

Below is a Makefile that will compile the module (Linux kernel version 2.6).

```
1  obj-m += page.o
2
3
4  all:
5          make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
6          gcc -o app -g app.c
7
8  clean:
9          make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
10         rm -fr app *~
```

The module is using a parameter `my_pid`. Hence while loading the module, we have to specify a pid of a process that is started. Then the module will dump the page table info of that process into the kernel log file: `/var/log/messages`.

We can load the module as follows:

```
sudo insmod page.ko my_pid=apid
```

We can remove the module as follows:

```
sudo rmmod page
```

We can see the dumped information at the tail of the kernel log file. The followinig command will show the last 200 lines of the file:

```
tail /var/log/messages -n 200
```

We can obtain such an output there.

```
1  May  6 11:24:45 korpe-desktop kernel: [770766.884652] module started: page table dump
2  May  6 11:24:45 korpe-desktop kernel: [770766.884655] process pid = 13556
3  May  6 11:24:45 korpe-desktop kernel: [770766.884694] (32,72)p#=32840,va=08048000 f#=382204,pa=5d4fc000
4  May  6 11:24:45 korpe-desktop kernel: [770766.884696] (32,73)p#=32841,va=08049000 f#=421643,pa=66f0b000
5  May  6 11:24:45 korpe-desktop kernel: [770766.884698] (32,74)p#=32842,va=0804a000 f#=421659,pa=66f1b000
6  May  6 11:24:45 korpe-desktop kernel: [770766.884700] (39,15)p#=39951,va=09c0f000 f#=389046,pa=5efb6000
7  May  6 11:24:45 korpe-desktop kernel: [770766.884703] (39,16)p#=39952,va=09c10000 f#=380918,pa=5cff6000
8  May  6 11:24:45 korpe-desktop kernel: [770766.884705] (39,17)p#=39953,va=09c11000 f#=394630,pa=60586000
9  May  6 11:24:45 korpe-desktop kernel: [770766.884707] (39,18)p#=39954,va=09c12000 f#=383437,pa=5d9cd000
10 May  6 11:24:45 korpe-desktop kernel: [770766.884709] (39,19)p#=39955,va=09c13000 f#=377581,pa=5c2ed000
11 May  6 11:24:45 korpe-desktop kernel: [770766.884712] (39,20)p#=39956,va=09c14000 f#=399547,pa=618bb000
12 May  6 11:24:45 korpe-desktop kernel: [770766.884714] (735,586)p#=753226,va=b7e4a000 f#=362379,pa=5878b000
13 May  6 11:24:45 korpe-desktop kernel: [770766.884716] (735,587)p#=753227,va=b7e4b000 f#=512596,pa=7d254000
```

```
May  6 11:24:45 korpe-desktop kernel: [770766.884719] (735,588)p#=753228,va=b7e4c000 f#=512025,pa=7d019000
May  6 11:24:45 korpe-desktop kernel: [770766.884721] (735,589)p#=753229,va=b7e4d000 f#=512423,pa=7d1a7000
May  6 11:24:45 korpe-desktop kernel: [770766.884723] (735,590)p#=753230,va=b7e4e000 f#=512026,pa=7d01a000
May  6 11:24:45 korpe-desktop kernel: [770766.884725] (735,591)p#=753231,va=b7e4f000 f#=512035,pa=7d023000
May  6 11:24:45 korpe-desktop kernel: [770766.884727] (735,592)p#=753232,va=b7e50000 f#=512034,pa=7d022000
May  6 11:24:45 korpe-desktop kernel: [770766.884730] (735,593)p#=753233,va=b7e51000 f#=507527,pa=7be87000
May  6 11:24:45 korpe-desktop kernel: [770766.884732] (735,594)p#=753234,va=b7e52000 f#=507526,pa=7be86000
May  6 11:24:45 korpe-desktop kernel: [770766.884734] (735,595)p#=753235,va=b7e53000 f#=512427,pa=7d1ab000
May  6 11:24:45 korpe-desktop kernel: [770766.884736] (735,596)p#=753236,va=b7e54000 f#=512426,pa=7d1aa000
May  6 11:24:45 korpe-desktop kernel: [770766.884807] (735,597)p#=753237,va=b7e55000 f#=512599,pa=7d257000
May  6 11:24:45 korpe-desktop kernel: [770766.884809] (735,598)p#=753238,va=b7e56000 f#=512598,pa=7d256000
May  6 11:24:45 korpe-desktop kernel: [770766.884812] (735,599)p#=753239,va=b7e57000 f#=512499,pa=7d1f3000
May  6 11:24:45 korpe-desktop kernel: [770766.884814] (735,600)p#=753240,va=b7e58000 f#=512498,pa=7d1f2000
May  6 11:24:45 korpe-desktop kernel: [770766.884816] (735,601)p#=753241,va=b7e59000 f#=512429,pa=7d1ad000
May  6 11:24:45 korpe-desktop kernel: [770766.884819] (735,602)p#=753242,va=b7e5a000 f#=512428,pa=7d1ac000
May  6 11:24:45 korpe-desktop kernel: [770766.884821] (735,603)p#=753243,va=b7e5b000 f#=512487,pa=7d1e7000
May  6 11:24:45 korpe-desktop kernel: [770766.884823] (735,604)p#=753244,va=b7e5c000 f#=512486,pa=7d1e6000
May  6 11:24:45 korpe-desktop kernel: [770766.884826] (735,605)p#=753245,va=b7e5d000 f#=512591,pa=7d24f000
May  6 11:24:45 korpe-desktop kernel: [770766.884828] (735,606)p#=753246,va=b7e5e000 f#=512590,pa=7d24e000
May  6 11:24:45 korpe-desktop kernel: [770766.884830] (735,607)p#=753247,va=b7e5f000 f#=512603,pa=7d25b000
May  6 11:24:45 korpe-desktop kernel: [770766.884833] (735,608)p#=753248,va=b7e60000 f#=512602,pa=7d25a000
May  6 11:24:45 korpe-desktop kernel: [770766.884835] (735,609)p#=753249,va=b7e61000 f#=512483,pa=7d1e3000
May  6 11:24:45 korpe-desktop kernel: [770766.884837] (735,630)p#=753270,va=b7e76000 f#=512017,pa=7d011000
May  6 11:24:45 korpe-desktop kernel: [770766.884839] (735,633)p#=753273,va=b7e79000 f#=512482,pa=7d1e2000
May  6 11:24:45 korpe-desktop kernel: [770766.884841] (735,634)p#=753274,va=b7e7a000 f#=512009,pa=7d009000
May  6 11:24:45 korpe-desktop kernel: [770766.884844] (735,649)p#=753289,va=b7e89000 f#=507439,pa=7be2f000
May  6 11:24:45 korpe-desktop kernel: [770766.884846] (735,650)p#=753290,va=b7e8a000 f#=507438,pa=7be2e000
May  6 11:24:45 korpe-desktop kernel: [770766.884848] (735,651)p#=753291,va=b7e8b000 f#=507437,pa=7be2d000
May  6 11:24:45 korpe-desktop kernel: [770766.884850] (735,652)p#=753292,va=b7e8c000 f#=507436,pa=7be2c000
May  6 11:24:45 korpe-desktop kernel: [770766.884853] (735,653)p#=753293,va=b7e8d000 f#=507895,pa=7bff7000
May  6 11:24:45 korpe-desktop kernel: [770766.884855] (735,655)p#=753295,va=b7e8f000 f#=507893,pa=7bff5000
May  6 11:24:45 korpe-desktop kernel: [770766.884857] (735,660)p#=753300,va=b7e94000 f#=512493,pa=7d1ed000
May  6 11:24:45 korpe-desktop kernel: [770766.884859] (735,681)p#=753321,va=b7ea9000 f#=512256,pa=7d100000
May  6 11:24:45 korpe-desktop kernel: [770766.884861] (735,684)p#=753324,va=b7eac000 f#=512361,pa=7d169000
May  6 11:24:45 korpe-desktop kernel: [770766.884864] (735,693)p#=753333,va=b7eb5000 f#=512146,pa=7d092000
May  6 11:24:45 korpe-desktop kernel: [770766.884866] (735,694)p#=753334,va=b7eb6000 f#=507480,pa=7be58000
May  6 11:24:45 korpe-desktop kernel: [770766.884868] (735,695)p#=753335,va=b7eb7000 f#=512393,pa=7d189000
May  6 11:24:45 korpe-desktop kernel: [770766.884870] (735,696)p#=753336,va=b7eb8000 f#=512000,pa=7d000000
May  6 11:24:45 korpe-desktop kernel: [770766.884873] (735,697)p#=753337,va=b7eb9000 f#=507486,pa=7be5e000
May  6 11:24:45 korpe-desktop kernel: [770766.884875] (735,698)p#=753338,va=b7eba000 f#=507876,pa=7bfe4000
May  6 11:24:45 korpe-desktop kernel: [770766.884877] (735,699)p#=753339,va=b7ebb000 f#=507798,pa=7bf96000
May  6 11:24:45 korpe-desktop kernel: [770766.884879] (735,700)p#=753340,va=b7ebc000 f#=507472,pa=7be50000
May  6 11:24:45 korpe-desktop kernel: [770766.884882] (735,701)p#=753341,va=b7ebd000 f#=512424,pa=7d1a8000
May  6 11:24:45 korpe-desktop kernel: [770766.884884] (735,702)p#=753342,va=b7ebe000 f#=512012,pa=7d00c000
May  6 11:24:45 korpe-desktop kernel: [770766.884886] (735,703)p#=753343,va=b7ebf000 f#=507891,pa=7bff3000
May  6 11:24:45 korpe-desktop kernel: [770766.884888] (735,706)p#=753346,va=b7ec2000 f#=507578,pa=7beba000
May  6 11:24:45 korpe-desktop kernel: [770766.884890] (735,707)p#=753347,va=b7ec3000 f#=512030,pa=7d01e000
May  6 11:24:45 korpe-desktop kernel: [770766.884893] (735,708)p#=753348,va=b7ec4000 f#=512023,pa=7d017000
May  6 11:24:45 korpe-desktop kernel: [770766.884895] (735,743)p#=753383,va=b7ee7000 f#=512588,pa=7d24c000
May  6 11:24:45 korpe-desktop kernel: [770766.884898] (735,794)p#=753434,va=b7f1a000 f#=507585,pa=7bec1000
May  6 11:24:45 korpe-desktop kernel: [770766.884900] (735,796)p#=753436,va=b7f1c000 f#=512469,pa=7d1d5000
May  6 11:24:45 korpe-desktop kernel: [770766.884902] (735,804)p#=753444,va=b7f24000 f#=507495,pa=7be67000
May  6 11:24:45 korpe-desktop kernel: [770766.884904] (735,808)p#=753448,va=b7f28000 f#=507483,pa=7be5b000
May  6 11:24:45 korpe-desktop kernel: [770766.884907] (735,812)p#=753452,va=b7f2c000 f#=512005,pa=7d005000
May  6 11:24:45 korpe-desktop kernel: [770766.884909] (735,826)p#=753466,va=b7f3a000 f#=507589,pa=7bec5000
May  6 11:24:45 korpe-desktop kernel: [770766.884911] (735,871)p#=753511,va=b7f67000 f#=507530,pa=7be8a000
May  6 11:24:45 korpe-desktop kernel: [770766.884913] (735,872)p#=753512,va=b7f68000 f#=507433,pa=7be29000
May  6 11:24:45 korpe-desktop kernel: [770766.884916] (735,875)p#=753515,va=b7f6b000 f#=512418,pa=7d1a2000
May  6 11:24:45 korpe-desktop kernel: [770766.884918] (735,897)p#=753537,va=b7f81000 f#=507522,pa=7be82000
May  6 11:24:45 korpe-desktop kernel: [770766.884921] (735,898)p#=753538,va=b7f82000 f#=507521,pa=7be81000
May  6 11:24:45 korpe-desktop kernel: [770766.884923] (735,931)p#=753571,va=b7fa3000 f#=393626,pa=6019a000
May  6 11:24:45 korpe-desktop kernel: [770766.884925] (735,932)p#=753572,va=b7fa4000 f#=377501,pa=5c29d000
May  6 11:24:45 korpe-desktop kernel: [770766.884927] (735,933)p#=753573,va=b7fa5000 f#=369901,pa=5a4ed000
May  6 11:24:45 korpe-desktop kernel: [770766.884930] (735,934)p#=753574,va=b7fa6000 f#=366932,pa=59954000
May  6 11:24:45 korpe-desktop kernel: [770766.884932] (735,936)p#=753576,va=b7fa8000 f#=374924,pa=5b88c000
```

```
May  6 11:24:45 korpe-desktop kernel: [770766.884934] (735,949)p#=753589,va=b7fb5000 f#=394631,pa=60587000
May  6 11:24:45 korpe-desktop kernel: [770766.884936] (735,950)p#=753590,va=b7fb6000 f#=421699,pa=66f43000
May  6 11:24:45 korpe-desktop kernel: [770766.884939] (735,951)p#=753591,va=b7fb7000 f#=363843,pa=58d43000
May  6 11:24:45 korpe-desktop kernel: [770766.884941] (735,952)p#=753592,va=b7fb8000 f#=507457,pa=7be41000
May  6 11:24:45 korpe-desktop kernel: [770766.884943] (735,953)p#=753593,va=b7fb9000 f#=512007,pa=7d007000
May  6 11:24:45 korpe-desktop kernel: [770766.884946] (735,954)p#=753594,va=b7fba000 f#=507883,pa=7bfeb000
May  6 11:24:45 korpe-desktop kernel: [770766.884948] (735,955)p#=753595,va=b7fbb000 f#=512006,pa=7d006000
May  6 11:24:45 korpe-desktop kernel: [770766.884950] (735,956)p#=753596,va=b7fbc000 f#=507579,pa=7bebb000
May  6 11:24:45 korpe-desktop kernel: [770766.884953] (735,957)p#=753597,va=b7fbd000 f#=512285,pa=7d11d000
May  6 11:24:45 korpe-desktop kernel: [770766.884955] (735,958)p#=753598,va=b7fbe000 f#=512551,pa=7d227000
May  6 11:24:45 korpe-desktop kernel: [770766.884957] (735,959)p#=753599,va=b7fbf000 f#=507887,pa=7bfef000
May  6 11:24:45 korpe-desktop kernel: [770766.884960] (735,960)p#=753600,va=b7fc0000 f#=512613,pa=7d265000
May  6 11:24:45 korpe-desktop kernel: [770766.884962] (735,961)p#=753601,va=b7fc1000 f#=512011,pa=7d00b000
May  6 11:24:45 korpe-desktop kernel: [770766.884964] (735,962)p#=753602,va=b7fc2000 f#=512743,pa=7d2e7000
May  6 11:24:45 korpe-desktop kernel: [770766.884966] (735,963)p#=753603,va=b7fc3000 f#=512755,pa=7d2f3000
May  6 11:24:45 korpe-desktop kernel: [770766.884969] (735,964)p#=753604,va=b7fc4000 f#=512008,pa=7d008000
May  6 11:24:45 korpe-desktop kernel: [770766.884971] (735,965)p#=753605,va=b7fc5000 f#=512517,pa=7d205000
May  6 11:24:45 korpe-desktop kernel: [770766.884973] (735,966)p#=753606,va=b7fc6000 f#=512020,pa=7d014000
May  6 11:24:45 korpe-desktop kernel: [770766.884975] (735,967)p#=753607,va=b7fc7000 f#=512029,pa=7d01d000
May  6 11:24:45 korpe-desktop kernel: [770766.884977] (735,968)p#=753608,va=b7fc8000 f#=512077,pa=7d04d000
May  6 11:24:45 korpe-desktop kernel: [770766.884980] (735,971)p#=753611,va=b7fcb000 f#=507799,pa=7bf97000
May  6 11:24:45 korpe-desktop kernel: [770766.884982] (735,972)p#=753612,va=b7fcc000 f#=507829,pa=7bfb5000
May  6 11:24:45 korpe-desktop kernel: [770766.884984] (735,973)p#=753613,va=b7fcd000 f#=512594,pa=7d252000
May  6 11:24:45 korpe-desktop kernel: [770766.884986] (735,974)p#=753614,va=b7fce000 f#=512290,pa=7d122000
May  6 11:24:45 korpe-desktop kernel: [770766.884989] (735,976)p#=753616,va=b7fd0000 f#=512620,pa=7d26c000
May  6 11:24:45 korpe-desktop kernel: [770766.884991] (735,977)p#=753617,va=b7fd1000 f#=507534,pa=7be8e000
May  6 11:24:45 korpe-desktop kernel: [770766.884993] (735,978)p#=753618,va=b7fd2000 f#=224270,pa=36c0e000
May  6 11:24:45 korpe-desktop kernel: [770766.884995] (735,979)p#=753619,va=b7fd3000 f#=378563,pa=5c6c3000
May  6 11:24:45 korpe-desktop kernel: [770766.884998] (735,980)p#=753620,va=b7fd4000 f#=361640,pa=584a8000
May  6 11:24:45 korpe-desktop kernel: [770766.885000] (766,209)p#=784593,va=bf8d1000 f#=348108,pa=54fcc000
May  6 11:24:45 korpe-desktop kernel: [770766.885002] (766,211)p#=784595,va=bf8d3000 f#=395226,pa=607da000
May  6 11:24:45 korpe-desktop kernel: [770766.885004] number of page frames allocated = 107
May  6 11:24:46 korpe-desktop kernel: [770767.902282] module ending:  page table dump
```

Chapter 9

File Systems

9.1 File Systems

9.1.1 Accessing Files and Directories

In Linux, we can access a file using the related system calls. We can use open() system call to open a file. We can use the read and write system calls to read from a file and write into a file. We can use the lseek system call to jump to an arbitrary (random) location in the file. We can then start reading from there or write into there. We can also read the attributes of a file and set some of the atributes.

The following program (`fileop.c`)is an example that uses some low level file operations.

```
1  /* -*- linux-c  -*-
2
3     $Id: fileop.c,v 1.3 2009/04/16 23:06:38 korpe Exp korpe $
4
5     This programs shows how to obtain the attributes of a file, how to
6     access a file using raw I/O functions such as read and write, and
7     how to perform random access to a file using the lseek
8     function. The functions read, write, and lseek have corresponding
9     system calls. If you want to get more information about these
10    functions (systems calls) you should read the related man pages
11    using the command "man -S 2 <functionname>".
12
13    The program creates a binary file and populates the file with
14    records. You can think the file as a database table. Then the
15    program makes some accesses to the file.
16 */
17
18 #include <stdio.h>
19 #include <stdlib.h>
```

```
20  #include <sys/types.h>
21  #include <sys/stat.h>
22  #include <fcntl.h>
23  #include <unistd.h>
24  #include <string.h>
25  #include <sys/mman.h>
26
27  #define STUCOUNT 1000
28  #define AFILENAME "x.dat"
29  #define RANDOMACCESS_COUNT 10
30
31  struct student {
32          int id;
33          char name[20];
34          int class;
35          float cgpa;
36  };
37
38  int
39  main(int argc, char **argv)
40  {
41          int fd;
42          struct stat finfo;
43          char filename[100];
44          int i;
45          struct student s;
46          int n;
47          int offset;
48          int x;
49          struct student *stutable;
50          struct student *sptr;
51
52          strcpy(filename, "x.dat");
53
54          unlink(filename);        /* remove file if exists */
55
56          /* first create a file and then open it. */
57          fd = open(filename, O_CREAT | O_RDWR, 0660);
58
59          if (fd == -1) {
60                  printf("open failed \n");
61                  exit(1);
62          } else
63                  printf("open success, fd = %d \n", fd);
64
65          /* get the file attributes */
66          if (fstat(fd, &finfo) == -1) {
67                  printf("fstat failed \n");
68                  exit(1);
69          }
70          printf("information about file = %s\n", filename);
71          printf("  filesize               = %d bytes\n", (int) finfo.st_size);
72          printf("  preferred block size   = %d bytes\n", (int) finfo.st_blksize);
73          printf("  last access time       = %d seconds\n", (int) finfo.st_atime);
74          printf("  last modification time = %d seconds\n", (int) finfo.st_mtime);
75          printf("  creation time          = %d seconds\n", (int) finfo.st_ctime);
```

```
76
77          srand(1000);
78
79          for (i = 0; i < STUCOUNT; ++i) {
80                  s.id = i;
81                  sprintf(s.name, "student%d", i);
82                  s.class = (random() % 4) + 1;
83                  s.cgpa = ((float) random() / (float) RAND_MAX) * 3 + 1;
84
85                  n = write(fd, (void *) &s, sizeof (struct student));
86                  if (n != sizeof (struct student)) {
87                          printf("write() failed\n");
88                          exit(1);
89                  }
90          }
91          printf("student table populated\n");
92
93          printf("\n\n...now doing sequential read \n");
94          /* do a sequential read */
95          /* rewind the file pointer to the start of the file */
96          offset = 0;
97          lseek(fd, offset, SEEK_SET);
98
99          while (1) {
100                 n = read(fd, (void *) &s, sizeof (struct student));
101                 if (n == 0)
102                         break;
103                 else if (n != sizeof (struct student)) {
104                         printf("read failed\n");
105                         exit(1);
106                 }
107
108                 printf("student id=%d, name=%s, class=%d, cgpa=%.2f\n",
109                        s.id, s.name, s.class, s.cgpa);
110         }
111
112         printf("\n\n...now doing random access\n");
113
114         /* do some random access */
115         for (i = 0; i < RANDOMACCESS_COUNT; ++i) {
116                 x = random() % STUCOUNT;
117                 printf("will retrieve the record of %d'th student\n", x);
118                 /* retrieve the record of x th student */
119                 offset = x * sizeof (struct student);
120                 lseek(fd, offset, SEEK_SET);
121                 n = read(fd, (void *) &s, sizeof (struct student));
122                 if (n != sizeof (struct student)) {
123                         printf("read failed\n");
124                         exit(1);
125                 }
126
127                 printf("student id=%d, name=%s, class=%d, cgpa=%.2f\n",
128                        s.id, s.name, s.class, s.cgpa);
129         }
130
131         /*
```

```
132             now we will map the file into a region of the virtual
133             memory address space of the process and then access the
134             file as if we were accesing a memory area. For this we will
135             use the "mmap" system call. You can obtain more information
136             about the mmap system call from its man page.
137          */
138
139         offset = 0;
140         stutable = (struct student *)
141             mmap(0, STUCOUNT * sizeof (struct student),
142                  PROT_READ | PROT_WRITE, MAP_SHARED, fd, offset);
143
144         if ((int) stutable == -1) {
145                 printf("mmap failed \n");
146                 exit(1);
147         } else {
148                 printf
149                     ("\n\n...mapped the file into virtual memory with success \n");
150         }
151
152         /* perform some random accesses */
153         printf("\n\n...doing random access on the memory mapped file\n");
154         for (i = 0; i < RANDOMACCESS_COUNT; ++i) {
155                 x = random() % STUCOUNT;
156                 printf("will retrieve the record of %d'th student\n", x);
157                 /* retrieve the record of x th student */
158                 sptr = &stutable[x];
159                 printf("student id=%d, name=%s, class=%d, cgpa=%.2f\n",
160                        sptr->id, sptr->name, sptr->class, sptr->cgpa);
161         }
162
163         if (munmap(stutable, STUCOUNT * sizeof (struct student)) == -1) {
164                 printf("munmap failed \n");
165                 exit(1);
166         }
167
168         if (close(fd) == -1) {
169                 printf("close failed \n");
170                 exit(1);
171         }
172
173         return (0);
174 }
```

We also include a Makefile to compile the program.

```
1  all: fileop
2
3  fileop: fileop.c
4          gcc -Wall -o fileop fileop.c
5
6
7  clean:
8          rm -fr *~ fileop
```

9.1.2 Linux File System

The file system commonly used by Linux is Ext3 File system (extended file system 3). We now have Ext4. We will exercise with Ext3 here.

For example, we have that file system installed in a partition of the local harddisk of our computer. That partition is referred by Linux with a device name like /dev/sda5.

We can get information about the ext3 file system installed on that partition by using the `/sbin/dumpe2fs` command. That will read the superblock of the partition (volume) and extract the summary and basic information about the file system. To do that, we execute the following command:

```
/sbin/dumpe2fs -h /dev/sda5
```

The output that we will get will look like the following:

```
Filesystem volume name:
Last mounted on:
Filesystem UUID:          2cedc3e0-d683-4374-b67c-b5ef0c4de783
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index
filetype needs_recovery sparse_super large_file
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              3662848
Block count:              7323624
Reserved block count:     366181
Free blocks:              4903592
Free inodes:              3288736
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      1024
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         16352
Inode blocks per group:   511
Filesystem created:       Thu Aug  2 15:30:13 2007
Last mount time:          Sun Mar  8 23:56:33 2009
Last write time:          Sun Mar  8 23:56:33 2009
Mount count:              4
Maximum mount count:      500
Last checked:             Mon Mar  2 19:42:39 2009
Check interval:           5184000 (2 months)
Next check after:         Fri May  1 20:42:39 2009
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
```

```
35 First inode:              11
36 Inode size:                           128
37 Journal inode:            8
38 First orphan inode:       3487551
39 Default directory hash:   tea
40 Directory Hash Seed:      9f1b22ee-072e-4699-8339-b32957260abe
41 Journal backup:           inode blocks
42 Journal size:             128M
```

The Linux ext3 file system is implemented in the subdirectory `fs/ext3/` of the kernel source. The related data structures are defined in file `include/linux/ext3_fs.h` in the kernel source tree. Similarly, the ext4 file system is implemented in the `fs/ext4` of the kernel source tree.

9.1.3 Making a New Filesystem

In Linux, we can create a new file system on a block device (like a hard-disk) or a partition of it. This is called high level formatting. The name of the command to do this is `mke2fs` (/sbin/mke2fs). It can be used to create an ext2 or ext3 filesystem. The command takes the special file corresponding to the device as an argument. For example, if we want to create an ext2 file system on a floppy disk, then we should give the corresponding spefial file as an argument. The corresponding special file is usually called as `/dev/fd0`. In Linux, each device or partition is referred with a special device filename. For example, a SCCI hard disk can be referred as `/dev/sda`. A partition of it can be referred as `/dev/sda5`.

We can also create a pseudo block-oriented device (not a physical one) that is associated with a file. First we create a file that will act as a file store (i.e. a pseudo disk). We can do it as follows:

```
sudo dd if=/dev/zero of=filedisk.img bs=4K count=100000
```

This will create a file in the current directory called `filedisk.img`. That file will act as a virtual disk. The virtual disk will have 100000 blocks and block size is 4KB.

Now we can make that file a block oriented file (virtual disk) that can be accessed via a block special device file. We can do this by the following command:

```
sudo /sbin/losetup /dev/loop0 filedisk.img
```

Now we have a block-oriented pseudo storage device filedisk.img that has the corresponding device filename /dev/loop0. That is a block oriented device. We can now create a file system on this. For that we type:

```
sudo /sbin/mke2fs -b4096 /dev/loop0 100000
```

This will create an ext2 file system on /dev/loop0, that has 100000 blocks. The block size is 4096 bytes. We can now check that it is realy formatted by using the dumpe2fs command:

```
/sbin/dumpe2fs /dev/loop0
```

We can get such an output:

```
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          c092c3fc-3e67-416d-b20e-4c3957131d2f
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      resize_inode dir_index filetype sparse_super
large_file
Default mount options:    (none)
Filesystem state:         not clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              100096
Block count:              100000
Reserved block count:     5000
Free blocks:              96780
Free inodes:              100085
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      24
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         25024
Inode blocks per group:   782
Filesystem created:       Sat Apr 18 23:36:29 2009
Last mount time:          Sat Apr 18 23:47:36 2009
Last write time:          Sat Apr 18 23:47:54 2009
Mount count:              1
Maximum mount count:      31
Last checked:             Sat Apr 18 23:36:29 2009
Check interval:           15552000 (6 months)
Next check after:         Thu Oct 15 23:36:29 2009
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Default directory hash:   tea
Directory Hash Seed:      0bb71bbd-0e88-4751-8f0e-259339a9e455

Group 0: (Blocks 0-32767)
  Primary superblock at 0, Group descriptors at 1-1
  Reserved GDT blocks at 2-25
  Block bitmap at 26 (+26), Inode bitmap at 27 (+27)
  Inode table at 28-809 (+28)
  31952 free blocks, 25012 free inodes, 2 directories
  Free blocks: 816-32767
```

```
  Free inodes: 13-25024
Group 1: (Blocks 32768-65535)
  Backup superblock at 32768, Group descriptors at 32769-32769
  Reserved GDT blocks at 32770-32793
  Block bitmap at 32794 (+26), Inode bitmap at 32795 (+27)
  Inode table at 32796-33577 (+28)
  31958 free blocks, 25024 free inodes, 0 directories
  Free blocks: 33578-65535
  Free inodes: 25025-50048
Group 2: (Blocks 65536-98303)
  Block bitmap at 65536 (+0), Inode bitmap at 65537 (+1)
  Inode table at 65538-66319 (+2)
  31984 free blocks, 25024 free inodes, 0 directories
  Free blocks: 66320-98303
  Free inodes: 50049-75072
Group 3: (Blocks 98304-99999)
  Backup superblock at 98304, Group descriptors at 98305-98305
  Reserved GDT blocks at 98306-98329
  Block bitmap at 98330 (+26), Inode bitmap at 98331 (+27)
  Inode table at 98332-99113 (+28)
  886 free blocks, 25024 free inodes, 0 directories
  Free blocks: 99114-99999
  Free inodes: 75073-100096
```

We can mount this new file system into our local directory tree. Lets create a mount point in /mnt directory and then mount the new file system there:

```
sudo mkdir /mnt/newfs
sudo mount /dev/loop0 /mnt/newfs
```

We can now change into that directory and do whatever we would like to do: list directory content, create a new file etc. The new file will be created in that new file system that is sitting on virtual disk filedisk.img accessed via /dev/loop0.

```
cd /mnt/newfs/
<tt>touch x.txt
```

We have a file x.txt created in the root directory of the virtual disk.

9.1.4 Getting Info about files and inodes

The following two programs are very helpful in getting information about a file, its inode, and its physical disk blocks: `stat`, and `/sbin/debugfs`. For example, if we say `stat out` for a file out in the current directory, the following information will be printed out:

```
  File: ‘out’
  Size: 4096            Blocks: 16         IO Block: 4096   regular
  file
Device: 805h/2053d      Inode: 524136      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   korpe)   Gid: (  100/
  users)
Access: 2009-04-19 22:08:34.000000000 +0300
Modify: 2009-04-19 22:07:08.000000000 +0300
Change: 2009-04-19 22:07:08.000000000 +0300
```

The index (number) of the inode allocated to the file on disk is: 524136. We can use the debugfs program to find out the block numbers that store the content of the file. For that we can type: `sudo /sbin/debugfs /dev/sda5` (/dev/sda5 is the hard disk partition where the ext3 file system is installed and the file out is sitting in). We will get a command prompt. At the prompt we can type: `bmap ...pathname.../out 0`. That means we would like to get the physical block number where the relative block 0 of the file out is sitting on. The output we will get will be: `1050629`. It is quite a short file, so it can fit into a single block.

9.1.5 Raw Access to Disk and File System Data

We can access a disk partition directly by opening the corresponding device file. For example, in a program, we can open a device /dev/sda5, corresponding to an ext3 partion. Our linux, for example, installed in that partition. Hence it is a bootable partition. It contans ext3 file system.

We can see the filesystems that are currently mounted and accessible via the mount command. Below is sample output. There we can see that we have a partition that is referred with a device file /dev/sda5 and that has ext3 file system on it.

```
korpe@pckorpe:~/book/filesystems$ mount
/dev/sda5 on / type ext3 (rw,acl,user_xattr)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
debugfs on /sys/kernel/debug type debugfs (rw)
udev on /dev type tmpfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
securityfs on /sys/kernel/security type securityfs (rw)
/dev/loop0 on /mnt/f type ext2 (rw)
```

We now present a user level program (that has to be executed with superuser privilage) that opens and accessed a hard disk partition in raw mode, without going trough the file system. The program can access any block of the partition directly. It can obtain file system metadata of the file system sitting in that partition.

Note that this program opens the partition in READ-ONLY mode. Do not open the partition for writing and do not write something to the partition in raw mode, unless you are very careful and you know what you are doing. You can corrupt your file system in an unrecoverable way.

Below is the program (`ext3parse.c`).

```
1  /* -*- linux-c -*- */
2  /* $Id: ext3parse.c,v 1.6 2009/05/15 07:07:34 korpe Exp korpe $ */
3
4
5  #define _FILE_OFFSET_BITS 64
6  #define _LARGEFILE64_SOURCE
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <sys/types.h>
12 #include <sys/stat.h>
13 #include <fcntl.h>
14 #include <string.h>
15 #include <linux/unistd.h>
16 #include <errno.h>
17 #include "ext3_fs.h"
18
19
20 #define DEBUG 1
21 #define PRINT_INODETABLE 0
22
23 #define BLOCKSIZE  4096
24 #define MAX_DIRNAME 256
25
26 #define SUPER_BLOCK       0
27 #define GROUP0_GDT_BLOCK1 1
28
29
30 void
31 get_block (int fd,  unsigned long bnum, unsigned char *block)
32 {
33         int num_read;
34         off64_t offset;
35         off64_t pos;
36
37
38         if (DEBUG)
39                 printf ("trying to retrieve block %lu\n", bnum);
40
41         offset = bnum;
42         offset = offset * BLOCKSIZE;
43
44         pos = lseek64 (fd, offset, SEEK_SET);
45         if (pos != offset) {
46                 printf ("lseek64 failed\n");
47                 exit (1);
48         }
```

```
49
50          num_read = read (fd, block, BLOCKSIZE);
51          if (num_read  !=  BLOCKSIZE) {
52                  printf ("can not read block\n");
53                  exit (1);
54          }
55
56          if (DEBUG)
57                  printf ("retrieved block %lu\n", bnum);
58  }
59
60
61  void
62  print_group_table (unsigned char *buffer, unsigned int buflen)
63  {
64          struct ext3_group_desc *g;
65          int i, gnum;
66
67          g = (struct ext3_group_desc *) buffer;
68
69          i = 0;
70          gnum = 0;
71          printf ("size of group desc structure  = %u\n", sizeof (struct ext3_group_desc));
72
73          while (1) {
74                  printf ("group %d: ", gnum);
75                  printf ("bitmap_block#=%u ", g->bg_block_bitmap);
76                  printf ("inodetable_block#=%u ", g->bg_inode_table);
77                  printf ("free_block_count=%u ",  g->bg_free_blocks_count);
78                  printf ("\n");
79
80
81                  i += sizeof (struct ext3_group_desc);
82                  if (i >= buflen)
83                          break;
84
85                  g++;
86                  gnum++;
87          }
88  }
89
90
91
92  void
93  print_inode_bitmap (int fd, struct ext3_group_desc *g, struct ext3_super_block *s)
94  {
95          unsigned char buffer[BLOCKSIZE];
96          int i, count;
97
98
99          get_block (fd,  g->bg_inode_bitmap, buffer);
100
101         count = s->s_inodes_per_group / 8;
102
103         for (i = 0; i < count; ++i) {
104                 printf ("%02x", buffer[i]);
```

```
105
106                 if ( ( (i+1) % 32) == 0)
107                         printf ("\n");
108         }
109         printf ("\n");
110 }
111
112
113 void
114 print_inodetable_in_group (int fd, struct ext3_group_desc *g, struct ext3_super_block *s)
115 {
116         int k, i;
117         unsigned char buffer[BLOCKSIZE];
118         struct ext3_inode *inode_p;
119         int i_block;       /* number of inodes per block */
120         int inode_blocks;  /* inode blocks per group */
121
122
123         i_block = BLOCKSIZE / s->s_inode_size;
124         inode_blocks = s->s_inodes_per_group / i_block;
125
126
127         for (k = 0; k < inode_blocks; ++k) {
128                 get_block (fd,  g->bg_inode_table + k, buffer);
129
130                 for (i = 0; i < i_block; ++i) {
131                         inode_p = (struct ext3_inode *) (buffer + (i * s->s_inode_size));
132
133                         printf("inode %u: ", 1 + k* i_block + i);
134
135                         if (inode_p->i_links_count > 0) {
136                                 printf("size=%u, ", inode_p->i_size);
137                                 printf("uid=%u, ", inode_p->i_uid);
138                                 printf("links_count=%u, ", inode_p->i_links_count);
139                                 printf("#blocks=%u, ", inode_p->i_blocks);
140                                 printf("first_3_datablocks=%u %u %u, ",
141                                        inode_p->i_block[0],
142                                        inode_p->i_block[1],
143                                        inode_p->i_block[2]);
144                         }
145
146
147                         printf("\n");
148
149                 }
150
151         }
152
153 }
154
155
156 void
157 print_dir (unsigned char *buffer, unsigned int buflen)
158 {
159         struct ext3_dir_entry_2  *dep;
160         int i;
```

```
161         char entry_name[MAX_DIRNAME];
162
163         i = 0;
164         while (1) {
165                 dep = (struct  ext3_dir_entry_2*) (buffer + i);
166
167                 strncpy (entry_name, dep->name, dep->name_len);
168                 entry_name[dep->name_len] = '\0';
169
170                 printf ("type=%d inode=%-10u name = %s\n",
171                         dep->file_type,
172                         dep->inode,
173                         entry_name);
174
175                 i += dep->rec_len;
176                 if (i >= BLOCKSIZE)
177                         break;
178         }
179 }
180
181 int
182 main (int argc, char **argv)
183 {
184         int  fd;
185         unsigned char buf[BLOCKSIZE];
186         struct ext3_super_block sb;
187         struct ext3_group_desc  group0;
188         struct ext3_inode       root_inode;
189         int n;
190
191
192         if (argc != 2) {
193                 printf ("usage: ext3parse <devicefile>\n");
194                 exit (1);
195         }
196
197         fd = open (argv[1], O_RDONLY);
198
199         if (fd < 0) {
200                 printf ("can not open device file\n");
201                 exit(1);
202         }
203
204         /*
205            access block 0, which contains superblock at offset 1024
206         */
207         get_block (fd, SUPER_BLOCK, buf);
208         memcpy ( (void*)&sb, (void*)(buf + 1024), sizeof (struct ext3_super_block));
209
210         printf ("\nsuperblock information:\n");
211         printf ("size_of_super_block_structure=%u\n", sizeof (struct ext3_super_block ));
212         printf ("inode_count=%u\n", sb.s_inodes_count);
213         printf ("block_count=%u\n", sb.s_blocks_count);
214         printf ("first_data_block=%u\n", sb.s_first_data_block);
215         printf ("magic_number=%x\n", (unsigned short) sb.s_magic);
216         printf ("inode_size=%d\n", sb.s_inode_size);
```

```
217         printf ("inodes_per_group=%d\n", sb.s_inodes_per_group);
218
219
220         /*
221            obtain group descriptor table (GDT) block and print info
222            about groups; we are just accessing the first block; there
223            may be more than one block containing GDT.
224         */
225         get_block (fd, GROUP0_GDT_BLOCK1, buf);
226         memcpy ( (void*) &group0, (void *) buf, sizeof (struct ext3_group_desc));
227         printf ("\ngroup descriptor table content in block 1\n");
228         print_group_table (buf, BLOCKSIZE);
229
230         /*
231            get the root directory inode; it is inode EXT3_ROOT_INO.
232            EXT3_ROOT_INO is defined in ext3_fs.h.
233            inode 2 is in group 0.
234         */
235         get_block (fd,  group0.bg_inode_table, buf);
236         if (sb.s_inode_size  <= sizeof (struct ext3_inode))
237                 n = sb.s_inode_size;
238         else
239                 n = sizeof (struct ext3_inode);
240
241         memcpy ( (void*) &root_inode,
242                  (void*) (buf + ( (EXT3_ROOT_INO-1) * sb.s_inode_size)),
243                  n);
244
245         if (PRINT_INODETABLE) {
246                 /*
247                    print inode table of group 0
248                  */
249                 printf ("\ninode table of group 0\n");
250                 print_inodetable_in_group (fd, &group0, &sb);
251         }
252
253         printf ("\ninode bitmap of the group\n");
254         print_inode_bitmap (fd, &group0, &sb);
255
256         /*
257            go to root directory and print it
258         */
259         get_block (fd, root_inode.i_block[0], buf);
260         printf ("\nroot directory content\n");
261         print_dir (buf, BLOCKSIZE);
262
263         close (fd);
264
265         return 0;
266 }
```

The program includes a header file whose content is shown below.

```
1 /*
```

```
 *  linux/include/linux/ext3_fs.h
 *
 * Copyright (C) 1992, 1993, 1994, 1995
 * Remy Card (card@masi.ibp.fr)
 * Laboratoire MASI - Institut Blaise Pascal
 * Universite Pierre et Marie Curie (Paris VI)
 *
 *  from
 *
 *  linux/include/linux/minix_fs.h
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

#ifndef _LINUX_EXT3_FS_H
#define _LINUX_EXT3_FS_H

#include <linux/types.h>

/*
 * The second extended filesystem constants/structures
 */

/*
 * Define EXT3FS_DEBUG to produce debug messages
 */
#undef EXT3FS_DEBUG

/*
 * Define EXT3_RESERVATION to reserve data blocks for expanding files
 */
#define EXT3_DEFAULT_RESERVE_BLOCKS     8
/*max window size: 1024(direct blocks) + 3([t,d]indirect blocks) */
#define EXT3_MAX_RESERVE_BLOCKS         1027
#define EXT3_RESERVE_WINDOW_NOT_ALLOCATED 0
/*
 * Always enable hashed directories
 */
#define CONFIG_EXT3_INDEX

/*
 * Debug code
 */
#ifdef EXT3FS_DEBUG
#define ext3_debug(f, a...)                                                     \
        do {                                                                     \
                printk (KERN_DEBUG "EXT3-fs DEBUG (%s, %d): %s:",        \
                        __FILE__, __LINE__, __FUNCTION__);                       \
                printk (KERN_DEBUG f, ## a);                                      \
        } while (0)
#else
#define ext3_debug(f, a...)        do {} while (0)
#endif

/*
 * Special inodes numbers
```

```
58  */
59  #define        EXT3_BAD_INO                 1        /* Bad blocks inode */
60  #define EXT3_ROOT_INO                2        /* Root inode */
61  #define EXT3_BOOT_LOADER_INO         5        /* Boot loader inode */
62  #define EXT3_UNDEL_DIR_INO         6        /* Undelete directory inode */
63  #define EXT3_RESIZE_INO                7        /* Reserved group descriptors inode */
64  #define EXT3_JOURNAL_INO         8        /* Journal inode */
65
66  /* First non-reserved inode for old ext3 filesystems */
67  #define EXT3_GOOD_OLD_FIRST_INO        11
68
69  /*
70   * The second extended file system magic number
71   */
72  #define EXT3_SUPER_MAGIC        0xEF53
73
74  /*
75   * Maximal count of links to a file
76   */
77  #define EXT3_LINK_MAX                32000
78
79  /*
80   * Macro-instructions used to manage several block sizes
81   */
82  #define EXT3_MIN_BLOCK_SIZE                1024
83  #define        EXT3_MAX_BLOCK_SIZE                4096
84  #define EXT3_MIN_BLOCK_LOG_SIZE                10
85  #ifdef __KERNEL__
86  # define EXT3_BLOCK_SIZE(s)                ((s)->s_blocksize)
87  #else
88  # define EXT3_BLOCK_SIZE(s)                (EXT3_MIN_BLOCK_SIZE << (s)->s_log_block_size)
89  #endif
90  #define        EXT3_ADDR_PER_BLOCK(s)                (EXT3_BLOCK_SIZE(s) / sizeof (__u32))
91  #ifdef __KERNEL__
92  # define EXT3_BLOCK_SIZE_BITS(s)        ((s)->s_blocksize_bits)
93  #else
94  # define EXT3_BLOCK_SIZE_BITS(s)        ((s)->s_log_block_size + 10)
95  #endif
96  #ifdef __KERNEL__
97  #define        EXT3_ADDR_PER_BLOCK_BITS(s)        (EXT3_SB(s)->s_addr_per_block_bits)
98  #define EXT3_INODE_SIZE(s)                (EXT3_SB(s)->s_inode_size)
99  #define EXT3_FIRST_INO(s)                (EXT3_SB(s)->s_first_ino)
100 #else
101 #define EXT3_INODE_SIZE(s)        (((s)->s_rev_level == EXT3_GOOD_OLD_REV) ? \
102                                   EXT3_GOOD_OLD_INODE_SIZE : \
103                                   (s)->s_inode_size)
104 #define EXT3_FIRST_INO(s)        (((s)->s_rev_level == EXT3_GOOD_OLD_REV) ? \
105                                   EXT3_GOOD_OLD_FIRST_INO : \
106                                   (s)->s_first_ino)
107 #endif
108
109 /*
110  * Macro-instructions used to manage fragments
111  */
112 #define EXT3_MIN_FRAG_SIZE                1024
113 #define        EXT3_MAX_FRAG_SIZE                4096
```

```
114 #define EXT3_MIN_FRAG_LOG_SIZE                  10
115 #ifdef __KERNEL__
116 # define EXT3_FRAG_SIZE(s)                (EXT3_SB(s)->s_frag_size)
117 # define EXT3_FRAGS_PER_BLOCK(s)        (EXT3_SB(s)->s_frags_per_block)
118 #else
119 # define EXT3_FRAG_SIZE(s)                (EXT3_MIN_FRAG_SIZE << (s)->s_log_frag_size)
120 # define EXT3_FRAGS_PER_BLOCK(s)        (EXT3_BLOCK_SIZE(s) / EXT3_FRAG_SIZE(s))
121 #endif
122
123 /*
124  * Structure of a blocks group descriptor
125  */
126 struct ext3_group_desc
127 {
128         __le32        bg_block_bitmap;               /* Blocks bitmap block */
129         __le32        bg_inode_bitmap;               /* Inodes bitmap block */
130         __le32        bg_inode_table;               /* Inodes table block */
131         __le16        bg_free_blocks_count;        /* Free blocks count */
132         __le16        bg_free_inodes_count;        /* Free inodes count */
133         __le16        bg_used_dirs_count;        /* Directories count */
134         __u16        bg_pad;
135         __le32        bg_reserved[3];
136 };
137
138 /*
139  * Macro-instructions used to manage group descriptors
140  */
141 #ifdef __KERNEL__
142 # define EXT3_BLOCKS_PER_GROUP(s)        (EXT3_SB(s)->s_blocks_per_group)
143 # define EXT3_DESC_PER_BLOCK(s)                (EXT3_SB(s)->s_desc_per_block)
144 # define EXT3_INODES_PER_GROUP(s)        (EXT3_SB(s)->s_inodes_per_group)
145 # define EXT3_DESC_PER_BLOCK_BITS(s)        (EXT3_SB(s)->s_desc_per_block_bits)
146 #else
147 # define EXT3_BLOCKS_PER_GROUP(s)        ((s)->s_blocks_per_group)
148 # define EXT3_DESC_PER_BLOCK(s)                (EXT3_BLOCK_SIZE(s) / sizeof (struct ext3_group_desc))
149 # define EXT3_INODES_PER_GROUP(s)        ((s)->s_inodes_per_group)
150 #endif
151
152 /*
153  * Constants relative to the data blocks
154  */
155 #define        EXT3_NDIR_BLOCKS                12
156 #define        EXT3_IND_BLOCK                        EXT3_NDIR_BLOCKS
157 #define        EXT3_DIND_BLOCK                        (EXT3_IND_BLOCK + 1)
158 #define        EXT3_TIND_BLOCK                        (EXT3_DIND_BLOCK + 1)
159 #define        EXT3_N_BLOCKS                        (EXT3_TIND_BLOCK + 1)
160
161 /*
162  * Inode flags
163  */
164 #define        EXT3_SECRM_FL                        0x00000001 /* Secure deletion */
165 #define        EXT3_UNRM_FL                        0x00000002 /* Undelete */
166 #define        EXT3_COMPR_FL                        0x00000004 /* Compress file */
167 #define EXT3_SYNC_FL                        0x00000008 /* Synchronous updates */
168 #define EXT3_IMMUTABLE_FL                0x00000010 /* Immutable file */
169 #define EXT3_APPEND_FL                        0x00000020 /* writes to file may only append */
```

```
170 #define EXT3_NODUMP_FL                          0x00000040 /* do not dump file */
171 #define EXT3_NOATIME_FL                         0x00000080 /* do not update atime */
172 /* Reserved for compression usage... */
173 #define EXT3_DIRTY_FL                       0x00000100
174 #define EXT3_COMPRBLK_FL                0x00000200 /* One or more compressed clusters */
175 #define EXT3_NOCOMPR_FL                         0x00000400 /* Don't compress */
176 #define EXT3_ECOMPR_FL                      0x00000800 /* Compression error */
177 /* End compression flags --- maybe not all used */
178 #define EXT3_INDEX_FL                           0x00001000 /* hash-indexed directory */
179 #define EXT3_IMAGIC_FL                          0x00002000 /* AFS directory */
180 #define EXT3_JOURNAL_DATA_FL                0x00004000 /* file data should be journaled */
181 #define EXT3_NOTAIL_FL                          0x00008000 /* file tail should not be merged */
182 #define EXT3_DIRSYNC_FL                         0x00010000 /* dirsync behaviour (directories only) */
183 #define EXT3_TOPDIR_FL                          0x00020000 /* Top of directory hierarchies*/
184 #define EXT3_RESERVED_FL                0x80000000 /* reserved for ext3 lib */
185
186 #define EXT3_FL_USER_VISIBLE                0x0003DFFF /* User visible flags */
187 #define EXT3_FL_USER_MODIFIABLE                 0x000380FF /* User modifiable flags */
188
189 /*
190  * Inode dynamic state flags
191  */
192 #define EXT3_STATE_JDATA                0x00000001 /* journaled data exists */
193 #define EXT3_STATE_NEW                          0x00000002 /* inode is newly created */
194 #define EXT3_STATE_XATTR                0x00000004 /* has in-inode xattrs */
195
196 /* Used to pass group descriptor data when online resize is done */
197 struct ext3_new_group_input {
198         __u32 group;            /* Group number for this data */
199         __u32 block_bitmap;     /* Absolute block number of block bitmap */
200         __u32 inode_bitmap;     /* Absolute block number of inode bitmap */
201         __u32 inode_table;      /* Absolute block number of inode table start */
202         __u32 blocks_count;     /* Total number of blocks in this group */
203         __u16 reserved_blocks;  /* Number of reserved blocks in this group */
204         __u16 unused;
205 };
206
207 /* The struct ext3_new_group_input in kernel space, with free_blocks_count */
208 struct ext3_new_group_data {
209         __u32 group;
210         __u32 block_bitmap;
211         __u32 inode_bitmap;
212         __u32 inode_table;
213         __u32 blocks_count;
214         __u16 reserved_blocks;
215         __u16 unused;
216         __u32 free_blocks_count;
217 };
218
219
220 /*
221  * ioctl commands
222  */
223 #define         EXT3_IOC_GETFLAGS               _IOR('f', 1, long)
224 #define         EXT3_IOC_SETFLAGS               _IOW('f', 2, long)
225 #define         EXT3_IOC_GETVERSION               _IOR('f', 3, long)
```

```
226 #define        EXT3_IOC_SETVERSION                _IOW('f', 4, long)
227 #define EXT3_IOC_GROUP_EXTEND              _IOW('f', 7, unsigned long)
228 #define EXT3_IOC_GROUP_ADD               _IOW('f', 8,struct ext3_new_group_input)
229 #define        EXT3_IOC_GETVERSION_OLD               _IOR('v', 1, long)
230 #define        EXT3_IOC_SETVERSION_OLD               _IOW('v', 2, long)
231 #ifdef CONFIG_JBD_DEBUG
232 #define EXT3_IOC_WAIT_FOR_READONLY       _IOR('f', 99, long)
233 #endif
234 #define EXT3_IOC_GETRSVSZ               _IOR('f', 5, long)
235 #define EXT3_IOC_SETRSVSZ               _IOW('f', 6, long)
236
237 /*
238  *  Mount options
239  */
240 struct ext3_mount_options {
241         unsigned long s_mount_opt;
242         uid_t s_resuid;
243         gid_t s_resgid;
244         unsigned long s_commit_interval;
245 #ifdef CONFIG_QUOTA
246         int s_jquota_fmt;
247         char *s_qf_names[MAXQUOTAS];
248 #endif
249 };
250
251 /*
252  * Structure of an inode on the disk
253  */
254 struct ext3_inode {
255         __le16        i_mode;                /* File mode */
256         __le16        i_uid;                /* Low 16 bits of Owner Uid */
257         __le32        i_size;                /* Size in bytes */
258         __le32        i_atime;        /* Access time */
259         __le32        i_ctime;        /* Creation time */
260         __le32        i_mtime;        /* Modification time */
261         __le32        i_dtime;        /* Deletion Time */
262         __le16        i_gid;                /* Low 16 bits of Group Id */
263         __le16        i_links_count;        /* Links count */
264         __le32        i_blocks;        /* Blocks count */
265         __le32        i_flags;        /* File flags */
266         union {
267                 struct {
268                         __u32  l_i_reserved1;
269                 } linux1;
270                 struct {
271                         __u32  h_i_translator;
272                 } hurd1;
273                 struct {
274                         __u32  m_i_reserved1;
275                 } masix1;
276         } osd1;                                /* OS dependent 1 */
277         __le32        i_block[EXT3_N_BLOCKS];/* Pointers to blocks */
278         __le32        i_generation;        /* File version (for NFS) */
279         __le32        i_file_acl;        /* File ACL */
280         __le32        i_dir_acl;        /* Directory ACL */
281         __le32        i_faddr;        /* Fragment address */
```

```
282         union {
283                 struct {
284                         __u8        l_i_frag;        /* Fragment number */
285                         __u8        l_i_fsize;        /* Fragment size */
286                         __u16        i_pad1;
287                         __le16        l_i_uid_high;        /* these 2 fields    */
288                         __le16        l_i_gid_high;        /* were reserved2[0] */
289                         __u32        l_i_reserved2;
290                 } linux2;
291                 struct {
292                         __u8        h_i_frag;        /* Fragment number */
293                         __u8        h_i_fsize;        /* Fragment size */
294                         __u16        h_i_mode_high;
295                         __u16        h_i_uid_high;
296                         __u16        h_i_gid_high;
297                         __u32        h_i_author;
298                 } hurd2;
299                 struct {
300                         __u8        m_i_frag;        /* Fragment number */
301                         __u8        m_i_fsize;        /* Fragment size */
302                         __u16        m_pad1;
303                         __u32        m_i_reserved2[2];
304                 } masix2;
305         } osd2;                                /* OS dependent 2 */
306         __le16        i_extra_isize;
307         __le16        i_pad1;
308 };
309
310 #define i_size_high        i_dir_acl
311
312 #if defined(__KERNEL__) || defined(__linux__)
313 #define i_reserved1        osd1.linux1.l_i_reserved1
314 #define i_frag                osd2.linux2.l_i_frag
315 #define i_fsize                osd2.linux2.l_i_fsize
316 #define i_uid_low        i_uid
317 #define i_gid_low        i_gid
318 #define i_uid_high        osd2.linux2.l_i_uid_high
319 #define i_gid_high        osd2.linux2.l_i_gid_high
320 #define i_reserved2        osd2.linux2.l_i_reserved2
321
322 #elif defined(__GNU__)
323
324 #define i_translator        osd1.hurd1.h_i_translator
325 #define i_frag                osd2.hurd2.h_i_frag;
326 #define i_fsize                osd2.hurd2.h_i_fsize;
327 #define i_uid_high        osd2.hurd2.h_i_uid_high
328 #define i_gid_high        osd2.hurd2.h_i_gid_high
329 #define i_author        osd2.hurd2.h_i_author
330
331 #elif defined(__masix__)
332
333 #define i_reserved1        osd1.masix1.m_i_reserved1
334 #define i_frag                osd2.masix2.m_i_frag
335 #define i_fsize                osd2.masix2.m_i_fsize
336 #define i_reserved2        osd2.masix2.m_i_reserved2
337
```

```
338 #endif /* defined(__KERNEL__) || defined(__linux__) */
339
340 /*
341  * File system states
342  */
343 #define        EXT3_VALID_FS                       0x0001        /* Unmounted cleanly */
344 #define        EXT3_ERROR_FS                       0x0002        /* Errors detected */
345 #define        EXT3_ORPHAN_FS                       0x0004       /* Orphans being recovered */
346
347 /*
348  * Mount flags
349  */
350 #define EXT3_MOUNT_CHECK                0x00001        /* Do mount-time checks */
351 #define EXT3_MOUNT_OLDALLOC               0x00002  /* Don't use the new Orlov allocator */
352 #define EXT3_MOUNT_GRPID                0x00004        /* Create files with directory's group */
353 #define EXT3_MOUNT_DEBUG                0x00008        /* Some debugging messages */
354 #define EXT3_MOUNT_ERRORS_CONT               0x00010        /* Continue on errors */
355 #define EXT3_MOUNT_ERRORS_RO              0x00020        /* Remount fs ro on errors */
356 #define EXT3_MOUNT_ERRORS_PANIC              0x00040        /* Panic on errors */
357 #define EXT3_MOUNT_MINIX_DF               0x00080        /* Mimics the Minix statfs */
358 #define EXT3_MOUNT_NOLOAD               0x00100        /* Don't use existing journal*/
359 #define EXT3_MOUNT_ABORT                0x00200        /* Fatal error detected */
360 #define EXT3_MOUNT_DATA_FLAGS                0x00C00        /* Mode for data writes: */
361 #define EXT3_MOUNT_JOURNAL_DATA                0x00400        /* Write data to journal */
362 #define EXT3_MOUNT_ORDERED_DATA                0x00800        /* Flush data before commit */
363 #define EXT3_MOUNT_WRITEBACK_DATA        0x00C00        /* No data ordering */
364 #define EXT3_MOUNT_UPDATE_JOURNAL        0x01000        /* Update the journal format */
365 #define EXT3_MOUNT_NO_UID32               0x02000  /* Disable 32-bit UIDs */
366 #define EXT3_MOUNT_XATTR_USER               0x04000        /* Extended user attributes */
367 #define EXT3_MOUNT_POSIX_ACL               0x08000        /* POSIX Access Control Lists */
368 #define EXT3_MOUNT_RESERVATION               0x10000        /* Preallocation */
369 #define EXT3_MOUNT_BARRIER               0x20000 /* Use block barriers */
370 #define EXT3_MOUNT_NOBH                     0x40000 /* No bufferheads */
371 #define EXT3_MOUNT_QUOTA                0x80000 /* Some quota option set */
372 #define EXT3_MOUNT_USRQUOTA               0x100000 /* "old" user quota */
373 #define EXT3_MOUNT_GRPQUOTA               0x200000 /* "old" group quota */
374
375 /* Compatibility, for having both ext2_fs.h and ext3_fs.h included at once */
376 #ifndef _LINUX_EXT2_FS_H
377 #define clear_opt(o, opt)               o &= ~EXT3_MOUNT_##opt
378 #define set_opt(o, opt)                     o |= EXT3_MOUNT_##opt
379 #define test_opt(sb, opt)               (EXT3_SB(sb)->s_mount_opt & \
380                                         EXT3_MOUNT_##opt)
381 #else
382 #define EXT2_MOUNT_NOLOAD               EXT3_MOUNT_NOLOAD
383 #define EXT2_MOUNT_ABORT                EXT3_MOUNT_ABORT
384 #define EXT2_MOUNT_DATA_FLAGS               EXT3_MOUNT_DATA_FLAGS
385 #endif
386
387 #define ext3_set_bit                       ext2_set_bit
388 #define ext3_set_bit_atomic                ext2_set_bit_atomic
389 #define ext3_clear_bit                       ext2_clear_bit
390 #define ext3_clear_bit_atomic               ext2_clear_bit_atomic
391 #define ext3_test_bit                       ext2_test_bit
392 #define ext3_find_first_zero_bit        ext2_find_first_zero_bit
393 #define ext3_find_next_zero_bit               ext2_find_next_zero_bit
```

```
394
395 /*
396  * Maximal mount counts between two filesystem checks
397  */
398 #define EXT3_DFL_MAX_MNT_COUNT                20        /* Allow 20 mounts */
399 #define EXT3_DFL_CHECKINTERVAL                0        /* Don't use interval check */
400
401 /*
402  * Behaviour when detecting errors
403  */
404 #define EXT3_ERRORS_CONTINUE                1        /* Continue execution */
405 #define EXT3_ERRORS_RO                        2        /* Remount fs read-only */
406 #define EXT3_ERRORS_PANIC                3        /* Panic */
407 #define EXT3_ERRORS_DEFAULT                EXT3_ERRORS_CONTINUE
408
409 /*
410  * Structure of the super block
411  */
412 struct ext3_super_block {
413 /*00*/        __le32        s_inodes_count;                /* Inodes count */
414         __le32        s_blocks_count;                /* Blocks count */
415         __le32        s_r_blocks_count;        /* Reserved blocks count */
416         __le32        s_free_blocks_count;        /* Free blocks count */
417 /*10*/        __le32        s_free_inodes_count;        /* Free inodes count */
418         __le32        s_first_data_block;        /* First Data Block */
419         __le32        s_log_block_size;        /* Block size */
420         __le32        s_log_frag_size;        /* Fragment size */
421 /*20*/        __le32        s_blocks_per_group;        /* # Blocks per group */
422         __le32        s_frags_per_group;        /* # Fragments per group */
423         __le32        s_inodes_per_group;        /* # Inodes per group */
424         __le32        s_mtime;                /* Mount time */
425 /*30*/        __le32        s_wtime;                /* Write time */
426         __le16        s_mnt_count;                /* Mount count */
427         __le16        s_max_mnt_count;        /* Maximal mount count */
428         __le16        s_magic;                /* Magic signature */
429         __le16        s_state;                /* File system state */
430         __le16        s_errors;                /* Behaviour when detecting errors */
431         __le16        s_minor_rev_level;        /* minor revision level */
432 /*40*/        __le32        s_lastcheck;                /* time of last check */
433         __le32        s_checkinterval;        /* max. time between checks */
434         __le32        s_creator_os;                /* OS */
435         __le32        s_rev_level;                /* Revision level */
436 /*50*/        __le16        s_def_resuid;                /* Default uid for reserved blocks */
437         __le16        s_def_resgid;                /* Default gid for reserved blocks */
438         /*
439          * These fields are for EXT3_DYNAMIC_REV superblocks only.
440          *
441          * Note: the difference between the compatible feature set and
442          * the incompatible feature set is that if there is a bit set
443          * in the incompatible feature set that the kernel doesn't
444          * know about, it should refuse to mount the filesystem.
445          *
446          * e2fsck's requirements are more strict; if it doesn't know
447          * about a feature in either the compatible or incompatible
448          * feature set, it must abort and not try to meddle with
449          * things it doesn't understand...
```

```
450         */
451         __le32        s_first_ino;               /* First non-reserved inode */
452         __le16   s_inode_size;               /* size of inode structure */
453         __le16        s_block_group_nr;        /* block group # of this superblock */
454         __le32        s_feature_compat;        /* compatible feature set */
455 /*60*/        __le32        s_feature_incompat;        /* incompatible feature set */
456         __le32        s_feature_ro_compat;       /* readonly-compatible feature set */
457 /*68*/        __u8        s_uuid[16];               /* 128-bit uuid for volume */
458 /*78*/        char        s_volume_name[16];        /* volume name */
459 /*88*/        char        s_last_mounted[64];        /* directory where last mounted */
460 /*C8*/        __le32        s_algorithm_usage_bitmap; /* For compression */
461         /*
462          * Performance hints.  Directory preallocation should only
463          * happen if the EXT3_FEATURE_COMPAT_DIR_PREALLOC flag is on.
464          */
465         __u8        s_prealloc_blocks;        /* Nr of blocks to try to preallocate*/
466         __u8        s_prealloc_dir_blocks;        /* Nr to preallocate for dirs */
467         __u16        s_reserved_gdt_blocks;        /* Per group desc for online growth */
468         /*
469          * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
470          */
471 /*D0*/        __u8        s_journal_uuid[16];        /* uuid of journal superblock */
472 /*E0*/        __le32        s_journal_inum;                /* inode number of journal file */
473         __le32        s_journal_dev;               /* device number of journal file */
474         __le32        s_last_orphan;               /* start of list of inodes to delete */
475         __le32        s_hash_seed[4];               /* HTREE hash seed */
476         __u8        s_def_hash_version;        /* Default hash version to use */
477         __u8        s_reserved_char_pad;
478         __u16        s_reserved_word_pad;
479         __le32        s_default_mount_opts;
480         __le32        s_first_meta_bg;         /* First metablock block group */
481         __u32        s_reserved[190];        /* Padding to the end of the block */
482 };
483
484 #ifdef __KERNEL__
485 #include <linux/ext3_fs_i.h>
486 #include <linux/ext3_fs_sb.h>
487 static inline struct ext3_sb_info * EXT3_SB(struct super_block *sb)
488 {
489         return sb->s_fs_info;
490 }
491 static inline struct ext3_inode_info *EXT3_I(struct inode *inode)
492 {
493         return container_of(inode, struct ext3_inode_info, vfs_inode);
494 }
495
496 static inline int ext3_valid_inum(struct super_block *sb, unsigned long ino)
497 {
498         return ino == EXT3_ROOT_INO ||
499                 ino == EXT3_JOURNAL_INO ||
500                 ino == EXT3_RESIZE_INO ||
501                 (ino >= EXT3_FIRST_INO(sb) &&
502                  ino <= le32_to_cpu(EXT3_SB(sb)->s_es->s_inodes_count));
503 }
504 #else
505 /* Assume that user mode programs are passing in an ext3fs superblock, not
```

```
506  * a kernel struct super_block.  This will allow us to call the feature-test
507  * macros from user land. */
508 #define EXT3_SB(sb)        (sb)
509 #endif
510
511 #define NEXT_ORPHAN(inode) EXT3_I(inode)->i_dtime
512
513 /*
514  * Codes for operating systems
515  */
516 #define EXT3_OS_LINUX               0
517 #define EXT3_OS_HURD                1
518 #define EXT3_OS_MASIX               2
519 #define EXT3_OS_FREEBSD             3
520 #define EXT3_OS_LITES               4
521
522 /*
523  * Revision levels
524  */
525 #define EXT3_GOOD_OLD_REV        0        /* The good old (original) format */
526 #define EXT3_DYNAMIC_REV         1        /* V2 format w/ dynamic inode sizes */
527
528 #define EXT3_CURRENT_REV         EXT3_GOOD_OLD_REV
529 #define EXT3_MAX_SUPP_REV        EXT3_DYNAMIC_REV
530
531 #define EXT3_GOOD_OLD_INODE_SIZE 128
532
533 /*
534  * Feature set definitions
535  */
536
537 #define EXT3_HAS_COMPAT_FEATURE(sb,mask)                        \
538         ( EXT3_SB(sb)->s_es->s_feature_compat & cpu_to_le32(mask) )
539 #define EXT3_HAS_RO_COMPAT_FEATURE(sb,mask)                      \
540         ( EXT3_SB(sb)->s_es->s_feature_ro_compat & cpu_to_le32(mask) )
541 #define EXT3_HAS_INCOMPAT_FEATURE(sb,mask)                       \
542         ( EXT3_SB(sb)->s_es->s_feature_incompat & cpu_to_le32(mask) )
543 #define EXT3_SET_COMPAT_FEATURE(sb,mask)                         \
544         EXT3_SB(sb)->s_es->s_feature_compat |= cpu_to_le32(mask)
545 #define EXT3_SET_RO_COMPAT_FEATURE(sb,mask)                      \
546         EXT3_SB(sb)->s_es->s_feature_ro_compat |= cpu_to_le32(mask)
547 #define EXT3_SET_INCOMPAT_FEATURE(sb,mask)                       \
548         EXT3_SB(sb)->s_es->s_feature_incompat |= cpu_to_le32(mask)
549 #define EXT3_CLEAR_COMPAT_FEATURE(sb,mask)                       \
550         EXT3_SB(sb)->s_es->s_feature_compat &= ~cpu_to_le32(mask)
551 #define EXT3_CLEAR_RO_COMPAT_FEATURE(sb,mask)                     \
552         EXT3_SB(sb)->s_es->s_feature_ro_compat &= ~cpu_to_le32(mask)
553 #define EXT3_CLEAR_INCOMPAT_FEATURE(sb,mask)                      \
554         EXT3_SB(sb)->s_es->s_feature_incompat &= ~cpu_to_le32(mask)
555
556 #define EXT3_FEATURE_COMPAT_DIR_PREALLOC        0x0001
557 #define EXT3_FEATURE_COMPAT_IMAGIC_INODES       0x0002
558 #define EXT3_FEATURE_COMPAT_HAS_JOURNAL         0x0004
559 #define EXT3_FEATURE_COMPAT_EXT_ATTR            0x0008
560 #define EXT3_FEATURE_COMPAT_RESIZE_INODE        0x0010
561 #define EXT3_FEATURE_COMPAT_DIR_INDEX           0x0020
```

```
562
563 #define EXT3_FEATURE_RO_COMPAT_SPARSE_SUPER        0x0001
564 #define EXT3_FEATURE_RO_COMPAT_LARGE_FILE        0x0002
565 #define EXT3_FEATURE_RO_COMPAT_BTREE_DIR        0x0004
566
567 #define EXT3_FEATURE_INCOMPAT_COMPRESSION        0x0001
568 #define EXT3_FEATURE_INCOMPAT_FILETYPE              0x0002
569 #define EXT3_FEATURE_INCOMPAT_RECOVER              0x0004 /* Needs recovery */
570 #define EXT3_FEATURE_INCOMPAT_JOURNAL_DEV        0x0008 /* Journal device */
571 #define EXT3_FEATURE_INCOMPAT_META_BG              0x0010
572
573 #define EXT3_FEATURE_COMPAT_SUPP        EXT2_FEATURE_COMPAT_EXT_ATTR
574 #define EXT3_FEATURE_INCOMPAT_SUPP        (EXT3_FEATURE_INCOMPAT_FILETYPE| \
575                                           EXT3_FEATURE_INCOMPAT_RECOVER| \
576                                           EXT3_FEATURE_INCOMPAT_META_BG)
577 #define EXT3_FEATURE_RO_COMPAT_SUPP        (EXT3_FEATURE_RO_COMPAT_SPARSE_SUPER| \
578                                           EXT3_FEATURE_RO_COMPAT_LARGE_FILE| \
579                                           EXT3_FEATURE_RO_COMPAT_BTREE_DIR)
580
581 /*
582  * Default values for user and/or group using reserved blocks
583  */
584 #define        EXT3_DEF_RESUID                0
585 #define        EXT3_DEF_RESGID                0
586
587 /*
588  * Default mount options
589  */
590 #define EXT3_DEFM_DEBUG                0x0001
591 #define EXT3_DEFM_BSDGROUPS        0x0002
592 #define EXT3_DEFM_XATTR_USER        0x0004
593 #define EXT3_DEFM_ACL                0x0008
594 #define EXT3_DEFM_UID16                0x0010
595 #define EXT3_DEFM_JMODE                0x0060
596 #define EXT3_DEFM_JMODE_DATA        0x0020
597 #define EXT3_DEFM_JMODE_ORDERED        0x0040
598 #define EXT3_DEFM_JMODE_WBACK        0x0060
599
600 /*
601  * Structure of a directory entry
602  */
603 #define EXT3_NAME_LEN 255
604
605 struct ext3_dir_entry {
606         __le32        inode;                        /* Inode number */
607         __le16        rec_len;                /* Directory entry length */
608         __le16        name_len;                /* Name length */
609         char        name[EXT3_NAME_LEN];        /* File name */
610 };
611
612 /*
613  * The new version of the directory entry.  Since EXT3 structures are
614  * stored in intel byte order, and the name_len field could never be
615  * bigger than 255 chars, it's safe to reclaim the extra byte for the
616  * file_type field.
617  */
```

```
618 struct ext3_dir_entry_2 {
619         __le32        inode;                        /* Inode number */
620         __le16        rec_len;                /* Directory entry length */
621         __u8        name_len;                /* Name length */
622         __u8        file_type;
623         char        name[EXT3_NAME_LEN];        /* File name */
624 };
625
626 /*
627  * Ext3 directory file types.  Only the low 3 bits are used.  The
628  * other bits are reserved for now.
629  */
630 #define EXT3_FT_UNKNOWN                0
631 #define EXT3_FT_REG_FILE        1
632 #define EXT3_FT_DIR                2
633 #define EXT3_FT_CHRDEV                3
634 #define EXT3_FT_BLKDEV                4
635 #define EXT3_FT_FIFO                5
636 #define EXT3_FT_SOCK                6
637 #define EXT3_FT_SYMLINK                7
638
639 #define EXT3_FT_MAX                8
640
641 /*
642  * EXT3_DIR_PAD defines the directory entries boundaries
643  *
644  * NOTE: It must be a multiple of 4
645  */
646 #define EXT3_DIR_PAD                        4
647 #define EXT3_DIR_ROUND                        (EXT3_DIR_PAD - 1)
648 #define EXT3_DIR_REC_LEN(name_len)        (((name_len) + 8 + EXT3_DIR_ROUND) & \
649                                         ~EXT3_DIR_ROUND)
650 /*
651  * Hash Tree Directory indexing
652  * (c) Daniel Phillips, 2001
653  */
654
655 #ifdef CONFIG_EXT3_INDEX
656   #define is_dx(dir) (EXT3_HAS_COMPAT_FEATURE(dir->i_sb, \
657                                         EXT3_FEATURE_COMPAT_DIR_INDEX) && \
658                       (EXT3_I(dir)->i_flags & EXT3_INDEX_FL))
659 #define EXT3_DIR_LINK_MAX(dir) (!is_dx(dir) && (dir)->i_nlink >= EXT3_LINK_MAX)
660 #define EXT3_DIR_LINK_EMPTY(dir) ((dir)->i_nlink == 2 || (dir)->i_nlink == 1)
661 #else
662   #define is_dx(dir) 0
663 #define EXT3_DIR_LINK_MAX(dir) ((dir)->i_nlink >= EXT3_LINK_MAX)
664 #define EXT3_DIR_LINK_EMPTY(dir) ((dir)->i_nlink == 2)
665 #endif
666
667 /* Legal values for the dx_root hash_version field: */
668
669 #define DX_HASH_LEGACY                0
670 #define DX_HASH_HALF_MD4        1
671 #define DX_HASH_TEA                2
672
673 #ifdef __KERNEL__
```

```
674
675 /* hash info structure used by the directory hash */
676 struct dx_hash_info
677 {
678         u32                hash;
679         u32                minor_hash;
680         int                hash_version;
681         u32                *seed;
682 };
683
684 #define EXT3_HTREE_EOF        0x7fffffff
685
686 /*
687  * Control parameters used by ext3_htree_next_block
688  */
689 #define HASH_NB_ALWAYS                1
690
691
692 /*
693  * Describe an inode's exact location on disk and in memory
694  */
695 struct ext3_iloc
696 {
697         struct buffer_head *bh;
698         unsigned long offset;
699         unsigned long block_group;
700 };
701
702 static inline struct ext3_inode *ext3_raw_inode(struct ext3_iloc *iloc)
703 {
704         return (struct ext3_inode *) (iloc->bh->b_data + iloc->offset);
705 }
706
707 /*
708  * This structure is stuffed into the struct file's private_data field
709  * for directories.  It is where we put information so that we can do
710  * readdir operations in hash tree order.
711  */
712 struct dir_private_info {
713         struct rb_root        root;
714         struct rb_node        *curr_node;
715         struct fname        *extra_fname;
716         loff_t                last_pos;
717         __u32                curr_hash;
718         __u32                curr_minor_hash;
719         __u32                next_hash;
720 };
721
722 /* calculate the first block number of the group */
723 static inline ext3_fsblk_t
724 ext3_group_first_block_no(struct super_block *sb, unsigned long group_no)
725 {
726         return group_no * (ext3_fsblk_t)EXT3_BLOCKS_PER_GROUP(sb) +
727                 le32_to_cpu(EXT3_SB(sb)->s_es->s_first_data_block);
728 }
729
```

```
730 /*
731  * Special error return code only used by dx_probe() and its callers.
732  */
733 #define ERR_BAD_DX_DIR        -75000
734
735 /*
736  * Function prototypes
737  */
738
739 /*
740  * Ok, these declarations are also in <linux/kernel.h> but none of the
741  * ext3 source programs needs to include it so they are duplicated here.
742  */
743 # define NORET_TYPE    /**/
744 # define ATTRIB_NORET  __attribute__((noreturn))
745 # define NORET_AND     noreturn,
746
747 /* balloc.c */
748 extern int ext3_bg_has_super(struct super_block *sb, int group);
749 extern unsigned long ext3_bg_num_gdb(struct super_block *sb, int group);
750 extern ext3_fsblk_t ext3_new_block (handle_t *handle, struct inode *inode,
751                         ext3_fsblk_t goal, int *errp);
752 extern ext3_fsblk_t ext3_new_blocks (handle_t *handle, struct inode *inode,
753                         ext3_fsblk_t goal, unsigned long *count, int *errp);
754 extern void ext3_free_blocks (handle_t *handle, struct inode *inode,
755                         ext3_fsblk_t block, unsigned long count);
756 extern void ext3_free_blocks_sb (handle_t *handle, struct super_block *sb,
757                                  ext3_fsblk_t block, unsigned long count,
758                                 unsigned long *pdquot_freed_blocks);
759 extern ext3_fsblk_t ext3_count_free_blocks (struct super_block *);
760 extern void ext3_check_blocks_bitmap (struct super_block *);
761 extern struct ext3_group_desc * ext3_get_group_desc(struct super_block * sb,
762                                                     unsigned int block_group,
763                                                     struct buffer_head ** bh);
764 extern int ext3_should_retry_alloc(struct super_block *sb, int *retries);
765 extern void ext3_init_block_alloc_info(struct inode *);
766 extern void ext3_rsv_window_add(struct super_block *sb, struct ext3_reserve_window_node *rsv);
767
768 /* dir.c */
769 extern int ext3_check_dir_entry(const char *, struct inode *,
770                                 struct ext3_dir_entry_2 *,
771                                 struct buffer_head *, unsigned long);
772 extern int ext3_htree_store_dirent(struct file *dir_file, __u32 hash,
773                                    __u32 minor_hash,
774                                    struct ext3_dir_entry_2 *dirent);
775 extern void ext3_htree_free_dir_info(struct dir_private_info *p);
776
777 /* fsync.c */
778 extern int ext3_sync_file (struct file *, struct dentry *, int);
779
780 /* hash.c */
781 extern int ext3fs_dirhash(const char *name, int len, struct
782                           dx_hash_info *hinfo);
783
784 /* ialloc.c */
785 extern struct inode * ext3_new_inode (handle_t *, struct inode *, int);
```

```
786 extern void ext3_free_inode (handle_t *, struct inode *);
787 extern struct inode * ext3_orphan_get (struct super_block *, unsigned long);
788 extern unsigned long ext3_count_free_inodes (struct super_block *);
789 extern unsigned long ext3_count_dirs (struct super_block *);
790 extern void ext3_check_inodes_bitmap (struct super_block *);
791 extern unsigned long ext3_count_free (struct buffer_head *, unsigned);
792
793
794 /* inode.c */
795 int ext3_forget(handle_t *handle, int is_metadata, struct inode *inode,
796                 struct buffer_head *bh, ext3_fsblk_t blocknr);
797 struct buffer_head * ext3_getblk (handle_t *, struct inode *, long, int, int *);
798 struct buffer_head * ext3_bread (handle_t *, struct inode *, int, int, int *);
799 int ext3_get_blocks_handle(handle_t *handle, struct inode *inode,
800         sector_t iblock, unsigned long maxblocks, struct buffer_head *bh_result,
801         int create, int extend_disksize);
802
803 extern void ext3_read_inode (struct inode *);
804 extern int  ext3_write_inode (struct inode *, int);
805 extern int  ext3_setattr (struct dentry *, struct iattr *);
806 extern void ext3_delete_inode (struct inode *);
807 extern int  ext3_sync_inode (handle_t *, struct inode *);
808 extern void ext3_discard_reservation (struct inode *);
809 extern void ext3_dirty_inode(struct inode *);
810 extern int ext3_change_inode_journal_flag(struct inode *, int);
811 extern int ext3_get_inode_loc(struct inode *, struct ext3_iloc *);
812 extern void ext3_truncate (struct inode *);
813 extern void ext3_set_inode_flags(struct inode *);
814 extern void ext3_set_aops(struct inode *inode);
815
816 /* ioctl.c */
817 extern int ext3_ioctl (struct inode *, struct file *, unsigned int,
818                        unsigned long);
819
820 /* namei.c */
821 extern int ext3_orphan_add(handle_t *, struct inode *);
822 extern int ext3_orphan_del(handle_t *, struct inode *);
823 extern int ext3_htree_fill_tree(struct file *dir_file, __u32 start_hash,
824                                 __u32 start_minor_hash, __u32 *next_hash);
825
826 /* resize.c */
827 extern int ext3_group_add(struct super_block *sb,
828                                 struct ext3_new_group_data *input);
829 extern int ext3_group_extend(struct super_block *sb,
830                                 struct ext3_super_block *es,
831                                 ext3_fsblk_t n_blocks_count);
832
833 /* super.c */
834 extern void ext3_error (struct super_block *, const char *, const char *, ...)
835         __attribute__ ((format (printf, 3, 4)));
836 extern void __ext3_std_error (struct super_block *, const char *, int);
837 extern void ext3_abort (struct super_block *, const char *, const char *, ...)
838         __attribute__ ((format (printf, 3, 4)));
839 extern void ext3_warning (struct super_block *, const char *, const char *, ...)
840         __attribute__ ((format (printf, 3, 4)));
841 extern void ext3_update_dynamic_rev (struct super_block *sb);
```

```
842
843 #define ext3_std_error(sb, errno)                                \
844 do {                                                             \
845         if ((errno))                                             \
846                 __ext3_std_error((sb), __FUNCTION__, (errno));       \
847 } while (0)
848
849 /*
850  * Inodes and files operations
851  */
852
853 /* dir.c */
854 extern const struct file_operations ext3_dir_operations;
855
856 /* file.c */
857 extern struct inode_operations ext3_file_inode_operations;
858 extern const struct file_operations ext3_file_operations;
859
860 /* namei.c */
861 extern struct inode_operations ext3_dir_inode_operations;
862 extern struct inode_operations ext3_special_inode_operations;
863
864 /* symlink.c */
865 extern struct inode_operations ext3_symlink_inode_operations;
866 extern struct inode_operations ext3_fast_symlink_inode_operations;
867
868
869 #endif        /* __KERNEL__ */
870
871 #endif        /* _LINUX_EXT3_FS_H */
```

This header file contains the definitions of structures for ext3 file system. That header file is normally in a standard place: /usr/include/linux. If it is not there, download it from here and include in your program.

Below is a Makefile to compile the program. ¡

```
1 all: ext3parse
2
3 ext3parse: ext3parse.c
4         gcc -Wall -o ext3parse ext3parse.c
5
6 clean:
7         rm -fr *~  ext3parse
```

We can run the program as follows:

```
sudo ./ext3parse /dev/sda5
```

We can get the following output.

```
1
2   superblock information:
3   size_of_super_block_structure=1024
4   inode_count=3662848
5   block_count=7323624
6   first_data_block=0
7   magic_number=ef53
8   inode_size=128
9   inodes_per_group=16352
10
11  group descriptor table content in block 1
12  size of group desc structure  = 32
13  group 0: bitmap_block#=1027 inodetable_block#=1029 free_block_count=0
14  group 1: bitmap_block#=33795 inodetable_block#=33797 free_block_count=0
15  group 2: bitmap_block#=65536 inodetable_block#=65538 free_block_count=0
16  group 3: bitmap_block#=99331 inodetable_block#=99333 free_block_count=6056
17  group 4: bitmap_block#=131072 inodetable_block#=131074 free_block_count=16438
18  group 5: bitmap_block#=164867 inodetable_block#=164869 free_block_count=9561
19  group 6: bitmap_block#=196608 inodetable_block#=196610 free_block_count=411
20  group 7: bitmap_block#=230403 inodetable_block#=230405 free_block_count=20787
21  group 8: bitmap_block#=262144 inodetable_block#=262146 free_block_count=8197
22  group 9: bitmap_block#=295939 inodetable_block#=295941 free_block_count=7057
23  group 10: bitmap_block#=327680 inodetable_block#=327682 free_block_count=19058
24  group 11: bitmap_block#=360448 inodetable_block#=360450 free_block_count=9858
25  group 12: bitmap_block#=393216 inodetable_block#=393218 free_block_count=19031
26  group 13: bitmap_block#=425984 inodetable_block#=425986 free_block_count=12506
27  group 14: bitmap_block#=458752 inodetable_block#=458754 free_block_count=11693
28  group 15: bitmap_block#=491520 inodetable_block#=491522 free_block_count=0
29  group 16: bitmap_block#=524288 inodetable_block#=524290 free_block_count=2977
30  group 17: bitmap_block#=557056 inodetable_block#=557058 free_block_count=10290
31  group 18: bitmap_block#=589824 inodetable_block#=589826 free_block_count=2921
32  group 19: bitmap_block#=622592 inodetable_block#=622594 free_block_count=9085
33  group 20: bitmap_block#=655360 inodetable_block#=655362 free_block_count=820
34  group 21: bitmap_block#=688128 inodetable_block#=688130 free_block_count=1458
35  group 22: bitmap_block#=720896 inodetable_block#=720898 free_block_count=6662
36  group 23: bitmap_block#=753664 inodetable_block#=753666 free_block_count=2082
37  group 24: bitmap_block#=786432 inodetable_block#=786434 free_block_count=633
38  group 25: bitmap_block#=820227 inodetable_block#=820229 free_block_count=639
39  group 26: bitmap_block#=851968 inodetable_block#=851970 free_block_count=416
40  group 27: bitmap_block#=885763 inodetable_block#=885765 free_block_count=797
41  group 28: bitmap_block#=917504 inodetable_block#=917506 free_block_count=531
42  group 29: bitmap_block#=950272 inodetable_block#=950274 free_block_count=524
43  group 30: bitmap_block#=983040 inodetable_block#=983042 free_block_count=3324
44  group 31: bitmap_block#=1015808 inodetable_block#=1015810 free_block_count=5014
45  group 32: bitmap_block#=1048576 inodetable_block#=1048578 free_block_count=3724
46  group 33: bitmap_block#=1081344 inodetable_block#=1081346 free_block_count=7
47  group 34: bitmap_block#=1114112 inodetable_block#=1114114 free_block_count=3
48  group 35: bitmap_block#=1146880 inodetable_block#=1146882 free_block_count=41
49  group 36: bitmap_block#=1179648 inodetable_block#=1179650 free_block_count=0
50  group 37: bitmap_block#=1212416 inodetable_block#=1212418 free_block_count=40
51  group 38: bitmap_block#=1245184 inodetable_block#=1245186 free_block_count=26
52  group 39: bitmap_block#=1277952 inodetable_block#=1277954 free_block_count=76
53  group 40: bitmap_block#=1310720 inodetable_block#=1310722 free_block_count=11
54  group 41: bitmap_block#=1343488 inodetable_block#=1343490 free_block_count=26142
55  group 42: bitmap_block#=1376256 inodetable_block#=1376258 free_block_count=32255
56  group 43: bitmap_block#=1409024 inodetable_block#=1409026 free_block_count=32054
```

```
group 44: bitmap_block#=1441792 inodetable_block#=1441794 free_block_count=32113
group 45: bitmap_block#=1474560 inodetable_block#=1474562 free_block_count=32250
group 46: bitmap_block#=1507328 inodetable_block#=1507330 free_block_count=32254
group 47: bitmap_block#=1540096 inodetable_block#=1540098 free_block_count=32222
group 48: bitmap_block#=1572864 inodetable_block#=1572866 free_block_count=32248
group 49: bitmap_block#=1606659 inodetable_block#=1606661 free_block_count=31153
group 50: bitmap_block#=1638400 inodetable_block#=1638402 free_block_count=32248
group 51: bitmap_block#=1671168 inodetable_block#=1671170 free_block_count=32250
group 52: bitmap_block#=1703936 inodetable_block#=1703938 free_block_count=32255
group 53: bitmap_block#=1736704 inodetable_block#=1736706 free_block_count=32255
group 54: bitmap_block#=1769472 inodetable_block#=1769474 free_block_count=32249
group 55: bitmap_block#=1802240 inodetable_block#=1802242 free_block_count=32253
group 56: bitmap_block#=1835008 inodetable_block#=1835010 free_block_count=17990
group 57: bitmap_block#=1867776 inodetable_block#=1867778 free_block_count=28486
group 58: bitmap_block#=1900544 inodetable_block#=1900546 free_block_count=32255
group 59: bitmap_block#=1933312 inodetable_block#=1933314 free_block_count=31990
group 60: bitmap_block#=1966080 inodetable_block#=1966082 free_block_count=32255
group 61: bitmap_block#=1998848 inodetable_block#=1998850 free_block_count=32168
group 62: bitmap_block#=2031616 inodetable_block#=2031618 free_block_count=32255
group 63: bitmap_block#=2064384 inodetable_block#=2064386 free_block_count=31965
group 64: bitmap_block#=2097152 inodetable_block#=2097154 free_block_count=32255
group 65: bitmap_block#=2129920 inodetable_block#=2129922 free_block_count=32093
group 66: bitmap_block#=2162688 inodetable_block#=2162690 free_block_count=32248
group 67: bitmap_block#=2195456 inodetable_block#=2195458 free_block_count=32246
group 68: bitmap_block#=2228224 inodetable_block#=2228226 free_block_count=32126
group 69: bitmap_block#=2260992 inodetable_block#=2260994 free_block_count=32131
group 70: bitmap_block#=2293760 inodetable_block#=2293762 free_block_count=32255
group 71: bitmap_block#=2326528 inodetable_block#=2326530 free_block_count=32055
group 72: bitmap_block#=2359296 inodetable_block#=2359298 free_block_count=27628
group 73: bitmap_block#=2392064 inodetable_block#=2392066 free_block_count=31967
group 74: bitmap_block#=2424832 inodetable_block#=2424834 free_block_count=32255
group 75: bitmap_block#=2457600 inodetable_block#=2457602 free_block_count=32122
group 76: bitmap_block#=2490368 inodetable_block#=2490370 free_block_count=32255
group 77: bitmap_block#=2523136 inodetable_block#=2523138 free_block_count=31904
group 78: bitmap_block#=2555904 inodetable_block#=2555906 free_block_count=32255
group 79: bitmap_block#=2588672 inodetable_block#=2588674 free_block_count=31877
group 80: bitmap_block#=2621440 inodetable_block#=2621442 free_block_count=32212
group 81: bitmap_block#=2655235 inodetable_block#=2655237 free_block_count=31091
group 82: bitmap_block#=2686976 inodetable_block#=2686978 free_block_count=32255
group 83: bitmap_block#=2719744 inodetable_block#=2719746 free_block_count=32255
group 84: bitmap_block#=2752512 inodetable_block#=2752514 free_block_count=7665
group 85: bitmap_block#=2785280 inodetable_block#=2785282 free_block_count=1059
group 86: bitmap_block#=2818048 inodetable_block#=2818050 free_block_count=6001
group 87: bitmap_block#=2850816 inodetable_block#=2850818 free_block_count=16517
group 88: bitmap_block#=2883584 inodetable_block#=2883586 free_block_count=32249
group 89: bitmap_block#=2916352 inodetable_block#=2916354 free_block_count=31990
group 90: bitmap_block#=2949120 inodetable_block#=2949122 free_block_count=32255
group 91: bitmap_block#=2981888 inodetable_block#=2981890 free_block_count=31895
group 92: bitmap_block#=3014656 inodetable_block#=3014658 free_block_count=32255
group 93: bitmap_block#=3047424 inodetable_block#=3047426 free_block_count=32177
group 94: bitmap_block#=3080192 inodetable_block#=3080194 free_block_count=32125
group 95: bitmap_block#=3112960 inodetable_block#=3112962 free_block_count=32145
group 96: bitmap_block#=3145728 inodetable_block#=3145730 free_block_count=32255
group 97: bitmap_block#=3178496 inodetable_block#=3178498 free_block_count=21762
group 98: bitmap_block#=3211264 inodetable_block#=3211266 free_block_count=1645
group 99: bitmap_block#=3244032 inodetable_block#=3244034 free_block_count=0
```

```
group 100: bitmap_block#=3276800 inodetable_block#=3276802 free_block_count=0
group 101: bitmap_block#=3309568 inodetable_block#=3309570 free_block_count=14551
group 102: bitmap_block#=3342336 inodetable_block#=3342338 free_block_count=17441
group 103: bitmap_block#=3375104 inodetable_block#=3375106 free_block_count=18916
group 104: bitmap_block#=3407872 inodetable_block#=3407874 free_block_count=17769
group 105: bitmap_block#=3440640 inodetable_block#=3440642 free_block_count=17709
group 106: bitmap_block#=3473408 inodetable_block#=3473410 free_block_count=1487
group 107: bitmap_block#=3506176 inodetable_block#=3506178 free_block_count=0
group 108: bitmap_block#=3538944 inodetable_block#=3538946 free_block_count=0
group 109: bitmap_block#=3571712 inodetable_block#=3571714 free_block_count=815
group 110: bitmap_block#=3604480 inodetable_block#=3604482 free_block_count=30796
group 111: bitmap_block#=3637248 inodetable_block#=3637250 free_block_count=32241
group 112: bitmap_block#=3670016 inodetable_block#=3670018 free_block_count=32027
group 113: bitmap_block#=3702784 inodetable_block#=3702786 free_block_count=32116
group 114: bitmap_block#=3735552 inodetable_block#=3735554 free_block_count=29420
group 115: bitmap_block#=3768320 inodetable_block#=3768322 free_block_count=28907
group 116: bitmap_block#=3801088 inodetable_block#=3801090 free_block_count=32088
group 117: bitmap_block#=3833856 inodetable_block#=3833858 free_block_count=31899
group 118: bitmap_block#=3866624 inodetable_block#=3866626 free_block_count=31867
group 119: bitmap_block#=3899392 inodetable_block#=3899394 free_block_count=31832
group 120: bitmap_block#=3932160 inodetable_block#=3932162 free_block_count=31530
group 121: bitmap_block#=3964928 inodetable_block#=3964930 free_block_count=31904
group 122: bitmap_block#=3997696 inodetable_block#=3997698 free_block_count=32163
group 123: bitmap_block#=4030464 inodetable_block#=4030466 free_block_count=31961
group 124: bitmap_block#=4063232 inodetable_block#=4063234 free_block_count=32168
group 125: bitmap_block#=4097027 inodetable_block#=4097029 free_block_count=31198
group 126: bitmap_block#=4128768 inodetable_block#=4128770 free_block_count=31693
group 127: bitmap_block#=4161536 inodetable_block#=4161538 free_block_count=32227

root directory content
type=2 inode=2          name = .
type=2 inode=2          name = ..
type=2 inode=11         name = lost+found
type=2 inode=915713     name = etc
type=2 inode=1945889    name = proc
type=2 inode=2959713    name = sys
type=2 inode=2534561    name = dev
type=2 inode=1373569    name = var
type=2 inode=3008769    name = usr
type=2 inode=1586145    name = opt
type=2 inode=3270401    name = bin
type=2 inode=1177345    name = boot
type=2 inode=3482977    name = home
type=2 inode=130817     name = lib
type=2 inode=3057825    name = media
type=2 inode=2665377    name = mnt
type=2 inode=474209     name = root
type=2 inode=3581089    name = sbin
type=2 inode=1618849    name = srv
type=2 inode=3074177    name = tmp
type=1 inode=52246      name = session_mm_apache2handler0.sem
```

9.2 Direct Access To File System

In this section, we will present how we can directly access a disk and a file system. We will be working with **ext2** file system of Linux. It is the native file system of the Linux operating system.

The ext file system (the extended file system) is an extended version of the file system of the Minix operating system. It is constantly improved and we have versions like ext, ext2, ext3, and now ext4.

The major feature on the newer versions, version 3 and 4, is the journaling support. Another major feature added to the ext4 is the use of extents in allocating blocks to a file, so that access to the blocks is much faster for large files. The ext2 and ext3 file systems are very similar to each other. Their on-disk data structures are nearly the same. We will work with ext2 file system here, because it has a lot of documentation on the web, explaining the internals and the design of the ext2 file system. It is a robust and popular file system. Learning the internals of that will enable you to learn the internals of other file systems very easily.

We can install an ext2 file system on a partition of an hard-disk. We call this as *making* a file system, or *formatting* the partition. This is *high level formatting*. When you install Linux in one your partitions of your hard disk, the Linux file system is also installed in that partition (i.e. that partition is formatted with the Linux file system). Then Linux files, directories, and programs are placed into that file system. The root directory of a Linux file system has a special name which is "/".

At any time, when you are running Linux, you can install (make) a new Linux file system in an unused partition as well. The command to do that in Linux is `mke2fs`. You have to specify where (in which partition, or storage device) you will make the file system. For that you use the corresponding special device file to refer to the storage device or partition.

For example, a partition of a hard disk may have a special device file in /dev directory that may be called /dev/sda5. Another partition of the hard disk may have a correponding special device file that is called /dev/sda4. Similarly, other storage devices and all other devices connected to the computer may have a corresponding special file created in the /dev directory. The cdrom, for example, has a corresponding special device file that has the name /dev/cdrom usually.

The special file corresponding to a storage device can be used to refer to the device and access the device. So, while making a file system, we will use the name of the corresponding special file to specify where (on which partition or storage device) we will make the file system.

Lets say we have an empty and unused partition in our hard-drive, which has the corresponding special file `/dev/sdaX`. Then we can create an ext2 file system on it using the

following command:

```
mke2fs -t ext2 /dev/sdaX
```

This will create an ext2 file system on that partition. The mke2fs command has a numbef of optional parameters. You can learn about them by typing "man mke2fs". One of those parameters is the block count. You can specify how many blocks the file system will have. Another parameter is the block size. You can specify the block size. It can be, for example, 1024 bytes, or 4096 bytes. You can also specify, for example, the inode size. You can set it 128 or 256.

Similarly, you can create a file system on pseudo storage device (virtual disk). That pseudo device, for example, can be a large file. Than, that large file will be acting as your storage device (pseudo-disk or file-disk or file store). It will have a corresponding special device created in the /dev directory.

We can create such a file store as follows:

```
sudo dd if=/dev/zero of=filedisk.img bs=4K count=250000
```

This will create a large file in the current directory called `filedisk.img`. That file will act as a virtual disk (virtual storage). The virtual disk will have 250000 blocks and block size is 4 KB. You can craete such a file with different number of blocks and block size.

In the above example, the create file is quite large a file: nearly 1 GB. That means we will be working with a virtual disk that is 1 GB.

Now we can make that file a block oriented file (virtual disk) that can be accessed via a block special device file. We can do this by the following command.

```
sudo /sbin/losetup /dev/loop0 filedisk.img
```

If we have "device busy" error, we can try using /dev/loop1. The /dev/loop0 (a loop device) was already in the /dev directory. We are just setting it up to correspond to our file-store `filedisk.img`.

Now we have a block-oriented pseudo storage device `filedisk.img` that has the corresponding device file `/dev/loop0`. That is a block oriented device (i.e. access to that happens in blocks; in other words, the transfer unit is block, not byte).

Making a File System (Formatting)

We can now create a file system on this. For that we type:

```
sudo /sbin/mke2fs -t ext2 -b4096 -I 128 /dev/loop0 250000 -O
 ^ext_attr,^resize_inode,^dir_index,^sparse_super
```

Note that the above command is just a single line.

This command will create an ext2 file system on /dev/loop0, that has 250000 blocks. The file system will be placed into our large file store `filedisk.img` that is referred with the special device file /dev/loop0. The block size of the created file system is 4096 bytes. The command has some other options. The ŝign before an option name disables the option. For example, d̂ir_index indicates that the file system will not use a hashed B tree to speed up the directory lookup operations. You can learn those options by reading the man page of mke2fs command.

When we make the file system using mke2fs program, We get the following output:

```
1  mke2fs 1.41.9 (22-Aug-2009)
2  Filesystem label=
3  OS type: Linux
4  Block size=4096 (log=2)
5  Fragment size=4096 (log=2)
6  62720 inodes, 250000 blocks
7  12500 blocks (5.00%) reserved for the super user
8  First data block=0
9  8 block groups
10 32768 blocks per group, 32768 fragments per group
11 7840 inodes per group
12 Superblock backups stored on blocks:
13         32768, 65536, 98304, 131072, 163840, 196608, 229376
14
15 Writing inode tables: done
16 Writing superblocks and filesystem accounting information: done
17
18 This filesystem will be automatically checked every 34 mounts or
19 180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

With this command the file system is created in the /dev/loop0 device (in the filedisk.img file). That means the virtual disk is formatted with ext2 file system. The related file system structures are placed and initialized on the (pseudo) device. A root directory is create with name "/".

We can now check that the virtual disk is really formatted by using the dumpe2fs command. Read the man page of dumpefs utility.

```
/sbin/dumpe2fs /dev/loop0
```

We can get such an output:

```
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          cc718714-f6fb-4271-b0ff-3716cf5cb661
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      filetype
Filesystem flags:         signed_directory_hash
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              62720
Block count:              250000
Reserved block count:     12500
Free blocks:              248003
Free inodes:              62709
First block:              0
Block size:               4096
Fragment size:            4096
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         7840
Inode blocks per group:   245
Filesystem created:       Wed Apr 28 11:38:00 2010
Last mount time:          n/a
Last write time:          Wed Apr 28 11:38:00 2010
Mount count:              0
Maximum mount count:      34
Last checked:             Wed Apr 28 11:38:00 2010
Check interval:           15552000 (6 months)
Next check after:         Mon Oct 25 11:38:00 2010
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:                  128
Default directory hash:   half_md4
Directory Hash Seed:      a5815398-974e-45a7-be53-f95850646bde


Group 0: (Blocks 0-32767)
  Primary superblock at 0, Group descriptors at 1-1
  Block bitmap at 2 (+2), Inode bitmap at 3 (+3)
  Inode table at 4-248 (+4)
  32514 free blocks, 7829 free inodes, 2 directories
  Free blocks: 254-32767
  Free inodes: 12-7840
Group 1: (Blocks 32768-65535)
  Backup superblock at 32768, Group descriptors at 32769-32769
  Block bitmap at 32770 (+2), Inode bitmap at 32771 (+3)
  Inode table at 32772-33016 (+4)
  32519 free blocks, 7840 free inodes, 0 directories
  Free blocks: 33017-65535
  Free inodes: 7841-15680
Group 2: (Blocks 65536-98303)
  Backup superblock at 65536, Group descriptors at 65537-65537
  Block bitmap at 65538 (+2), Inode bitmap at 65539 (+3)
```

```
  Inode table at 65540-65784 (+4)
  32519 free blocks, 7840 free inodes, 0 directories
  Free blocks: 65785-98303
  Free inodes: 15681-23520
Group 3: (Blocks 98304-131071)
  Backup superblock at 98304, Group descriptors at 98305-98305
  Block bitmap at 98306 (+2), Inode bitmap at 98307 (+3)
  Inode table at 98308-98552 (+4)
  32519 free blocks, 7840 free inodes, 0 directories
  Free blocks: 98553-131071
  Free inodes: 23521-31360
Group 4: (Blocks 131072-163839)
  Backup superblock at 131072, Group descriptors at 131073-131073
  Block bitmap at 131074 (+2), Inode bitmap at 131075 (+3)
  Inode table at 131076-131320 (+4)
  32519 free blocks, 7840 free inodes, 0 directories
  Free blocks: 131321-163839
  Free inodes: 31361-39200
Group 5: (Blocks 163840-196607)
  Backup superblock at 163840, Group descriptors at 163841-163841
  Block bitmap at 163842 (+2), Inode bitmap at 163843 (+3)
  Inode table at 163844-164088 (+4)
  32519 free blocks, 7840 free inodes, 0 directories
  Free blocks: 164089-196607
  Free inodes: 39201-47040
Group 6: (Blocks 196608-229375)
  Backup superblock at 196608, Group descriptors at 196609-196609
  Block bitmap at 196610 (+2), Inode bitmap at 196611 (+3)
  Inode table at 196612-196856 (+4)
  32519 free blocks, 7840 free inodes, 0 directories
  Free blocks: 196857-229375
  Free inodes: 47041-54880
Group 7: (Blocks 229376-249999)
  Backup superblock at 229376, Group descriptors at 229377-229377
  Block bitmap at 229378 (+2), Inode bitmap at 229379 (+3)
  Inode table at 229380-229624 (+4)
  20375 free blocks, 7840 free inodes, 0 directories
  Free blocks: 229625-249999
  Free inodes: 54881-62720
```

The output gives information about the newly created file system.

9.2.1 Mounting a File System

We can now mount this new file system into our local directory tree. Lets create first a mount point in the /mnt directory of our local directory tree (the directory tree of the the file system that is created when we installed Linux).

```
sudo mkdir /mnt/myfs
```

The mount point (i.e. the subdirectory in our local file system) is `/mnt/myfs`.

We will change the owner of that directory to be the current user (not the root user) by the following command. In thiy the user will be able to acess the file system without doing sudo. The current user in our system is 'korpe'. So we type the following command:

```
sudo chown korpe:korpe /mnt/myfs
```

We will now mount to that point the new file system (i.e. we will attach the new file system to that point in our local -main- file system). For that we use the following command:

```
sudo mount /dev/loop0 /mnt/myfs
```

We can see the mounted file systems in our computer using the `mount` command, Type:

```
mount
```

We will get the following output that is giving information about the currently mounted file systems.

```
1  /dev/sda6 on / type ext4 (rw,errors=remount-ro)
2  proc on /proc type proc (rw)
3  none on /sys type sysfs (rw,noexec,nosuid,nodev)
4  none on /sys/fs/fuse/connections type fusectl (rw)
5  none on /sys/kernel/debug type debugfs (rw)
6  none on /sys/kernel/security type securityfs (rw)
7  udev on /dev type tmpfs (rw,mode=0755)
8  none on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
9  none on /dev/shm type tmpfs (rw,nosuid,nodev)
10 none on /var/run type tmpfs (rw,nosuid,mode=0755)
11 none on /var/lock type tmpfs (rw,noexec,nosuid,nodev)
12 none on /lib/init/rw type tmpfs (rw,nosuid,mode=0755)
13 none on /proc/fs/vmblock/mountPoint type vmblock (rw)
14 binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,noexec,nosuid,nodev)
15 gvfs-fuse-daemon on /home/korpe/.gvfs type fuse.gvfs-fuse-daemon (rw,nosuid,nodev,user=korpe)
16 /dev/loop0 on /mnt/myfs type ext2 (rw)
```

The last line shows that the new file system sitting on virtual disk /dev/loop0 is mounted to the point /mnt/myfs in our directory tree.

We can unmount a file system using the umount command. To unmount our new file system we type:

`sudo umount /dev/loop0` or `sudo umount /mnt/myfs`

You can mount and unmount a file system as many times as you wish.

9.2.2 Using a Mounted Filesystem

This is simple. You already know it. We can now change our current directory to be the mount point: /mnt/myfs. When we do that, that means we changing in fact into the root directory of the new file system. To change into that directory we simple type:

`cd /mnt/myfs`.

Then we can do whatever we would like to do in that directory: list directory content, create a new file, create a new directory, etc. The new file will be created in that new file system that is sitting on the virtual disk filedisk.img accessed via /dev/loop0. For example, we can create an empty file `file1.txt` using the touch command.

```
touch file1.txt
```

We have a file file1.txt created in the root directory of the virtual disk. We can also create directories in the new file system. Let us create a directory dir1 in the root directory. We type:

```
mkdir dir1 cd dir1
```

We created a directory dir1 and changed into that. In that directory, we can create another directory and change into it.

```
$ mkdir dir2 $ cd dir2
```

Now we are in /dir1/dir2 directory of the new file system. We can refer to that directory as /mnt/myfs/dir1/dir2 from our existing file system that was installed at the time we installed Linux.

We can create some files in that directory. Let us create four files:

```
$ touch x
$ touch y
$ touch z
$ touch w
```

We edit the files x and w and put some text into them.

Lets is overwrite the file y with some existing file from our home directory.

```
$ cp ~korpe/afile /mnt/myfs/dir1/dir2/y
```

Now the file y in dirrectory dir2 is no longer an empty file. It is a binary file. We can list the file currently existing in the dir2 directory as follows:

```
korpe@pckorpe:/mnt/myfs/dir1/dir2$ ls -la
total 52
drwxr-xr-x 2 korpe korpe  4096 2010-04-28 16:43 .
drwxr-xr-x 3 korpe korpe  4096 2010-04-28 14:55 ..
-rw-r--r-- 1 korpe korpe    52 2010-04-28 15:07 w
-rw-r--r-- 1 korpe korpe   100 2010-04-28 14:55 x
-rwxr-xr-x 1 korpe korpe 35479 2010-04-28 16:46 y
-rw-r--r-- 1 korpe korpe     0 2010-04-28 14:55 z
```

As the output shows the file y is a non-empty file and its size is 35479 bytes.

9.2.3 Getting Info about Files and Inodes in a File System

The following two utility programs of Linux are very helpful in getting information about a file, its inode, and its physical disk blocks: `stat`, and `/sbin/debugfs`. For example, if we say `stat w` for a file w in the current directory (/dir1/dir2/w of the new file system) , the following information will be printed out:

```
korpe@pckorpe:/mnt/myfs/dir1/dir2$ stat w
  File: ‘w’
  Size: 0          Blocks: 0          IO Block: 4096   regular empty file
Device: 701h/1793d Inode: 7843        Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   korpe)   Gid: ( 1000/   korpe)
Access: 2010-04-28 15:00:50.000000000 +0300
Modify: 2010-04-28 15:00:50.000000000 +0300
Change: 2010-04-28 15:00:50.000000000 +0300
```

The index (number) of the inode allocated to the file w is: 7843. We can use the debugfs program to find out the block numbers that store the content of the file w. For that we can type:

```
sudo/sbin/debugfs /dev/loop0
```

We will get a command prompt. At the prompt we can type:

`bmap /dir1/dir2/w 0`.

That means we would like to get the physical block number where the relative block 0 of the file w is sitting on. The output we will get will be: `53248`. It is quite a short file, so it can fit into a single block.

```
korpe@pckorpe:/mnt/myfs/dir1/dir2$ sudo debugfs /dev/loop0
debugfs 1.41.9 (22-Aug-2009)
debugfs:  bmap /dir1/dir2/w
bmap: Usage: bmap <file> logical_blk
debugfs:  bmap /dir1/dir2/w 0
53248
debugfs:
```

You can see the available commands in the debugs by typing `help` in the command prompt of the debugfs.

9.2.4 Directly Accessing Disk and File System

We can access a disk partition (or a storage device, or a virtual disk like the one that we have created) directly by opening the corresponding device file in the /dev directory. For example, in a program, we can open the device file /dev/loop0 this is corresponding to the virtual disk that we have craeted and formatted with ext2. Then we can read the whole virtual disk byte by byte, or block by block using read and write system calls. We can also access random locations using the lseek system call.

We now present a user level program (findblocks.c), a tool that we developed, (that has to be executed with superuser privilage - sudo) that opens and accesses a virtual disk (with an ext2 file system on it) in raw mode. The program can access *any* block of the storage directly. A block may contain file system data (metadata) or user data (file data/content).

Note that if you are using this program to operate on your hard disk that you are currently using, make sure that the corresponding special device file is opened in READ-ONLY mode. Do not open the partition for writing and do not write something to the partition in raw mode, unless you are very careful and you know what you are doing. You can corrupt your file system in an unrecoverable way.

We will use this program to operate on our virtual disk, therefore we can open the corresponding device in Read/Write mode if we wish. Here since we will just read the disk content block by block, we will open the device file in Read-Only mode.

Below is the program `findblocks.c`.

```
1
2   /* -*- linux-c -*- */
3   /* $Id: findblocks.c,v 1.28 2009/05/14 17:52:44 korpe Exp korpe $ */
4
5   #define _FILE_OFFSET_BITS 64
6   #define _LARGEFILE64_SOURCE
7
8   #include <stdio.h>
9   #include <stdlib.h>
10  #include <unistd.h>
11  #include <sys/types.h>
12  #include <sys/stat.h>
13  #include <fcntl.h>
14  #include <string.h>
15  #include <linux/unistd.h>
16  #include <errno.h>
17  #include "fs.h"
18  #include "ext2_fs.h"
19
20  int _llseek(unsigned int fd, unsigned long offset_high,
21              unsigned long offset_low, loff_t *result,
22              unsigned int whence);
23
24  #define NR_LLSEEK 140  /* system call number of llseek system call */
25
26  #define DEBUG 0
27  #define DEBUG_BLOCK_ACCESS 0
28
29  #define BLOCKSIZE  4096
30  #define ADDRSIZE   4  /* disk address size */
31  #define NUM_ADDR_PER_BLOCK (BLOCKSIZE/ADDRSIZE)
32
33  #define MAX_DIRNAME 256
34  #define MAX_FILENAME 256
35  #define MAX_PATHNAME 1024
36  #define MAX_DIRS  16
37
38  #define SUPER_BLOCK       0
39  #define SUPER_BLOCK_OFFSET 1024
40  #define GROUP0_GDT_BLOCK1 1
41
42  #define HEX_DUMP  0
43  #define ASCI_DUMP 1
44
45  #define NO_INODE_NUMBER -1
46
47  struct fs_data {
48          struct ext2_super_block *sb;
49          struct ext2_group_desc  *groups;
50          int group_count;
51          int inodes_per_block;
52          int inodeblocks_per_group;
53          int fd;                     /* device file desc */
54  };
55
56  void
```

```
57  get_block (int fd,  unsigned long bnum, unsigned char *block)
58  {
59          unsigned long  n;
60          unsigned long high32;
61          unsigned long low32;
62
63
64          if (DEBUG_BLOCK_ACCESS)
65                  printf ("trying to retrieve block %lu\n", bnum);
66
67          high32 = bnum >> 20;
68          low32 = bnum & 0x000FFFFF;
69          low32 = low32 * BLOCKSIZE;
70
71          syscall(NR_LLSEEK, fd,  high32, low32, NULL, SEEK_SET);
72
73          n = read (fd, block, BLOCKSIZE);
74          if (n  !=  BLOCKSIZE) {
75                  printf ("can not read block\n");
76                  exit (1);
77          }
78
79          if (DEBUG_BLOCK_ACCESS)
80                  printf ("retrieved block %lu\n", bnum);
81  }
82
83
84  void
85  dump_block (unsigned char *block)
86  {
87          int i;
88
89          for (i = 0; i < BLOCKSIZE; ++i) {
90                  printf ("%02x", block[i]);
91                  if ( ( (i + 1) % 4) == 0)
92                          printf (" ");
93                  if ( ( (i+1) % 32) == 0)
94                          printf ("\n");
95          }
96          printf ("\n");
97  }
98
99
100 void
101 print_group_table (struct fs_data *fs)
102 {
103         int i;
104
105         printf ("size of group desc structure  = %u\n",
106                 sizeof (struct ext2_group_desc));
107
108         for (i = 0;  i < fs->group_count; ++i) {
109                 printf ("group %d: ", i);
110                 printf ("bitmap_block#=%u ", fs->groups[i].bg_block_bitmap);
111                 printf ("inodetable_block#=%u ", fs->groups[i].bg_inode_table);
112                 printf ("free_block_count=%u ",
```

```
113                         fs->groups[i].bg_free_blocks_count);
114                 printf ("\n");
115         }
116 }
117
118
119 void
120 print_inode_bitmap (struct fs_data *fs,
121                     struct ext2_group_desc *g,
122                     struct ext2_super_block *s)
123 {
124         unsigned char block[BLOCKSIZE];
125         int i, count;
126
127         get_block (fs->fd,  g->bg_inode_bitmap, block);
128         count = fs->sb->s_inodes_per_group / 8;
129         for (i = 0; i < count; ++i) {
130                 printf ("%02x", block[i]);
131                 if ( ( (i+1) % 32) == 0)
132                         printf ("\n");
133         }
134         printf ("\n");
135 }
136
137
138 void
139 print_inode (struct ext2_inode *inode, long inum)
140 {
141         int i;
142         int size, limit;
143
144         if (inode->i_size == 0)
145                 size = 0;
146         else
147                 size = 1 + ( (inode->i_size - 1) / BLOCKSIZE );
148
149         if (inum != NO_INODE_NUMBER)
150                 printf ("inode_number=%lu\n", inum);
151
152         printf("size_in_bytes=%u\n", inode->i_size);
153         printf("uid=%u\n", inode->i_uid);
154         printf("links_count=%u\n", inode->i_links_count);
155         printf("blocks_count=%u\n", inode->i_blocks);
156         printf("mode=%x\n", inode->i_mode);
157         printf("size_in_blocks=%d\n", size);
158
159
160         if (size <= 0)
161                 limit = 0;
162         else if (size <= EXT2_NDIR_BLOCKS)
163                 limit = size;
164         else if (size <= (EXT2_NDIR_BLOCKS + NUM_ADDR_PER_BLOCK))
165                 limit = EXT2_NDIR_BLOCKS + 1;
166         else if (size <= (EXT2_NDIR_BLOCKS  +
167                           NUM_ADDR_PER_BLOCK +
168                           (NUM_ADDR_PER_BLOCK *
```

```
169                             NUM_ADDR_PER_BLOCK)))
170                 limit = EXT2_NDIR_BLOCKS + 2;
171         else
172                 limit = EXT2_NDIR_BLOCKS + 3;
173
174         for (i = 0; i < limit; ++i)
175                 printf("i_block[%d]=%u\n", i, inode->i_block[i]);
176
177         printf ("\n");
178 }
179
180 void
181 print_inodes_in_block (struct fs_data *fs, unsigned char *block)
182 {
183         int i;
184         unsigned char *p;
185
186         p = block;
187         for (i = 0; i < fs->inodes_per_block; ++i) {
188                 print_inode ( (struct ext2_inode *)p, NO_INODE_NUMBER);
189                 p = p + fs->sb->s_inode_size;
190         }
191 }
192
193
194 void
195 print_inodetable_in_group (struct fs_data *fs, struct ext2_group_desc *g)
196 {
197         int k;
198         unsigned char block[BLOCKSIZE];
199
200         for (k = 0; k < fs->inodeblocks_per_group; ++k) {
201                 get_block (fs->fd,  g->bg_inode_table + k, block);
202                 print_inodes_in_block (fs, block);
203         }
204
205 }
206
207
208 void
209 print_dir_entry (struct ext2_dir_entry_2  *dep)
210 {
211
212         char entry_name[MAX_DIRNAME];
213
214         if (dep == NULL)
215                 return;
216
217         strncpy (entry_name, dep->name, dep->name_len);
218         entry_name[dep->name_len] = '\0';
219
220         printf ("type=%d inode=%-10u rec_len=%-5d name_len=%-5d name = %s\n",
221                 dep->file_type,
222                 dep->inode,
223                 dep->rec_len,
224                 dep->name_len,
```

```
225                 entry_name);
226 }
227
228
229 int
230 get_inode (struct fs_data *fs, int inum, struct ext2_inode *inode)
231 {
232         int group;
233         unsigned int bnum;
234         unsigned char block[BLOCKSIZE];
235         struct ext2_inode *p;
236         int n;
237         int inode_in_group, block_in_group, inode_in_block;
238
239         group = (inum - 1)  / fs->sb->s_inodes_per_group;
240         inode_in_group = (inum - 1) % fs->sb->s_inodes_per_group;
241         block_in_group = inode_in_group / fs->inodes_per_block;
242         inode_in_block = inode_in_group % fs->inodes_per_block;
243
244         bnum = fs->groups[group].bg_inode_table + block_in_group;
245
246         get_block (fs->fd, bnum, block);
247
248         p = (struct ext2_inode*)
249                 (block + (inode_in_block * fs->sb->s_inode_size));
250         if (fs->sb->s_inode_size  <= sizeof (struct ext2_inode))
251                 n = fs->sb->s_inode_size;
252         else
253                 n = sizeof (struct ext2_inode);
254
255         memcpy ( (void*) inode, (void *) p, n);
256         return (0);
257 }
258
259
260 int
261 parse_pathname (char path[], char names[MAX_DIRS][MAX_DIRNAME])
262 {
263         int i, k;
264         int len;
265         char astr[MAX_DIRNAME];
266         int n;
267
268         i = 0;
269         n = 0;
270         len = strlen (path);
271
272
273         if (len <=0 )
274                 return -1;
275
276         if (path[0] != '/')
277                 return -1;
278
279         strcpy (names[n], "/");
280
```

```
281         n++;
282         i = 1;
283         while (i < len) {
284                 k = 0;
285                 while (path[i] != '/') {
286                         astr[k] = path[i];
287                         k++;
288                         i++;
289                         if (i == len)
290                                 break;
291                 }
292                 if (k == 0)
293                         return -1;
294
295                 astr[k] = '\0';
296                 strcpy (names[n], astr);
297                 n++;
298                 i++;
299         }
300
301         if (DEBUG)
302                 for (i = 0; i < n; ++i)
303                         printf ("parsed_dirname%d=%s\n", i, names[i]);
304
305         return (n);
306 }
307
308
309 int
310 get_data_block (struct fs_data *fs,
311           struct ext2_inode *inode,
312           int n, /* requested file data block */
313           unsigned char *block)
314 {
315         unsigned char ind_block[BLOCKSIZE];
316         unsigned char dind_block[BLOCKSIZE];
317         unsigned char tind_block[BLOCKSIZE];
318         unsigned int size; /* size of file in blocks */
319         unsigned int *p1, *p2, *p3;
320         unsigned int div2, rem2, div3, rem3;
321
322
323         if (inode->i_size == 0)
324                 size = 0;
325         else
326                 size = 1 + ( (inode->i_size - 1) / BLOCKSIZE );
327
328         if (size  == 0)
329                 return -1;
330
331         if ( (n < 0)  || (n >= size))
332              return -1;
333
334         if (n < EXT2_NDIR_BLOCKS)
335         {
336                 get_block (fs->fd, inode->i_block[n], block);
```

```
337                 return 0;
338         }
339
340
341         if (n < (EXT2_NDIR_BLOCKS +
342                  NUM_ADDR_PER_BLOCK))
343         {
344                 get_block (fs->fd, inode->i_block[EXT2_IND_BLOCK], ind_block);
345                 p1 = (unsigned int *) &ind_block;
346                 get_block (fs->fd, p1[n - EXT2_NDIR_BLOCKS], block);
347                 return 0;
348         }
349
350
351         if (n < (EXT2_NDIR_BLOCKS +
352                  NUM_ADDR_PER_BLOCK +
353                  (NUM_ADDR_PER_BLOCK * NUM_ADDR_PER_BLOCK)))
354         {
355                 get_block (fs->fd, inode->i_block[EXT2_DIND_BLOCK], dind_block);
356                 p2 = (unsigned int *) &dind_block;
357                 div2 = (n - EXT2_NDIR_BLOCKS - NUM_ADDR_PER_BLOCK) /
358                         NUM_ADDR_PER_BLOCK;
359                 rem2 = (n - EXT2_NDIR_BLOCKS - NUM_ADDR_PER_BLOCK) %
360                         NUM_ADDR_PER_BLOCK;
361                 get_block (fs->fd, p2[div2], ind_block);
362                 p1 = (unsigned int *) &ind_block;
363                 get_block (fs->fd, p1[rem2], block);
364                 return 0;
365         }
366
367
368         if (n < (EXT2_NDIR_BLOCKS +
369                  NUM_ADDR_PER_BLOCK +
370                  (NUM_ADDR_PER_BLOCK * NUM_ADDR_PER_BLOCK) +
371                  (NUM_ADDR_PER_BLOCK *
372                   NUM_ADDR_PER_BLOCK *
373                   NUM_ADDR_PER_BLOCK)))
374         {
375
376                 get_block (fs->fd, inode->i_block[EXT2_TIND_BLOCK], tind_block);
377                 p3 = (unsigned int *) &tind_block;
378                 div3 = (n - EXT2_NDIR_BLOCKS - NUM_ADDR_PER_BLOCK -
379                         (NUM_ADDR_PER_BLOCK * NUM_ADDR_PER_BLOCK)) /
380                         (NUM_ADDR_PER_BLOCK * NUM_ADDR_PER_BLOCK);
381                 rem3 = (n - EXT2_NDIR_BLOCKS - NUM_ADDR_PER_BLOCK -
382                         (NUM_ADDR_PER_BLOCK * NUM_ADDR_PER_BLOCK)) %
383                         (NUM_ADDR_PER_BLOCK * NUM_ADDR_PER_BLOCK);
384
385                 get_block (fs->fd, p3[div3], dind_block);
386                 p2 = (unsigned int *) &dind_block;
387                 div2 = rem3 / NUM_ADDR_PER_BLOCK;
388                 rem2 = rem3 % NUM_ADDR_PER_BLOCK;
389
390                 get_block (fs->fd, p2[div2], ind_block);
391                 p1 = (unsigned int *) &ind_block;
392                 get_block (fs->fd, p1[rem2], block);
```

```
393
394                 return 0;
395         }
396
397         return -1;
398 }
399
400
401 void
402 print_blockinfo (struct fs_data *fs, unsigned int fileb, unsigned int diskb)
403 {
404         unsigned int group;
405
406         group = diskb / fs->sb->s_blocks_per_group;
407
408         printf ("file_block=%-5u disk_block=%-10u group=%-10u\n",
409                 fileb, diskb, group);
410 }
411
412
413 void
414 access_data_block_numbers  (struct fs_data *fs,
415                             struct ext2_inode *inode)
416 {
417
418         unsigned char ind_block[BLOCKSIZE];
419         unsigned char dind_block[BLOCKSIZE];
420         unsigned char tind_block[BLOCKSIZE];
421         unsigned int i, j, k;
422         unsigned int count, size;
423         unsigned int *p, *p2, *p3;
424
425
426         if (inode->i_size == 0)
427                 size = 0;
428         else
429                 size = 1 + ( (inode->i_size - 1) / BLOCKSIZE );
430
431         count = 0;
432         if (count >= size)
433                 return;
434
435         for (i = 0; i < EXT2_NDIR_BLOCKS; ++i) {
436                 print_blockinfo (fs, count, inode->i_block[i]);
437                 count++;
438                 if ( count >= size)
439                      return;
440         }
441
442         get_block (fs->fd, inode->i_block[EXT2_IND_BLOCK], ind_block);
443         p = (unsigned int *) &ind_block;
444         for (i = 0; i < NUM_ADDR_PER_BLOCK; ++i) {
445                 print_blockinfo (fs, count, p[i]);
446                 count++;
447                 if (count >= size)
448                         return;
```

```
449         }
450
451         get_block (fs->fd, inode->i_block[EXT2_DIND_BLOCK], dind_block);
452         p2 = (unsigned int *) &dind_block;
453         for (i = 0; i < NUM_ADDR_PER_BLOCK; ++i) {
454                 get_block (fs->fd, p2[i], ind_block);
455                 p = (unsigned int *) &ind_block;
456                 for (j = 0; j < NUM_ADDR_PER_BLOCK; ++j) {
457                         print_blockinfo (fs, count, p[j]);
458                         count++;
459                         if (count >= size)
460                                 return;
461                 }
462         }
463
464         get_block (fs->fd, inode->i_block[EXT2_TIND_BLOCK], tind_block);
465         p3 = (unsigned int *) &tind_block;
466         for (i = 0; i < NUM_ADDR_PER_BLOCK; ++i) {
467                 get_block (fs->fd, p3[i], dind_block);
468                 p2 = (unsigned int *) &dind_block;
469                 for (j = 0; j < NUM_ADDR_PER_BLOCK; ++j) {
470                         get_block (fs->fd, p2[j], ind_block);
471                         p = (unsigned int *) &ind_block;
472                         for (k = 0; k < NUM_ADDR_PER_BLOCK; ++k) {
473                                 print_blockinfo (fs, count, p[k]);
474                                 count++;
475                                 if (count >= size)
476                                         return;
477                         }
478                 }
479
480         }
481
482         return;
483 }
484
485
486
487
488 void
489 access_data_blocks (struct fs_data *fs, struct ext2_inode *inode)
490 {
491         unsigned char block[BLOCKSIZE];
492         unsigned int i, size;
493         int ret;
494
495
496
497         if (inode->i_size == 0)
498                 size = 0;
499         else
500                 size = 1 + ( (inode->i_size - 1) / BLOCKSIZE );
501
502         for (i = 0; i < size; ++i) {
503                 ret = get_data_block (fs, inode, i, block);
504                 if (ret == 0)
```

```
505                         printf ("retrieved file block %d\n", i);
506         }
507 }
508
509
510 void
511 dump_region (unsigned char *buf, int len, int type)
512 {
513         static unsigned long count  = 0;
514         int i;
515
516         for (i = 0; i < len; ++i) {
517                 if (type == HEX_DUMP) {
518                         printf ("%02x", buf[i]);
519                         fflush (stdout);
520                         if ( ( (count + 1) % 4) == 0)
521                                 printf (" ");
522                         if ( ( (count + 1) % 32) == 0)
523                                 printf ("\n");
524                         count++;
525                 }
526                 else if (type == ASCI_DUMP) {
527                         printf ("%c", (char) buf[i]);
528                         fflush (stdout);
529                         count++;
530                 }
531         }
532 }
533
534 /*
535    assuming file size if less than 4 GB
536 */
537 void
538 access_data_region (struct fs_data *fs, struct ext2_inode *inode,
539                     unsigned long start, unsigned long length,
540                     int type)
541 {
542         unsigned char block[BLOCKSIZE];
543         unsigned long first,  last, first_off, last_off;
544         int ret;
545         int i;
546
547         if (length == 0)
548                 return;
549
550         if ( (start + length) > inode->i_size)
551                 return;
552
553         first = start / BLOCKSIZE;
554         first_off = start % BLOCKSIZE;
555         last = (start + length - 1) / BLOCKSIZE;
556         last_off = (start + length - 1) % BLOCKSIZE;
557
558
559         if (DEBUG) {
560                 printf ("first = %lu\n", first);
```

```
561                 printf ("last  = %lu\n", last);
562                 printf ("first_off = %lu\n", first_off);
563                 printf ("last_off = %lu\n", last_off);
564         }
565
566         for (i = first; i <= last; ++i) {
567                 ret = get_data_block (fs, inode, i, block);
568                 if (ret == -1) {
569                         printf ("can not get data block %d\n", i);
570                         exit (1);
571                 }
572                 if ((i == first) && (i == last))
573                         dump_region (block + first_off, last_off - first_off + 1, type);
574                 else if (i == first)
575                         dump_region (block + first_off, BLOCKSIZE - first_off, type);
576                 else if (i == last)
577                         dump_region (block, last_off + 1, type);
578                 else
579                         dump_region (block, BLOCKSIZE, type);
580         }
581         printf ("\n");
582 }
583
584
585
586
587 int find_dir_entry(struct fs_data *fs, struct ext2_inode *dir_inode, char *name,
588                 struct ext2_dir_entry_2  *diren)
589 {
590         struct ext2_dir_entry_2  *dep;
591         int i, count;
592         char entry_name[MAX_DIRNAME];
593         unsigned char block[BLOCKSIZE];
594         int logical;
595
596
597         logical  = 0;
598         get_data_block (fs, dir_inode, logical, block);
599         i = 0;
600         count = 0;
601         while (1) {
602                 dep = (struct  ext2_dir_entry_2*) (block + i);
603
604                 strncpy (entry_name, dep->name, dep->name_len);
605                 entry_name[dep->name_len] = '\0';
606
607
608                 if (strcmp (entry_name, name)  == 0) {
609                         memcpy ( (void *) diren, (void *) dep,
610                                  sizeof (struct   ext2_dir_entry_2));
611                         return (0);
612
613                 }
614                 i += dep->rec_len;
615                 count += dep->rec_len;
616
```

```
617                 if (count >= dir_inode->i_size)
618                         break;
619
620                 if (i >= BLOCKSIZE) {
621                         i = i % BLOCKSIZE;
622                         logical++;
623                         get_data_block (fs, dir_inode, logical, block);
624                 }
625         }
626         return (-1);
627 }
628
629
630
631 void
632 print_dir (struct fs_data *fs, struct ext2_inode *dir_inode)
633 {
634
635         struct ext2_dir_entry_2  *dep;
636         int i, count;
637         char entry_name[MAX_DIRNAME];
638         unsigned char block[BLOCKSIZE];
639         int logical;
640
641         logical = 0;
642         get_data_block (fs, dir_inode, 0, block);
643         i = 0;
644         count = 0;
645         while (1) {
646                 dep = (struct  ext2_dir_entry_2*) (block + i);
647
648                 strncpy (entry_name, dep->name, dep->name_len);
649                 entry_name[dep->name_len] = '\0';
650
651                 i += dep->rec_len;
652                 count += dep->rec_len;
653
654                 print_dir_entry (dep);
655
656                 if (count >= dir_inode->i_size)
657                         break;
658
659                 if (i >= BLOCKSIZE) {
660                         i = i % BLOCKSIZE;
661                         logical++;
662                         get_data_block (fs, dir_inode, logical, block);
663                 }
664         }
665 }
666
667
668
669 void
670 print_superblock (struct ext2_super_block *sb)
671 {
672         printf ("inode_count=%u\n", sb->s_inodes_count);
```

```
673         printf ("block_count=%u\n", sb->s_blocks_count);
674         printf ("first_data_block=%u\n", sb->s_first_data_block);
675         printf ("magic_number=%x\n", (unsigned short) sb->s_magic);
676         printf ("inode_size=%d\n", sb->s_inode_size);
677         printf ("inodes_per_group=%d\n", sb->s_inodes_per_group);
678         printf ("blocks_per_group=%d\n", sb->s_blocks_per_group);
679         printf ("log_block_size=%d\n", sb->s_log_block_size);
680 }
681
682
683 struct fs_data *
684 create_fill_fsdata (char *devicename)
685 {
686         struct fs_data *fs;
687         unsigned char block[BLOCKSIZE];
688         int i, j, gd_per_block;
689         int blk_count;
690         struct ext2_group_desc *g;
691
692
693         fs = (struct fs_data *) malloc (sizeof (struct fs_data));
694         if (!fs) {
695                 printf ("can not malloc \n");
696                 exit (1);
697         }
698
699         fs->sb = (struct ext2_super_block *) malloc (sizeof (struct ext2_super_block));
700         if (!fs->sb) {
701                 printf ("can not malloc\n");
702                 exit (1);
703         }
704
705
706         fs->fd = open (devicename, O_RDONLY);
707         if (fs->fd < 0) {
708                 printf ("can not open device file\n");
709                 exit(1);
710         }
711
712         get_block (fs->fd, SUPER_BLOCK, block);
713
714         memcpy ( (void*)fs->sb,
715                  (void*)(block + SUPER_BLOCK_OFFSET),
716                  sizeof (struct ext2_super_block));
717
718         if (DEBUG)
719                 print_superblock (fs->sb);
720
721         fs->group_count = 1 +
722                 ((fs->sb->s_blocks_count - 1 ) / fs->sb->s_blocks_per_group);
723         fs->inodes_per_block = BLOCKSIZE / fs->sb->s_inode_size;
724         fs->inodeblocks_per_group =
725                 fs->sb->s_inodes_per_group  /  fs->inodes_per_block;
726
727         if (DEBUG) {
728                 printf ("\n");
```

```
729                 printf ("blocksize=%d\n", BLOCKSIZE);
730                 printf ("group_count=%d\n", fs->group_count);
731                 printf ("inodes_per_block=%d\n", fs->inodes_per_block);
732                 printf ("inodeblocks_per_group=%d\n", fs->inodeblocks_per_group);
733                 printf ("\n");
734         }
735
736         fs->groups = (struct ext2_group_desc *)
737                 malloc (fs->group_count * sizeof (struct ext2_group_desc));
738
739         gd_per_block  =  BLOCKSIZE / sizeof (struct ext2_group_desc);
740         blk_count = 1 + ((fs->group_count - 1) / gd_per_block);
741
742         for (i = 0; i < blk_count-1; ++i) {
743                 get_block (fs->fd, GROUP0_GDT_BLOCK1 + i, block);
744                 g = (struct ext2_group_desc *) block;
745                 for (j = 0; j < gd_per_block; ++j) {
746                         memcpy ( (void *) &(fs->groups[i * gd_per_block + j]),
747                                 (void *) &g[j],
748                                 sizeof (struct ext2_group_desc));
749                 }
750         }
751         /* read last block of gdt */
752         get_block (fs->fd, GROUP0_GDT_BLOCK1 + i, block);
753         g = (struct ext2_group_desc *) block;
754         for (j = 0; j < (fs->group_count % gd_per_block);  ++j) {
755                 memcpy ( (void *) &(fs->groups[i * gd_per_block + j]),
756                          (void *) &g[j],
757                          sizeof (struct ext2_group_desc));
758         }
759
760         if (DEBUG)
761                 print_group_table(fs);
762
763         return (fs);
764 }
765
766
767 void
768 close_destroy_fsdata (struct fs_data *fs)
769 {
770
771         close (fs->fd);
772
773         free (fs->sb);
774         free (fs->groups);
775         free (fs);
776 }
777
778
779 /*
780    pathname must be an absolute pathname
781 */
782 int
783 path_to_inode (struct fs_data *fs, char *path,
784                struct ext2_inode *inode,
```

```
785                struct ext2_dir_entry_2 *diren)
786 {
787         struct ext2_dir_entry_2 dep;
788         struct ext2_inode nod;
789         char names[MAX_DIRS][MAX_DIRNAME];
790         int name_count = 0;
791         int i;
792         int ret;
793 
794         name_count = parse_pathname (path, names);
795         if (name_count == -1) {
796                 printf ("parse error for pathname\n");
797                 exit (1);
798         }
799 
800         /*
801            create a directory entry for root dir
802         */
803         dep.file_type = EXT2_FT_DIR;
804         dep.inode = EXT2_ROOT_INO;
805         dep.name_len = 1;
806         dep.name[0] = '/';
807         dep.rec_len =  EXT2_DIR_REC_LEN(dep.name_len);
808 
809         get_inode (fs, dep.inode, &nod);
810 
811         i = 1;
812         while ((i < name_count) && (dep.file_type == EXT2_FT_DIR)  ) {
813                 ret = find_dir_entry (fs, &nod, names[i], &dep);
814                 if (ret == -1) {
815                         printf ("can not find dir entry %s\n", names[i]);
816                         return (-1);
817                 }
818 
819                 get_inode (fs, dep.inode, &nod);
820                 i++;
821         }
822 
823         memcpy ( (void *)inode, (void*) &nod, sizeof (struct  ext2_inode));
824         memcpy ( (void *) diren, (void*) &dep, sizeof (struct  ext2_dir_entry_2));
825 
826         return (dep.inode);
827 }
828 
829 
830 void
831 print_path_directories (struct fs_data *fs, char *path)
832 {
833         struct ext2_dir_entry_2 dep;
834         struct ext2_inode nod;
835         char names[MAX_DIRS][MAX_DIRNAME];
836         int name_count = 0;
837         int i;
838         int ret;
839 
840         name_count = parse_pathname (path, names);
```

```
841         if (name_count == -1) {
842                 printf ("parse error for pathname\n");
843                 exit (1);
844         }
845
846         path_to_inode (fs, "/", &nod, &dep);
847
848         printf ("directory: %s\n", names[0]);
849         print_dir (fs, &nod);
850         printf ("\n");
851
852         i = 1;
853         while ((i < name_count) && (dep.file_type == EXT2_FT_DIR)  ) {
854                 ret = find_dir_entry (fs, &nod, names[i], &dep);
855                 if (ret == -1) {
856                         printf ("can not find dir entry %s\n", names[i]);
857                         exit (1);
858                 }
859
860                 get_inode (fs, dep.inode, &nod);
861                 if (dep.file_type == EXT2_FT_DIR) {
862                         printf ("directory: %s\n", names[i]);
863                         print_dir (fs, &nod);
864                         printf ("\n");
865
866                 }
867                 i++;
868         }
869 }
870
871
872 void
873 print_help ()
874 {
875         printf ("usage:\n");
876         printf ("findblocks <devicefilename>\n");
877         printf ("     -s : display superblock info \n");
878         printf ("     -g : display groups info\n");
879         printf ("     -i <pathname> : display inode info for file/dir <pathname> \n");
880         printf ("     -dumpblock <blocknum> -x : dump disk block <blocknum> in hex \n");
881         printf ("     -dumpblock <blocknum> -t : dump disk block <blocknum> in ascii \n");
882         printf ("     -d <pathname> : display directory content for directory <pathname> \n");
883         printf ("     -dt <pathname> : traverse directories in <pathname> listing content \n");
884         printf ("     -de <pathname> : display the directory entry for <pathname> \n");
885         printf ("     -blocks <pathname> : display disk block numbers of file/dir <pathname> \n");
886         printf ("     -data <pathname> -x : dump content of file/dir <pathname> in hex \n");
887         printf ("     -data <pathname> -t : dump content of file/dir <pathname> in ascii \n");
888         printf ("     -data <pathname> -x <start> <length> : dump file region content  in hex \n");
889         printf ("     -data <pathname> -t <start> <length> : dump file region content in ascii \n");
890 }
891
892
893 void
894 do_abs_path (char *pathname, char *p)
895 {
896         int end;
```

```
897
898         if (p[0] != '/') {
899                 getcwd (pathname, MAX_PATHNAME);
900                 end = strlen (pathname);
901                 pathname[end] = '/';
902                 strcpy (pathname + end + 1, p);
903         }
904         else {
905                 strcpy (pathname, p);
906         }
907 }
908
909 void
910 do_command (struct fs_data *fs, int argc, char **argv, char *env[])
911 {
912         char pathname[MAX_PATHNAME];
913         unsigned char block[BLOCKSIZE];
914         struct ext2_inode inode;
915         int ret;
916         unsigned long start;
917         unsigned long length;
918         char command[128];
919         struct ext2_dir_entry_2  diren;
920         unsigned int blocknum;
921
922
923         strcpy (command, argv[2]);
924
925         if (strcmp (command, "-s") == 0) {
926                 print_superblock (fs->sb);
927         }
928         else if (strcmp (command, "-g") == 0) {
929                 print_group_table (fs);
930         }
931         else if (strcmp (command, "-i") == 0) {
932                 do_abs_path (pathname, argv[3]);
933                 ret = path_to_inode (fs, pathname, &inode, &diren);
934                 if (ret == -1) {
935                         printf ("could not find inode for %s\n", pathname);
936                         exit (1);
937                 }
938                 print_inode (&inode, ret);
939         }
940         else if (strcmp (command, "-dumpblock") == 0) {
941                 blocknum = atoi (argv[3]);
942                 get_block (fs->fd, blocknum, block);
943                 if ( strcmp(argv[4], "-x") == 0)
944                         dump_region (block, BLOCKSIZE, HEX_DUMP);
945                 else if ( strcmp(argv[4], "-t") == 0)
946                         dump_region (block, BLOCKSIZE, ASCI_DUMP);
947         }
948         else if (strcmp (command, "-d") == 0) {
949                 do_abs_path (pathname, argv[3]);
950                 ret = path_to_inode (fs, pathname, &inode, &diren);
951                 if (diren.file_type != EXT2_FT_DIR) {
952                         printf ("is not directory\n");
```

```
953                     exit (1);
954             }
955             else
956                     print_dir (fs, &inode);
957     }
958     else if (strcmp (command, "-dt") == 0) {
959             do_abs_path (pathname, argv[3]);
960             print_path_directories (fs, pathname);
961     }
962     else if (strcmp (command, "-de") == 0) {
963             do_abs_path (pathname, argv[3]);
964             ret = path_to_inode (fs, pathname, &inode, &diren);
965             if (ret == -1) {
966                     printf ("could not find inode for %s\n", pathname);
967                     exit (1);
968             }
969             print_dir_entry (&diren);
970     }
971     else if (strcmp (command, "-blocks") == 0) {
972             do_abs_path (pathname, argv[3]);
973             ret = path_to_inode (fs, pathname, &inode, &diren);
974             if (ret == -1) {
975                     printf ("could not find inode for %s\n", pathname);
976                     exit (1);
977             }
978             access_data_block_numbers (fs, &inode);
979     }
980     else if (strcmp (command, "-data") == 0) {
981             do_abs_path (pathname, argv[3]);
982             ret = path_to_inode (fs, pathname, &inode, &diren);
983             if (ret == -1) {
984                     printf ("could not find inode for %s\n", pathname);
985                     exit (1);
986             }
987             if (strcmp(argv[4], "-x") == 0) {
988                     if (argc == 5) {
989                             access_data_region (fs, &inode, 0,
990                                                 inode.i_size, HEX_DUMP);
991                     }
992                     else {
993                             start = atoi (argv[5]);
994                             length = atoi (argv[6]);
995                             access_data_region (fs,
996                                                 &inode,
997                                                 start,
998                                                 length, HEX_DUMP);
999                     }
1000            }
1001            else if (strcmp(argv[4], "-t") == 0) {
1002                    if (argc == 5) {
1003                            access_data_region (fs, &inode,
1004                                                0, inode.i_size,
1005                                                ASCI_DUMP);
1006                    }
1007                    else {
1008                            start = atoi (argv[5]);
```

```
                                length = atoi (argv[6]);
                                access_data_region (fs,
                                                    &inode,
                                                    start,
                                                    length,
                                                    ASCI_DUMP);
                        }
                }
                else {
                        printf ("invalid option \n");
                        print_help();

                        exit (1);
                }
        }
        else {
                printf ("invalid command\n");
                exit (1);
        }
}


int
main (int argc, char **argv, char **environ)
{
        struct fs_data *fsdata;
        char devname[MAX_PATHNAME];


        if (argc < 2) {
                printf ("no device file specified\n");
                print_help();
                exit (1);
        }

        strcpy (devname, argv[1]);

        fsdata = create_fill_fsdata (devname);

        do_command (fsdata, argc, argv, environ);

        close_destroy_fsdata (fsdata);

        return 0;
}
```

The program includes two header files `fs.h` and `ext2_fs.h`. Below is the file fs.h:

```
#ifndef _LINUX_FS_H
#define _LINUX_FS_H

/*
 * This file has definitions for some important file table
 * structures etc.
```

```
 */

#include <linux/limits.h>
#include <linux/ioctl.h>

/*
 * It's silly to have NR_OPEN bigger than NR_FILE, but you can change
 * the file limit at runtime and only root can increase the per-process
 * nr_file rlimit, so it's safe to set up a ridiculously high absolute
 * upper limit on files-per-process.
 *
 * Some programs (notably those using select()) may have to be
 * recompiled to take full advantage of the new limits..
 */

/* Fixed constants first: */
#undef NR_OPEN
#define INR_OPEN 1024               /* Initial setting for nfile rlimits */

#define BLOCK_SIZE_BITS 10
#define BLOCK_SIZE (1<<BLOCK_SIZE_BITS)

#define SEEK_SET        0       /* seek relative to beginning of file */
#define SEEK_CUR        1       /* seek relative to current file position */
#define SEEK_END        2       /* seek relative to end of file */
#define SEEK_MAX        SEEK_END

/* And dynamically-tunable limits and defaults: */
struct files_stat_struct {
        int nr_files;               /* read only */
        int nr_free_files;        /* read only */
        int max_files;               /* tunable */
};

struct inodes_stat_t {
        int nr_inodes;
        int nr_unused;
        int dummy[5];               /* padding for sysctl ABI compatibility */
};


#define NR_FILE  8192        /* this can well be larger on a larger system */

#define MAY_EXEC 1
#define MAY_WRITE 2
#define MAY_READ 4
#define MAY_APPEND 8
#define MAY_ACCESS 16
#define MAY_OPEN 32

/*
 * flags in file.f_mode.  Note that FMODE_READ and FMODE_WRITE must correspond
 * to O_WRONLY and O_RDWR via the strange trick in __dentry_open()
 */

/* file is open for reading */
```

```
63  #define FMODE_READ               ((fmode_t)1)
64  /* file is open for writing */
65  #define FMODE_WRITE               ((fmode_t)2)
66  /* file is seekable */
67  #define FMODE_LSEEK               ((fmode_t)4)
68  /* file can be accessed using pread */
69  #define FMODE_PREAD               ((fmode_t)8)
70  /* file can be accessed using pwrite */
71  #define FMODE_PWRITE               ((fmode_t)16)
72  /* File is opened for execution with sys_execve / sys_uselib */
73  #define FMODE_EXEC               ((fmode_t)32)
74  /* File is opened with O_NDELAY (only set for block devices) */
75  #define FMODE_NDELAY               ((fmode_t)64)
76  /* File is opened with O_EXCL (only set for block devices) */
77  #define FMODE_EXCL               ((fmode_t)128)
78  /* File is opened using open(.., 3, ..) and is writeable only for ioctls
79     (specialy hack for floppy.c) */
80  #define FMODE_WRITE_IOCTL        ((fmode_t)256)
81
82  /*
83   * Don't update ctime and mtime.
84   *
85   * Currently a special hack for the XFS open_by_handle ioctl, but we'll
86   * hopefully graduate it to a proper O_CMTIME flag supported by open(2) soon.
87   */
88  #define FMODE_NOCMTIME               ((fmode_t)2048)
89
90  /*
91   * The below are the various read and write types that we support. Some of
92   * them include behavioral modifiers that send information down to the
93   * block layer and IO scheduler. Terminology:
94   *
95   *        The block layer uses device plugging to defer IO a little bit, in
96   *        the hope that we will see more IO very shortly. This increases
97   *        coalescing of adjacent IO and thus reduces the number of IOs we
98   *        have to send to the device. It also allows for better queuing,
99   *        if the IO isn't mergeable. If the caller is going to be waiting
100  *        for the IO, then he must ensure that the device is unplugged so
101  *        that the IO is dispatched to the driver.
102  *
103  *        All IO is handled async in Linux. This is fine for background
104  *        writes, but for reads or writes that someone waits for completion
105  *        on, we want to notify the block layer and IO scheduler so that they
106  *        know about it. That allows them to make better scheduling
107  *        decisions. So when the below references 'sync' and 'async', it
108  *        is referencing this priority hint.
109  *
110  * With that in mind, the available types are:
111  *
112  * READ                        A normal read operation. Device will be plugged.
113  * READ_SYNC                A synchronous read. Device is not plugged, caller can
114  *                          immediately wait on this read without caring about
115  *                          unplugging.
116  * READA                Used for read-ahead operations. Lower priority, and the
117  *                          block layer could (in theory) choose to ignore this
118  *                          request if it runs into resource problems.
```

```
119  * WRITE                A normal async write. Device will be plugged.
120  * SWRITE                Like WRITE, but a special case for ll_rw_block() that
121  *                       tells it to lock the buffer first. Normally a buffer
122  *                       must be locked before doing IO.
123  * WRITE_SYNC_PLUG       Synchronous write. Identical to WRITE, but passes down
124  *                       the hint that someone will be waiting on this IO
125  *                       shortly. The device must still be unplugged explicitly,
126  *                       WRITE_SYNC_PLUG does not do this as we could be
127  *                       submitting more writes before we actually wait on any
128  *                       of them.
129  * WRITE_SYNC               Like WRITE_SYNC_PLUG, but also unplugs the device
130  *                       immediately after submission. The write equivalent
131  *                       of READ_SYNC.
132  * WRITE_ODIRECT        Special case write for O_DIRECT only.
133  * SWRITE_SYNC
134  * SWRITE_SYNC_PLUG        Like WRITE_SYNC/WRITE_SYNC_PLUG, but locks the buffer.
135  *                       See SWRITE.
136  * WRITE_BARRIER        Like WRITE, but tells the block layer that all
137  *                       previously submitted writes must be safely on storage
138  *                       before this one is started. Also guarantees that when
139  *                       this write is complete, it itself is also safely on
140  *                       storage. Prevents reordering of writes on both sides
141  *                       of this IO.
142  *
143  */
144 #define RW_MASK                 1
145 #define RWA_MASK        2
146 #define READ 0
147 #define WRITE 1
148 #define READA 2                 /* read-ahead  - don't block if no resources */
149 #define SWRITE 3        /* for ll_rw_block() - wait for buffer lock */
150 #define READ_SYNC        (READ | (1 << BIO_RW_SYNCIO) | (1 << BIO_RW_UNPLUG))
151 #define READ_META        (READ | (1 << BIO_RW_META))
152 #define WRITE_SYNC_PLUG         (WRITE | (1 << BIO_RW_SYNCIO) | (1 << BIO_RW_NOIDLE))
153 #define WRITE_SYNC        (WRITE_SYNC_PLUG | (1 << BIO_RW_UNPLUG))
154 #define WRITE_ODIRECT        (WRITE | (1 << BIO_RW_SYNCIO) | (1 << BIO_RW_UNPLUG))
155 #define SWRITE_SYNC_PLUG         \
156                         (SWRITE | (1 << BIO_RW_SYNCIO) | (1 << BIO_RW_NOIDLE))
157 #define SWRITE_SYNC        (SWRITE_SYNC_PLUG | (1 << BIO_RW_UNPLUG))
158 #define WRITE_BARRIER        (WRITE | (1 << BIO_RW_BARRIER))
159
160 /*
161  * These aren't really reads or writes, they pass down information about
162  * parts of device that are now unused by the file system.
163  */
164 #define DISCARD_NOBARRIER (1 << BIO_RW_DISCARD)
165 #define DISCARD_BARRIER ((1 << BIO_RW_DISCARD) | (1 << BIO_RW_BARRIER))
166
167 #define SEL_IN                1
168 #define SEL_OUT                2
169 #define SEL_EX                4
170
171 /* public flags for file_system_type */
172 #define FS_REQUIRES_DEV 1
173 #define FS_BINARY_MOUNTDATA 2
174 #define FS_HAS_SUBTYPE 4
```

```
175 #define FS_REVAL_DOT        16384        /* Check the paths ".", ".." for staleness */
176 #define FS_RENAME_DOES_D_MOVE        32768        /* FS will handle d_move()
177                                          * during rename() internally.
178                                          */
179
180 /*
181  * These are the fs-independent mount-flags: up to 32 flags are supported
182  */
183 #define MS_RDONLY         1        /* Mount read-only */
184 #define MS_NOSUID         2        /* Ignore suid and sgid bits */
185 #define MS_NODEV         4        /* Disallow access to device special files */
186 #define MS_NOEXEC         8        /* Disallow program execution */
187 #define MS_SYNCHRONOUS        16        /* Writes are synced at once */
188 #define MS_REMOUNT        32        /* Alter flags of a mounted FS */
189 #define MS_MANDLOCK        64        /* Allow mandatory locks on an FS */
190 #define MS_DIRSYNC        128        /* Directory modifications are synchronous */
191 #define MS_NOATIME        1024        /* Do not update access times. */
192 #define MS_NODIRATIME        2048        /* Do not update directory access times */
193 #define MS_BIND               4096
194 #define MS_MOVE               8192
195 #define MS_REC                16384
196 #define MS_VERBOSE        32768        /* War is peace. Verbosity is silence.
197                                     MS_VERBOSE is deprecated. */
198 #define MS_SILENT        32768
199 #define MS_POSIXACL        (1<<16)        /* VFS does not apply the umask */
200 #define MS_UNBINDABLE        (1<<17)        /* change to unbindable */
201 #define MS_PRIVATE        (1<<18)        /* change to private */
202 #define MS_SLAVE        (1<<19)        /* change to slave */
203 #define MS_SHARED        (1<<20)        /* change to shared */
204 #define MS_RELATIME        (1<<21)        /* Update atime relative to mtime/ctime. */
205 #define MS_KERNMOUNT        (1<<22) /* this is a kern_mount call */
206 #define MS_I_VERSION        (1<<23) /* Update inode I_version field */
207 #define MS_STRICTATIME        (1<<24) /* Always perform atime updates */
208 #define MS_ACTIVE        (1<<30)
209 #define MS_NOUSER        (1<<31)
210
211 /*
212  * Superblock flags that can be altered by MS_REMOUNT
213  */
214 #define MS_RMT_MASK        (MS_RDONLY|MS_SYNCHRONOUS|MS_MANDLOCK|MS_I_VERSION)
215
216 /*
217  * Old magic mount flag and mask
218  */
219 #define MS_MGC_VAL 0xC0ED0000
220 #define MS_MGC_MSK 0xffff0000
221
222 /* Inode flags - they have nothing to superblock flags now */
223
224 #define S_SYNC                1        /* Writes are synced at once */
225 #define S_NOATIME        2        /* Do not update access times */
226 #define S_APPEND        4        /* Append-only file */
227 #define S_IMMUTABLE        8        /* Immutable file */
228 #define S_DEAD                16        /* removed, but still open directory */
229 #define S_NOQUOTA        32        /* Inode is not counted to quota */
230 #define S_DIRSYNC        64        /* Directory modifications are synchronous */
```

```
231 #define S_NOCMTIME        128        /* Do not update file c/mtime */
232 #define S_SWAPFILE        256        /* Do not truncate: swapon got its bmaps */
233 #define S_PRIVATE        512        /* Inode is fs-internal */
234
235 /*
236  * Note that nosuid etc flags are inode-specific: setting some file-system
237  * flags just means all the inodes inherit those flags by default. It might be
238  * possible to override it selectively if you really wanted to with some
239  * ioctl() that is not currently implemented.
240  *
241  * Exception: MS_RDONLY is always applied to the entire file system.
242  *
243  * Unfortunately, it is possible to change a filesystems flags with it mounted
244  * with files in use.  This means that all of the inodes will not have their
245  * i_flags updated.  Hence, i_flags no longer inherit the superblock mount
246  * flags, so these have to be checked separately. -- rmk@arm.uk.linux.org
247  */
248 #define __IS_FLG(inode,flg) ((inode)->i_sb->s_flags & (flg))
249
250 #define IS_RDONLY(inode) ((inode)->i_sb->s_flags & MS_RDONLY)
251 #define IS_SYNC(inode)                (__IS_FLG(inode, MS_SYNCHRONOUS) || \
252                                         ((inode)->i_flags & S_SYNC))
253 #define IS_DIRSYNC(inode)        (__IS_FLG(inode, MS_SYNCHRONOUS|MS_DIRSYNC) || \
254                                         ((inode)->i_flags & (S_SYNC|S_DIRSYNC)))
255 #define IS_MANDLOCK(inode)        __IS_FLG(inode, MS_MANDLOCK)
256 #define IS_NOATIME(inode)   __IS_FLG(inode, MS_RDONLY|MS_NOATIME)
257 #define IS_I_VERSION(inode)   __IS_FLG(inode, MS_I_VERSION)
258
259 #define IS_NOQUOTA(inode)        ((inode)->i_flags & S_NOQUOTA)
260 #define IS_APPEND(inode)        ((inode)->i_flags & S_APPEND)
261 #define IS_IMMUTABLE(inode)        ((inode)->i_flags & S_IMMUTABLE)
262 #define IS_POSIXACL(inode)        __IS_FLG(inode, MS_POSIXACL)
263
264 #define IS_DEADDIR(inode)        ((inode)->i_flags & S_DEAD)
265 #define IS_NOCMTIME(inode)        ((inode)->i_flags & S_NOCMTIME)
266 #define IS_SWAPFILE(inode)        ((inode)->i_flags & S_SWAPFILE)
267 #define IS_PRIVATE(inode)        ((inode)->i_flags & S_PRIVATE)
268
269 /* the read-only stuff doesn't really belong here, but any other place is
270    probably as bad and I don't want to create yet another include file. */
271
272 #define BLKROSET   _IO(0x12,93)        /* set device read-only (0 = read-write) */
273 #define BLKROGET   _IO(0x12,94)        /* get read-only status (0 = read_write) */
274 #define BLKRRPART  _IO(0x12,95)        /* re-read partition table */
275 #define BLKGETSIZE _IO(0x12,96)        /* return device size /512 (long *arg) */
276 #define BLKFLSBUF  _IO(0x12,97)        /* flush buffer cache */
277 #define BLKRASET   _IO(0x12,98)        /* set read ahead for block device */
278 #define BLKRAGET   _IO(0x12,99)        /* get current read ahead setting */
279 #define BLKFRASET  _IO(0x12,100)/* set filesystem (mm/filemap.c) read-ahead */
280 #define BLKFRAGET  _IO(0x12,101)/* get filesystem (mm/filemap.c) read-ahead */
281 #define BLKSECTSET _IO(0x12,102)/* set max sectors per request (ll_rw_blk.c) */
282 #define BLKSECTGET _IO(0x12,103)/* get max sectors per request (ll_rw_blk.c) */
283 #define BLKSSZGET  _IO(0x12,104)/* get block device sector size */
284 #if 0
285 #define BLKPG      _IO(0x12,105)/* See blkpg.h */
286
```

```
287 /* Some people are morons.  Do not use sizeof! */
288
289 #define BLKELVGET  _IOR(0x12,106,size_t)/* elevator get */
290 #define BLKELVSET  _IOW(0x12,107,size_t)/* elevator set */
291 /* This was here just to show that the number is taken -
292    probably all these _IO(0x12,*) ioctls should be moved to blkpg.h. */
293 #endif
294 /* A jump here: 108-111 have been used for various private purposes. */
295 #define BLKBSZGET  _IOR(0x12,112,size_t)
296 #define BLKBSZSET  _IOW(0x12,113,size_t)
297 #define BLKGETSIZE64 _IOR(0x12,114,size_t)       /* return device size in bytes (u64 *arg) */
298 #define BLKTRACESETUP _IOWR(0x12,115,struct blk_user_trace_setup)
299 #define BLKTRACESTART _IO(0x12,116)
300 #define BLKTRACESTOP _IO(0x12,117)
301 #define BLKTRACETEARDOWN _IO(0x12,118)
302 #define BLKDISCARD _IO(0x12,119)
303
304 #define BMAP_IOCTL 1               /* obsolete - kept for compatibility */
305 #define FIBMAP           _IO(0x00,1)        /* bmap access */
306 #define FIGETBSZ   _IO(0x00,2)        /* get the block size used for bmap */
307 #define FIFREEZE        _IOWR('X', 119, int)        /* Freeze */
308 #define FITHAW                _IOWR('X', 120, int)        /* Thaw */
309
310 #define        FS_IOC_GETFLAGS                     _IOR('f', 1, long)
311 #define        FS_IOC_SETFLAGS                     _IOW('f', 2, long)
312 #define        FS_IOC_GETVERSION               _IOR('v', 1, long)
313 #define        FS_IOC_SETVERSION               _IOW('v', 2, long)
314 #define FS_IOC_FIEMAP                       _IOWR('f', 11, struct fiemap)
315 #define FS_IOC32_GETFLAGS               _IOR('f', 1, int)
316 #define FS_IOC32_SETFLAGS               _IOW('f', 2, int)
317 #define FS_IOC32_GETVERSION               _IOR('v', 1, int)
318 #define FS_IOC32_SETVERSION               _IOW('v', 2, int)
319
320 /*
321  * Inode flags (FS_IOC_GETFLAGS / FS_IOC_SETFLAGS)
322  */
323 #define        FS_SECRM_FL                     0x00000001 /* Secure deletion */
324 #define        FS_UNRM_FL                     0x00000002 /* Undelete */
325 #define        FS_COMPR_FL                     0x00000004 /* Compress file */
326 #define FS_SYNC_FL                     0x00000008 /* Synchronous updates */
327 #define FS_IMMUTABLE_FL                     0x00000010 /* Immutable file */
328 #define FS_APPEND_FL                     0x00000020 /* writes to file may only append */
329 #define FS_NODUMP_FL                     0x00000040 /* do not dump file */
330 #define FS_NOATIME_FL                     0x00000080 /* do not update atime */
331 /* Reserved for compression usage... */
332 #define FS_DIRTY_FL                     0x00000100
333 #define FS_COMPRBLK_FL                     0x00000200 /* One or more compressed clusters */
334 #define FS_NOCOMP_FL                     0x00000400 /* Don't compress */
335 #define FS_ECOMPR_FL                     0x00000800 /* Compression error */
336 /* End compression flags --- maybe not all used */
337 #define FS_BTREE_FL                     0x00001000 /* btree format dir */
338 #define FS_INDEX_FL                     0x00001000 /* hash-indexed directory */
339 #define FS_IMAGIC_FL                     0x00002000 /* AFS directory */
340 #define FS_JOURNAL_DATA_FL               0x00004000 /* Reserved for ext3 */
341 #define FS_NOTAIL_FL                     0x00008000 /* file tail should not be merged */
342 #define FS_DIRSYNC_FL                     0x00010000 /* dirsync behaviour (directories only) */
```

```
343 #define FS_TOPDIR_FL                    0x00020000 /* Top of directory hierarchies*/
344 #define FS_EXTENT_FL                    0x00080000 /* Extents */
345 #define FS_DIRECTIO_FL                    0x00100000 /* Use direct i/o */
346 #define FS_RESERVED_FL                    0x80000000 /* reserved for ext2 lib */
347
348 #define FS_FL_USER_VISIBLE              0x0003DFFF /* User visible flags */
349 #define FS_FL_USER_MODIFIABLE              0x000380FF /* User modifiable flags */
350
351
352 #define SYNC_FILE_RANGE_WAIT_BEFORE     1
353 #define SYNC_FILE_RANGE_WRITE             2
354 #define SYNC_FILE_RANGE_WAIT_AFTER      4
355
356 #endif /* _LINUX_FS_H */
```

Below is the file ext2_fs.h:

```
1  /*
2   *  linux/include/linux/ext2_fs.h
3   *
4   * Copyright (C) 1992, 1993, 1994, 1995
5   * Remy Card (card@masi.ibp.fr)
6   * Laboratoire MASI - Institut Blaise Pascal
7   * Universite Pierre et Marie Curie (Paris VI)
8   *
9   *  from
10  *
11  *  linux/include/linux/minix_fs.h
12  *
13  *  Copyright (C) 1991, 1992  Linus Torvalds
14  */
15
16 #ifndef _LINUX_EXT2_FS_H
17 #define _LINUX_EXT2_FS_H
18
19 #include <linux/types.h>
20 #include <linux/magic.h>
21
22 /*
23  * The second extended filesystem constants/structures
24  */
25
26 /*
27  * Define EXT2FS_DEBUG to produce debug messages
28  */
29 #undef EXT2FS_DEBUG
30
31 /*
32  * Define EXT2_RESERVATION to reserve data blocks for expanding files
33  */
34 #define EXT2_DEFAULT_RESERVE_BLOCKS     8
35 /*max window size: 1024(direct blocks) + 3([t,d]indirect blocks) */
36 #define EXT2_MAX_RESERVE_BLOCKS         1027
37 #define EXT2_RESERVE_WINDOW_NOT_ALLOCATED 0
```

```
38 /*
39  * The second extended file system version
40  */
41 #define EXT2FS_DATE                "95/08/09"
42 #define EXT2FS_VERSION                "0.5b"
43
44 /*
45  * Debug code
46  */
47 #ifdef EXT2FS_DEBUG
48 #        define ext2_debug(f, a...)        { \
49                                         printk ("EXT2-fs DEBUG (%s, %d): %s:", \
50                                                 __FILE__, __LINE__, __func__); \
51                                           printk (f, ## a); \
52                                         }
53 #else
54 #        define ext2_debug(f, a...)        /**/
55 #endif
56
57 /*
58  * Special inode numbers
59  */
60 #define        EXT2_BAD_INO                 1        /* Bad blocks inode */
61 #define EXT2_ROOT_INO                2        /* Root inode */
62 #define EXT2_BOOT_LOADER_INO         5        /* Boot loader inode */
63 #define EXT2_UNDEL_DIR_INO         6        /* Undelete directory inode */
64
65 /* First non-reserved inode for old ext2 filesystems */
66 #define EXT2_GOOD_OLD_FIRST_INO        11
67
68 /* Assume that user mode programs are passing in an ext2fs superblock, not
69  * a kernel struct super_block.  This will allow us to call the feature-test
70  * macros from user land. */
71 #define EXT2_SB(sb)        (sb)
72
73 /*
74  * Maximal count of links to a file
75  */
76 #define EXT2_LINK_MAX                32000
77
78 /*
79  * Macro-instructions used to manage several block sizes
80  */
81 #define EXT2_MIN_BLOCK_SIZE                1024
82 #define        EXT2_MAX_BLOCK_SIZE                4096
83 #define EXT2_MIN_BLOCK_LOG_SIZE                10
84 # define EXT2_BLOCK_SIZE(s)                (EXT2_MIN_BLOCK_SIZE << (s)->s_log_block_size)
85 #define        EXT2_ADDR_PER_BLOCK(s)                (EXT2_BLOCK_SIZE(s) / sizeof (__u32))
86 # define EXT2_BLOCK_SIZE_BITS(s)        ((s)->s_log_block_size + 10)
87 #define EXT2_INODE_SIZE(s)        (((s)->s_rev_level == EXT2_GOOD_OLD_REV) ? \
88                                    EXT2_GOOD_OLD_INODE_SIZE : \
89                                    (s)->s_inode_size)
90 #define EXT2_FIRST_INO(s)        (((s)->s_rev_level == EXT2_GOOD_OLD_REV) ? \
91                                    EXT2_GOOD_OLD_FIRST_INO : \
92                                    (s)->s_first_ino)
93
```

```
94  /*
95   * Macro-instructions used to manage fragments
96   */
97  #define EXT2_MIN_FRAG_SIZE              1024
98  #define        EXT2_MAX_FRAG_SIZE               4096
99  #define EXT2_MIN_FRAG_LOG_SIZE                10
100 # define EXT2_FRAG_SIZE(s)               (EXT2_MIN_FRAG_SIZE << (s)->s_log_frag_size)
101 # define EXT2_FRAGS_PER_BLOCK(s)        (EXT2_BLOCK_SIZE(s) / EXT2_FRAG_SIZE(s))
102
103 /*
104  * Structure of a blocks group descriptor
105  */
106 struct ext2_group_desc
107 {
108         __le32        bg_block_bitmap;               /* Blocks bitmap block */
109         __le32        bg_inode_bitmap;               /* Inodes bitmap block */
110         __le32        bg_inode_table;               /* Inodes table block */
111         __le16        bg_free_blocks_count;        /* Free blocks count */
112         __le16        bg_free_inodes_count;        /* Free inodes count */
113         __le16        bg_used_dirs_count;        /* Directories count */
114         __le16        bg_pad;
115         __le32        bg_reserved[3];
116 };
117
118 /*
119  * Macro-instructions used to manage group descriptors
120  */
121 # define EXT2_BLOCKS_PER_GROUP(s)        ((s)->s_blocks_per_group)
122 # define EXT2_DESC_PER_BLOCK(s)               (EXT2_BLOCK_SIZE(s) / sizeof (struct ext2_group_desc))
123 # define EXT2_INODES_PER_GROUP(s)        ((s)->s_inodes_per_group)
124
125 /*
126  * Constants relative to the data blocks
127  */
128 #define        EXT2_NDIR_BLOCKS               12
129 #define        EXT2_IND_BLOCK                      EXT2_NDIR_BLOCKS
130 #define        EXT2_DIND_BLOCK                       (EXT2_IND_BLOCK + 1)
131 #define        EXT2_TIND_BLOCK                       (EXT2_DIND_BLOCK + 1)
132 #define        EXT2_N_BLOCKS                      (EXT2_TIND_BLOCK + 1)
133
134 /*
135  * Inode flags (GETFLAGS/SETFLAGS)
136  */
137 #define        EXT2_SECRM_FL                       FS_SECRM_FL        /* Secure deletion */
138 #define        EXT2_UNRM_FL                      FS_UNRM_FL        /* Undelete */
139 #define        EXT2_COMPR_FL                       FS_COMPR_FL        /* Compress file */
140 #define EXT2_SYNC_FL                     FS_SYNC_FL        /* Synchronous updates */
141 #define EXT2_IMMUTABLE_FL               FS_IMMUTABLE_FL        /* Immutable file */
142 #define EXT2_APPEND_FL                       FS_APPEND_FL        /* writes to file may only append */
143 #define EXT2_NODUMP_FL                       FS_NODUMP_FL        /* do not dump file */
144 #define EXT2_NOATIME_FL                       FS_NOATIME_FL        /* do not update atime */
145 /* Reserved for compression usage... */
146 #define EXT2_DIRTY_FL                     FS_DIRTY_FL
147 #define EXT2_COMPRBLK_FL                FS_COMPRBLK_FL        /* One or more compressed clusters */
148 #define EXT2_NOCOMP_FL                       FS_NOCOMP_FL        /* Don't compress */
149 #define EXT2_ECOMPR_FL                       FS_ECOMPR_FL        /* Compression error */
```

```
150 /* End compression flags --- maybe not all used */
151 #define EXT2_BTREE_FL                       FS_BTREE_FL        /* btree format dir */
152 #define EXT2_INDEX_FL                       FS_INDEX_FL        /* hash-indexed directory */
153 #define EXT2_IMAGIC_FL                       FS_IMAGIC_FL       /* AFS directory */
154 #define EXT2_JOURNAL_DATA_FL               FS_JOURNAL_DATA_FL /* Reserved for ext3 */
155 #define EXT2_NOTAIL_FL                       FS_NOTAIL_FL       /* file tail should not be merged */
156 #define EXT2_DIRSYNC_FL                       FS_DIRSYNC_FL       /* dirsync behaviour (directories only) */
157 #define EXT2_TOPDIR_FL                       FS_TOPDIR_FL       /* Top of directory hierarchies*/
158 #define EXT2_RESERVED_FL               FS_RESERVED_FL       /* reserved for ext2 lib */
159
160 #define EXT2_FL_USER_VISIBLE               FS_FL_USER_VISIBLE       /* User visible flags */
161 #define EXT2_FL_USER_MODIFIABLE               FS_FL_USER_MODIFIABLE       /* User modifiable flags */
162
163 /* Flags that should be inherited by new inodes from their parent. */
164 #define EXT2_FL_INHERITED (EXT2_SECRM_FL | EXT2_UNRM_FL | EXT2_COMPR_FL |\
165                            EXT2_SYNC_FL | EXT2_IMMUTABLE_FL | EXT2_APPEND_FL |\
166                            EXT2_NODUMP_FL | EXT2_NOATIME_FL | EXT2_COMPRBLK_FL|\
167                            EXT2_NOCOMP_FL | EXT2_JOURNAL_DATA_FL |\
168                            EXT2_NOTAIL_FL | EXT2_DIRSYNC_FL)
169
170 /* Flags that are appropriate for regular files (all but dir-specific ones). */
171 #define EXT2_REG_FLMASK (~(EXT2_DIRSYNC_FL | EXT2_TOPDIR_FL))
172
173 /* Flags that are appropriate for non-directories/regular files. */
174 #define EXT2_OTHER_FLMASK (EXT2_NODUMP_FL | EXT2_NOATIME_FL)
175
176 /* Mask out flags that are inappropriate for the given type of inode. */
177 static __inline__ __u32 ext2_mask_flags(umode_t mode, __u32 flags)
178 {
179         if (S_ISDIR(mode))
180                 return flags;
181         else if (S_ISREG(mode))
182                 return flags & EXT2_REG_FLMASK;
183         else
184                 return flags & EXT2_OTHER_FLMASK;
185 }
186
187 /*
188  * ioctl commands
189  */
190 #define        EXT2_IOC_GETFLAGS               FS_IOC_GETFLAGS
191 #define        EXT2_IOC_SETFLAGS               FS_IOC_SETFLAGS
192 #define        EXT2_IOC_GETVERSION               FS_IOC_GETVERSION
193 #define        EXT2_IOC_SETVERSION               FS_IOC_SETVERSION
194 #define        EXT2_IOC_GETRSVSZ               _IOR('f', 5, long)
195 #define        EXT2_IOC_SETRSVSZ               _IOW('f', 6, long)
196
197 /*
198  * ioctl commands in 32 bit emulation
199  */
200 #define EXT2_IOC32_GETFLAGS               FS_IOC32_GETFLAGS
201 #define EXT2_IOC32_SETFLAGS               FS_IOC32_SETFLAGS
202 #define EXT2_IOC32_GETVERSION               FS_IOC32_GETVERSION
203 #define EXT2_IOC32_SETVERSION               FS_IOC32_SETVERSION
204
205 /*
```

```
206  * Structure of an inode on the disk
207  */
208 struct ext2_inode {
209         __le16        i_mode;               /* File mode */
210         __le16        i_uid;               /* Low 16 bits of Owner Uid */
211         __le32        i_size;               /* Size in bytes */
212         __le32        i_atime;        /* Access time */
213         __le32        i_ctime;        /* Creation time */
214         __le32        i_mtime;        /* Modification time */
215         __le32        i_dtime;        /* Deletion Time */
216         __le16        i_gid;                /* Low 16 bits of Group Id */
217         __le16        i_links_count;        /* Links count */
218         __le32        i_blocks;        /* Blocks count */
219         __le32        i_flags;        /* File flags */
220         union {
221                 struct {
222                         __le32  l_i_reserved1;
223                 } linux1;
224                 struct {
225                         __le32  h_i_translator;
226                 } hurd1;
227                 struct {
228                         __le32  m_i_reserved1;
229                 } masix1;
230         } osd1;                                 /* OS dependent 1 */
231         __le32        i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
232         __le32        i_generation;        /* File version (for NFS) */
233         __le32        i_file_acl;        /* File ACL */
234         __le32        i_dir_acl;        /* Directory ACL */
235         __le32        i_faddr;        /* Fragment address */
236         union {
237                 struct {
238                         __u8        l_i_frag;        /* Fragment number */
239                         __u8        l_i_fsize;        /* Fragment size */
240                         __u16        i_pad1;
241                         __le16        l_i_uid_high;        /* these 2 fields    */
242                         __le16        l_i_gid_high;        /* were reserved2[0] */
243                         __u32        l_i_reserved2;
244                 } linux2;
245                 struct {
246                         __u8        h_i_frag;        /* Fragment number */
247                         __u8        h_i_fsize;        /* Fragment size */
248                         __le16        h_i_mode_high;
249                         __le16        h_i_uid_high;
250                         __le16        h_i_gid_high;
251                         __le32        h_i_author;
252                 } hurd2;
253                 struct {
254                         __u8        m_i_frag;        /* Fragment number */
255                         __u8        m_i_fsize;        /* Fragment size */
256                         __u16        m_pad1;
257                         __u32        m_i_reserved2[2];
258                 } masix2;
259         } osd2;                                 /* OS dependent 2 */
260 };
261
```

```
262 #define i_size_high        i_dir_acl
263
264 #if defined(__KERNEL__) || defined(__linux__)
265 #define i_reserved1        osd1.linux1.l_i_reserved1
266 #define i_frag                osd2.linux2.l_i_frag
267 #define i_fsize                osd2.linux2.l_i_fsize
268 #define i_uid_low        i_uid
269 #define i_gid_low        i_gid
270 #define i_uid_high        osd2.linux2.l_i_uid_high
271 #define i_gid_high        osd2.linux2.l_i_gid_high
272 #define i_reserved2        osd2.linux2.l_i_reserved2
273 #endif
274
275 #ifdef        __hurd__
276 #define i_translator        osd1.hurd1.h_i_translator
277 #define i_frag                osd2.hurd2.h_i_frag
278 #define i_fsize                osd2.hurd2.h_i_fsize
279 #define i_uid_high        osd2.hurd2.h_i_uid_high
280 #define i_gid_high        osd2.hurd2.h_i_gid_high
281 #define i_author        osd2.hurd2.h_i_author
282 #endif
283
284 #ifdef        __masix__
285 #define i_reserved1        osd1.masix1.m_i_reserved1
286 #define i_frag                osd2.masix2.m_i_frag
287 #define i_fsize                osd2.masix2.m_i_fsize
288 #define i_reserved2        osd2.masix2.m_i_reserved2
289 #endif
290
291 /*
292  * File system states
293  */
294 #define        EXT2_VALID_FS                        0x0001        /* Unmounted cleanly */
295 #define        EXT2_ERROR_FS                        0x0002        /* Errors detected */
296
297 /*
298  * Mount flags
299  */
300 #define EXT2_MOUNT_CHECK                0x000001  /* Do mount-time checks */
301 #define EXT2_MOUNT_OLDALLOC                0x000002  /* Don't use the new Orlov allocator */
302 #define EXT2_MOUNT_GRPID                0x000004  /* Create files with directory's group */
303 #define EXT2_MOUNT_DEBUG                0x000008  /* Some debugging messages */
304 #define EXT2_MOUNT_ERRORS_CONT                0x000010  /* Continue on errors */
305 #define EXT2_MOUNT_ERRORS_RO                0x000020  /* Remount fs ro on errors */
306 #define EXT2_MOUNT_ERRORS_PANIC                0x000040  /* Panic on errors */
307 #define EXT2_MOUNT_MINIX_DF                0x000080  /* Mimics the Minix statfs */
308 #define EXT2_MOUNT_NOBH                        0x000100  /* No buffer_heads */
309 #define EXT2_MOUNT_NO_UID32                0x000200  /* Disable 32-bit UIDs */
310 #define EXT2_MOUNT_XATTR_USER                0x004000  /* Extended user attributes */
311 #define EXT2_MOUNT_POSIX_ACL                0x008000  /* POSIX Access Control Lists */
312 #define EXT2_MOUNT_XIP                        0x010000  /* Execute in place */
313 #define EXT2_MOUNT_USRQUOTA                0x020000  /* user quota */
314 #define EXT2_MOUNT_GRPQUOTA                0x040000  /* group quota */
315 #define EXT2_MOUNT_RESERVATION                0x080000  /* Preallocation */
316
317
```

```
318 #define clear_opt(o, opt)               o &= ~EXT2_MOUNT_##opt
319 #define set_opt(o, opt)                      o |= EXT2_MOUNT_##opt
320 #define test_opt(sb, opt)               (EXT2_SB(sb)->s_mount_opt & \
321                                         EXT2_MOUNT_##opt)
322 /*
323  * Maximal mount counts between two filesystem checks
324  */
325 #define EXT2_DFL_MAX_MNT_COUNT               20        /* Allow 20 mounts */
326 #define EXT2_DFL_CHECKINTERVAL               0        /* Don't use interval check */
327
328 /*
329  * Behaviour when detecting errors
330  */
331 #define EXT2_ERRORS_CONTINUE               1        /* Continue execution */
332 #define EXT2_ERRORS_RO                       2        /* Remount fs read-only */
333 #define EXT2_ERRORS_PANIC               3        /* Panic */
334 #define EXT2_ERRORS_DEFAULT               EXT2_ERRORS_CONTINUE
335
336 /*
337  * Structure of the super block
338  */
339 struct ext2_super_block {
340         __le32        s_inodes_count;               /* Inodes count */
341         __le32        s_blocks_count;               /* Blocks count */
342         __le32        s_r_blocks_count;        /* Reserved blocks count */
343         __le32        s_free_blocks_count;        /* Free blocks count */
344         __le32        s_free_inodes_count;        /* Free inodes count */
345         __le32        s_first_data_block;        /* First Data Block */
346         __le32        s_log_block_size;        /* Block size */
347         __le32        s_log_frag_size;        /* Fragment size */
348         __le32        s_blocks_per_group;        /* # Blocks per group */
349         __le32        s_frags_per_group;        /* # Fragments per group */
350         __le32        s_inodes_per_group;        /* # Inodes per group */
351         __le32        s_mtime;                /* Mount time */
352         __le32        s_wtime;                /* Write time */
353         __le16        s_mnt_count;                /* Mount count */
354         __le16        s_max_mnt_count;        /* Maximal mount count */
355         __le16        s_magic;                /* Magic signature */
356         __le16        s_state;                /* File system state */
357         __le16        s_errors;                /* Behaviour when detecting errors */
358         __le16        s_minor_rev_level;        /* minor revision level */
359         __le32        s_lastcheck;                /* time of last check */
360         __le32        s_checkinterval;        /* max. time between checks */
361         __le32        s_creator_os;                /* OS */
362         __le32        s_rev_level;                /* Revision level */
363         __le16        s_def_resuid;                /* Default uid for reserved blocks */
364         __le16        s_def_resgid;                /* Default gid for reserved blocks */
365         /*
366          * These fields are for EXT2_DYNAMIC_REV superblocks only.
367          *
368          * Note: the difference between the compatible feature set and
369          * the incompatible feature set is that if there is a bit set
370          * in the incompatible feature set that the kernel doesn't
371          * know about, it should refuse to mount the filesystem.
372          *
373          * e2fsck's requirements are more strict; if it doesn't know
```

```
374         * about a feature in either the compatible or incompatible
375         * feature set, it must abort and not try to meddle with
376         * things it doesn't understand...
377         */
378        __le32        s_first_ino;               /* First non-reserved inode */
379        __le16   s_inode_size;               /* size of inode structure */
380        __le16        s_block_group_nr;         /* block group # of this superblock */
381        __le32        s_feature_compat;         /* compatible feature set */
382        __le32        s_feature_incompat;         /* incompatible feature set */
383        __le32        s_feature_ro_compat;         /* readonly-compatible feature set */
384        __u8        s_uuid[16];               /* 128-bit uuid for volume */
385        char        s_volume_name[16];         /* volume name */
386        char        s_last_mounted[64];         /* directory where last mounted */
387        __le32        s_algorithm_usage_bitmap; /* For compression */
388        /*
389         * Performance hints.  Directory preallocation should only
390         * happen if the EXT2_COMPAT_PREALLOC flag is on.
391         */
392        __u8        s_prealloc_blocks;        /* Nr of blocks to try to preallocate*/
393        __u8        s_prealloc_dir_blocks;        /* Nr to preallocate for dirs */
394        __u16        s_padding1;
395        /*
396         * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
397         */
398        __u8        s_journal_uuid[16];        /* uuid of journal superblock */
399        __u32        s_journal_inum;                /* inode number of journal file */
400        __u32        s_journal_dev;                /* device number of journal file */
401        __u32        s_last_orphan;                /* start of list of inodes to delete */
402        __u32        s_hash_seed[4];                /* HTREE hash seed */
403        __u8        s_def_hash_version;        /* Default hash version to use */
404        __u8        s_reserved_char_pad;
405        __u16        s_reserved_word_pad;
406        __le32        s_default_mount_opts;
407         __le32        s_first_meta_bg;         /* First metablock block group */
408        __u32        s_reserved[190];        /* Padding to the end of the block */
409    };
410
411    /*
412     * Codes for operating systems
413     */
414    #define EXT2_OS_LINUX                0
415    #define EXT2_OS_HURD                1
416    #define EXT2_OS_MASIX                2
417    #define EXT2_OS_FREEBSD                3
418    #define EXT2_OS_LITES                4
419
420    /*
421     * Revision levels
422     */
423    #define EXT2_GOOD_OLD_REV        0        /* The good old (original) format */
424    #define EXT2_DYNAMIC_REV        1        /* V2 format w/ dynamic inode sizes */
425
426    #define EXT2_CURRENT_REV        EXT2_GOOD_OLD_REV
427    #define EXT2_MAX_SUPP_REV        EXT2_DYNAMIC_REV
428
429    #define EXT2_GOOD_OLD_INODE_SIZE 128
```

```
430
431 /*
432  * Feature set definitions
433  */
434
435 #define EXT2_HAS_COMPAT_FEATURE(sb,mask)                        \
436         ( EXT2_SB(sb)->s_es->s_feature_compat & cpu_to_le32(mask) )
437 #define EXT2_HAS_RO_COMPAT_FEATURE(sb,mask)                      \
438         ( EXT2_SB(sb)->s_es->s_feature_ro_compat & cpu_to_le32(mask) )
439 #define EXT2_HAS_INCOMPAT_FEATURE(sb,mask)                       \
440         ( EXT2_SB(sb)->s_es->s_feature_incompat & cpu_to_le32(mask) )
441 #define EXT2_SET_COMPAT_FEATURE(sb,mask)                        \
442         EXT2_SB(sb)->s_es->s_feature_compat |= cpu_to_le32(mask)
443 #define EXT2_SET_RO_COMPAT_FEATURE(sb,mask)                      \
444         EXT2_SB(sb)->s_es->s_feature_ro_compat |= cpu_to_le32(mask)
445 #define EXT2_SET_INCOMPAT_FEATURE(sb,mask)                       \
446         EXT2_SB(sb)->s_es->s_feature_incompat |= cpu_to_le32(mask)
447 #define EXT2_CLEAR_COMPAT_FEATURE(sb,mask)                      \
448         EXT2_SB(sb)->s_es->s_feature_compat &= ~cpu_to_le32(mask)
449 #define EXT2_CLEAR_RO_COMPAT_FEATURE(sb,mask)                    \
450         EXT2_SB(sb)->s_es->s_feature_ro_compat &= ~cpu_to_le32(mask)
451 #define EXT2_CLEAR_INCOMPAT_FEATURE(sb,mask)                     \
452         EXT2_SB(sb)->s_es->s_feature_incompat &= ~cpu_to_le32(mask)
453
454 #define EXT2_FEATURE_COMPAT_DIR_PREALLOC        0x0001
455 #define EXT2_FEATURE_COMPAT_IMAGIC_INODES       0x0002
456 #define EXT3_FEATURE_COMPAT_HAS_JOURNAL         0x0004
457 #define EXT2_FEATURE_COMPAT_EXT_ATTR            0x0008
458 #define EXT2_FEATURE_COMPAT_RESIZE_INO          0x0010
459 #define EXT2_FEATURE_COMPAT_DIR_INDEX           0x0020
460 #define EXT2_FEATURE_COMPAT_ANY                 0xffffffff
461
462 #define EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER     0x0001
463 #define EXT2_FEATURE_RO_COMPAT_LARGE_FILE       0x0002
464 #define EXT2_FEATURE_RO_COMPAT_BTREE_DIR        0x0004
465 #define EXT2_FEATURE_RO_COMPAT_ANY              0xffffffff
466
467 #define EXT2_FEATURE_INCOMPAT_COMPRESSION       0x0001
468 #define EXT2_FEATURE_INCOMPAT_FILETYPE          0x0002
469 #define EXT3_FEATURE_INCOMPAT_RECOVER           0x0004
470 #define EXT3_FEATURE_INCOMPAT_JOURNAL_DEV       0x0008
471 #define EXT2_FEATURE_INCOMPAT_META_BG           0x0010
472 #define EXT2_FEATURE_INCOMPAT_ANY               0xffffffff
473
474 #define EXT2_FEATURE_COMPAT_SUPP        EXT2_FEATURE_COMPAT_EXT_ATTR
475 #define EXT2_FEATURE_INCOMPAT_SUPP        (EXT2_FEATURE_INCOMPAT_FILETYPE| \
476                                            EXT2_FEATURE_INCOMPAT_META_BG)
477 #define EXT2_FEATURE_RO_COMPAT_SUPP       (EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER| \
478                                            EXT2_FEATURE_RO_COMPAT_LARGE_FILE| \
479                                            EXT2_FEATURE_RO_COMPAT_BTREE_DIR)
480 #define EXT2_FEATURE_RO_COMPAT_UNSUPPORTED      ~EXT2_FEATURE_RO_COMPAT_SUPP
481 #define EXT2_FEATURE_INCOMPAT_UNSUPPORTED       ~EXT2_FEATURE_INCOMPAT_SUPP
482
483 /*
484  * Default values for user and/or group using reserved blocks
485  */
```

```
#define        EXT2_DEF_RESUID                0
#define        EXT2_DEF_RESGID                0

/*
 * Default mount options
 */
#define EXT2_DEFM_DEBUG                0x0001
#define EXT2_DEFM_BSDGROUPS        0x0002
#define EXT2_DEFM_XATTR_USER        0x0004
#define EXT2_DEFM_ACL                0x0008
#define EXT2_DEFM_UID16                0x0010
    /* Not used by ext2, but reserved for use by ext3 */
#define EXT3_DEFM_JMODE                0x0060
#define EXT3_DEFM_JMODE_DATA        0x0020
#define EXT3_DEFM_JMODE_ORDERED        0x0040
#define EXT3_DEFM_JMODE_WBACK        0x0060

/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
        __le32        inode;                        /* Inode number */
        __le16        rec_len;                /* Directory entry length */
        __le16        name_len;                /* Name length */
        char        name[EXT2_NAME_LEN];        /* File name */
};

/*
 * The new version of the directory entry.  Since EXT2 structures are
 * stored in intel byte order, and the name_len field could never be
 * bigger than 255 chars, it's safe to reclaim the extra byte for the
 * file_type field.
 */
struct ext2_dir_entry_2 {
        __le32        inode;                        /* Inode number */
        __le16        rec_len;                /* Directory entry length */
        __u8        name_len;                /* Name length */
        __u8        file_type;
        char        name[EXT2_NAME_LEN];        /* File name */
};

/*
 * Ext2 directory file types.  Only the low 3 bits are used.  The
 * other bits are reserved for now.
 */
enum {
        EXT2_FT_UNKNOWN,
        EXT2_FT_REG_FILE,
        EXT2_FT_DIR,
        EXT2_FT_CHRDEV,
        EXT2_FT_BLKDEV,
        EXT2_FT_FIFO,
        EXT2_FT_SOCK,
        EXT2_FT_SYMLINK,
```

```
542         EXT2_FT_MAX
543 };
544
545 /*
546  * EXT2_DIR_PAD defines the directory entries boundaries
547  *
548  * NOTE: It must be a multiple of 4
549  */
550 #define EXT2_DIR_PAD                        4
551 #define EXT2_DIR_ROUND                        (EXT2_DIR_PAD - 1)
552 #define EXT2_DIR_REC_LEN(name_len)        (((name_len) + 8 + EXT2_DIR_ROUND) & \
553                                           ~EXT2_DIR_ROUND)
554 #define EXT2_MAX_REC_LEN                ((1<<16)-1)
555
556 #endif        /* _LINUX_EXT2_FS_H */
```

This header file contains the definitions of structures for ext2 file system. That header file can also be found in /usr/include/linux of your Linux system.

```
1  all: findblocks
2
3  #findblocks: findblocks.c
4  #            gcc  -g -Wall -D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE -o findblocks findblocks.c
5
6  findblocks: findblocks.c
7              gcc  -g -Wall -o findblocks findblocks.c
8
9
10 clean:
11         rm -fr *~  findblocks
```

After compiling the program, we can now use it to access and operate on our new file system directly. The name of the program is `findblocks` and when we just type `findblocks`, it will give the following help about the parameters that it expects:

```
findblocks <devicefilename>
     -s : display superblock info
     -g : display groups info
     -i <pathname> : display inode info for file/dir <pathname>
     -dumpblock <blocknum> -x : dump disk block <blocknum> in hex
     -dumpblock <blocknum> -t : dump disk block <blocknum> in ascii
     -d <pathname> : display directory content for directory <pathname>
     -dt <pathname> : traverse directories in <pathname> listing content
     -de <pathname> : display the directory entry for <pathname>
     -blocks <pathname> : display disk block numbers of file/dir <pathname>
     -data <pathname> -x : dump content of file/dir <pathname> in hex
     -data <pathname> -t : dump content of file/dir <pathname> in ascii
```

```
-data <pathname> -x <start> <length> : dump file region content  in hex
-data <pathname> -t <start> <length> : dump file region content in ascii
```

Now, lets run the program to get superblock information of our new file system. We type:

```
findblocks /dev/loop0 -s
```

And we get the following output, which a small portion of the information included in the superblock of the file system:

```
$ sudo ./findblocks  /dev/loop0 -s
inode_count=62720
block_count=250000
first_data_block=0
magic_number=ef53
inode_size=128
inodes_per_group=7840
blocks_per_group=32768
log_block_size=2
```

Now, lets learn about the block groups on the virtual disk. Note that the ext2 file system groups the blocks of a disk so that it tries to allocate blocks to a file from the same group (to reduce the seek time). As the superblock information shows, the number of blocks in one group is 32768. We type the following to get groups information:

```
findblocks /dev/loop0 -g
```

Below is the output.

```
$ sudo ./findblocks  /dev/loop0 -g
size of group desc structure  = 32
group 0: bitmap_block#=2 inodetable_block#=4 free_block_count=32514
group 1: bitmap_block#=32770 inodetable_block#=32772 free_block_count=32515
group 2: bitmap_block#=65538 inodetable_block#=65540 free_block_count=32519
group 3: bitmap_block#=98306 inodetable_block#=98308 free_block_count=32519
group 4: bitmap_block#=131074 inodetable_block#=131076 free_block_count=32519
group 5: bitmap_block#=163842 inodetable_block#=163844 free_block_count=32519
group 6: bitmap_block#=196610 inodetable_block#=196612 free_block_count=32519
group 7: bitmap_block#=229378 inodetable_block#=229380 free_block_count=20375
```

There are 8 block groups: 0 through 7. For each block group, the output tells where we can find the corresponding bitmap and inode table. Inode table is a table of inodes. There

is one inode per file. Some of the inodes in the table may be used at the moment; the remaining ones nay be unused at the moment. If a file is to be created in a group, the file system first searched for an unused inode in the inode table of that group and allocates that inode to the file. Note that all inodes are created initially when we create the file system.

Let us now print the contents of the directories in the following path of the new file system: /dir1/dir2/w. We type:

```
sudo ./findblocks /dev/loop0 -dt /dir1/dir2/w
```

And we get the following output:

```
korpe@pckorpe:~/data/book/findblocks_ext2$ sudo ./findblocks /dev/loop0 -dt /dir1/dir2/w
directory: /
type=2 inode=2          rec_len=12    name_len=1     name = .
type=2 inode=2          rec_len=12    name_len=2     name = ..
type=2 inode=11         rec_len=20    name_len=10    name = lost+found
type=2 inode=7841       rec_len=12    name_len=4     name = dir1
type=1 inode=12         rec_len=20    name_len=9     name = file1.txt
type=1 inode=13         rec_len=4020  name_len=9     name = file2.txt

directory: dir1
type=2 inode=7841       rec_len=12    name_len=1     name = .
type=2 inode=2          rec_len=12    name_len=2     name = ..
type=2 inode=7842       rec_len=4072  name_len=4     name = dir2

directory: dir2
type=2 inode=7842       rec_len=12    name_len=1     name = .
type=2 inode=7841       rec_len=12    name_len=2     name = ..
type=1 inode=7847       rec_len=12    name_len=1     name = x
type=1 inode=7844       rec_len=12    name_len=1     name = y
type=1 inode=7845       rec_len=12    name_len=1     name = z
type=1 inode=7848       rec_len=4036  name_len=1     name = w
```

We can also get information about the inode of a file in the new file system. Lets get information about the inode of the file: /dir1/dir2/w. We type:

```
sudo ./findblocks /dev/loop0 -i /dir1/dir2/w
```

and we get:

```
korpe@pckorpe:~/data/book/findblocks_ext2$ sudo ./findblocks /dev/loop0 -i /dir1/dir2/w
```

```
inode_number=7848
size_in_bytes=52
uid=1000
links_count=1
blocks_count=8
mode=81a4
size_in_blocks=1
i_block[0]=53248
```

We can also dump the content of the file in hex or in asci. To dump the content of the file /dir1/dir2/w in asci, we type:

```
sudo ./findblocks /dev/loop0 -data /dir1/dir2/w -t
```

and we get the following output:

```
$ sudo ./findblocks /dev/loop0 -data /dir1/dir2/w -t
This is another file.

The name of the file is: w
```

Lets exercise with another file that is slightly larger: the file y. To learn about the data blocks of the file we type:

```
sudo ./findblocks /dev/loop0 -blocks /dir1/dir2/y
```

Below is the output:

```
korpe@pckorpe:~/data/book/findblocks_ext2$ sudo ./findblocks /dev/loop1 -blocks /dir1/dir2/y
file_block=0     disk_block=2048       group=0
file_block=1     disk_block=2049       group=0
file_block=2     disk_block=2050       group=0
file_block=3     disk_block=2051       group=0
file_block=4     disk_block=2052       group=0
file_block=5     disk_block=2053       group=0
file_block=6     disk_block=2054       group=0
file_block=7     disk_block=2055       group=0
file_block=8     disk_block=2056       group=0
```

It tells that there are 9 disk blocks allocated for the file y to store the content of the file. It also tells which disk blocks (their numbers) these blocks are. For example, the file block

0 is located/stored on disk block 2048. The file block 4 is located in disk block 2052. By chance, the blocks allocated to this file are contiguous in the disk. That does not have to like that all the time.

Chapter 10

Kernel Programming and A Memory Tool

10.1 Kernel Module Programming and A Memory Tool

In this section we will provide an example about how to develop a kernel module as a virtual driver. Then we will also show the implementation of a memory tool that will show how the module can be used to get some memory usage information for processes.

Assume we want to write a program (programs) that will enable us to view the page table content and frame content (the content of a frame allocated to the process) of a process. Our program will have two parts: a driver part (module part) and a user-space program part (tool) that will use the driver. The driver part will be running in kernel mode and kernel space. It can access the page table content and frame content for a process. The tool part will ask the driver part to retrieve the information from the kernel to an application buffer in the tool. The tool part is a user-space program running in user mode. From now on, we will refer to these two parts as tool (user-space part) and driver (kernel part).

The tool part will be a program that will be invoked by a user with some command line parameters. Depending on what parameters are specified, it will do one of the following:

- Given a 'pid', print the outer page table content of the process.
- Given a 'pid' and 'index', print the content of the the inner page table with index 'index'.
- Given a 'pid' and 'index1' and 'index2', print the content of the corresponding memory frame (corresponding page). The driver will find and access the frame content and will give the content to the tool. The driver will find the content of the frame

in RAM as follows. The driver will first use the 'index1' as index to the outer page table of the process. From there it will retrieve the frame number of the frame containing the inner table. Then the inner table will be accessed. This means accessing the inner table pointed by the entry 'index1' of the outer table. The driver will use 'index2' as index to the innner table. It will look to that entry (with index index'2') in the inner table. That entry will give a frame number. It is the number of the frame holding the page of the process that we are interested in. We would like to the print/dump the content of that frame (i.e. that page of the process) The driver will access that frame and will give (copy) the content of the frame to the tool.

Below is the code of our memory tool (tool.c). This is an application level code.

```
1  /* -*- linux-c -*- */
2  /* $Id: tool.c,v 1.1 2010/04/14 12:16:40 korpe Exp korpe $ */
3
4  #include <stdio.h>
5  #include <unistd.h>
6  #include <malloc.h>
7  #include <stdlib.h>
8  #include <malloc.h>
9  #include <string.h>
10 #include <fcntl.h>
11
12 #define DEVICE_NAME "/dev/mydevice"
13
14 #define PAGESIZE 4096
15 #define NUM_ENTRIES (PAGESIZE/4)
16 #define KERN_OFFSET 768
17
18 #define ONEMEGABYTE 1048576 /* 2^20 */
19
20 #define INDEX_REFERRING_OUTER_TABLE  NUM_ENTRIES+1
21
22 struct order_struct {
23         int code;
24         int p1;
25         int p2;
26         int p3;
27 };
28 #define INVALID_CODE  -1
29 #define GET_FRAME 1
30 #define GET_OUTER_TABLE 2
31 #define GET_INNER_TABLE 3
32
33
34 #define HEX_DUMP 1
35 #define ASCI_DUMP 2
36
37 /* little endian machine */
38 struct pt_e {
39         unsigned int present : 1;   /* b0: Present Bit */
40         unsigned int rdwr : 1;      /* b1: Read/Write */
```

```
41          unsigned int user_kern : 1; /* b2: User/Supervisor */
42          unsigned int write_thr : 1; /* b3: Write-Through */
43          unsigned int cache_dis : 1; /* b4: Cache Disabled */
44          unsigned int accessed : 1;  /* b5: Accessed */
45          unsigned int dirty : 1;     /* b6: Reserved (outer), Dirty (inner) */
46          unsigned int pgsize : 1;    /* b7: Page Size (outer), Reserved (inner) */
47          unsigned int glob : 1;      /* b8: Global Page */
48          unsigned int avail : 3;     /* b11..b9: 1110 0000 0000: Available */
49          unsigned int framenum : 20; /* b31..b12: Frame Number */
50  };
51
52
53  void
54  error_exit (char *str)
55  {
56          printf ("%s", str);
57          exit (1);
58  }
59
60  void
61  print_table (unsigned char *buf, int index )
62  {
63          int *p;
64          int i, k;
65          unsigned int entry;
66          struct pt_e *pte;
67          int inner_count = 0;
68          int kernel_count = 0;
69
70          p = (unsigned int *) buf;
71
72          for (i = 0; i < NUM_ENTRIES; ++i) {
73                  pte = (struct pt_e *) &(p[i]);
74                  if (pte->present) {
75                          printf ("pte[%4d]: ", i);
76                          if (index  == INDEX_REFERRING_OUTER_TABLE) {
77                                  if (i < KERN_OFFSET) {
78                                          inner_count++;
79                                          printf ("it=%-6d ",  inner_count);
80                                  }
81                                  else {
82                                          kernel_count++;
83                                          printf ("kt=%-6d ",  kernel_count);
84                                  }
85                          }
86                          else
87                                  printf ("pg=%-6d ",  (index * NUM_ENTRIES) + i);
88
89                          printf ("f=%-6x av=%d g=%d ps=%d d=%d a=%d cd=%d wt=%d us=%d rw=%d p=%d\n",
90                                  pte->framenum, pte->avail, pte->glob, pte->pgsize,
91                                  pte->dirty, pte->accessed,
92                                  pte->cache_dis, pte->write_thr, pte->user_kern,
93                                  pte->rdwr, pte->present);
94                  }
95          }
96  }
```

```
97
98
99  unsigned int
100 find_phy_memsize (int fd, int pid)
101 {
102         struct pt_e *o_pte, *i_pte;
103         int i, j;
104         unsigned int outer_table[NUM_ENTRIES];
105         unsigned int inner_table[NUM_ENTRIES];
106         int fr_count = 0;
107         int ret;
108         struct order_struct  order;
109         unsigned char buf[PAGESIZE];
110
111
112         order.code = GET_OUTER_TABLE;
113         order.p1 = pid;
114         memcpy ( (void*)buf, (void *) &order, sizeof (order));
115
116         ret = write (fd, buf, sizeof(order));
117         if (ret != 0)
118                 error_exit ("write failed\n");
119
120         ret = read (fd, (unsigned char *) outer_table, PAGESIZE);
121         if (ret != 0)
122                 error_exit ("read failed\n");
123
124         for (i = 0; i < KERN_OFFSET; ++i) {
125                 o_pte = (struct pt_e *) &(outer_table[i]);
126                 if (o_pte->present)  {
127                         order.code = GET_INNER_TABLE;
128                         order.p1 = pid;
129                         order.p2 = i;
130                         memcpy ( (void*)buf, (void *) &order, sizeof (order));
131
132                         ret = write (fd, buf, sizeof(order));
133                         if (ret != 0)
134                                 error_exit ("write failed\n");
135
136                         ret = read (fd, (unsigned char *) inner_table, PAGESIZE);
137                         if (ret != 0)
138                                 error_exit ("read failed\n");
139                         for (j = 0; j < NUM_ENTRIES; ++j) {
140                                 i_pte = (struct pt_e *) &(inner_table[j]);
141                                 if (i_pte->present)
142                                         fr_count++;
143                         }
144                 }
145         }
146
147         return (fr_count);
148 }
149
150
151
152
```

```
153 void
154 dump_region (unsigned char *buf, int len, int type, unsigned int start)
155 {
156         unsigned int count  = 0;
157         int i;
158 
159         count = start;
160 
161         for (i = 0; i < len; ++i) {
162                 if (type == HEX_DUMP) {
163                         if (count % 32 == 0)
164                                 printf ("%08x: ", count);
165                         printf ("%02x", buf[i]);
166                         fflush (stdout);
167                         if ( ( (count + 1) % 4) == 0)
168                                 printf (" ");
169                         if ( ( (count + 1) % 32) == 0)
170                                 printf ("\n");
171                         count++;
172                 }
173                 else if (type == ASCI_DUMP) {
174                         printf ("%c", (char) buf[i]);
175                         fflush (stdout);
176                         count++;
177                 }
178         }
179 }
180 
181 void dump_frame (char *buf, int size, unsigned int start_virt_addr)
182 {
183         printf ("dumping frame \n");
184         dump_region (buf, PAGESIZE, HEX_DUMP, start_virt_addr);
185 }
186 
187 
188 int
189 main(int argc, char **argv)
190 {
191         unsigned char buf[PAGESIZE];  /* page size */
192         int fd;
193         int pid;
194         int index;
195         int ret;
196         int count, i;
197         unsigned int p_mem;
198         char comm[128];
199         struct order_struct  order;
200         unsigned int st_vaddr;
201 
202         fd = open (DEVICE_NAME, O_RDWR);
203         if (fd < 0) {
204                 printf ("can not open\n");
205                 exit (1);
206         }
207 
208         if (argc < 3) {
```

```
209                 printf ("usage: \n");
210                 printf ("  info <pid> -m: give RAM usage\n");
211                 printf ("  info <pid> -pt -o: print outer page table\n");
212                 printf ("  info <pid> -pt -i <index>: print inner page table <index>\n");
213                 printf ("  info <pid>  -f <index1> <index2> \n");
214                 exit (1);
215         }
216
217         pid = atoi (argv[1]);
218
219         strcpy (comm, argv[2]);
220
221         if (strcmp(comm, "-m") == 0) {
222                 p_mem = find_phy_memsize (fd, pid);
223                 printf ("RAM usage: %d frames, %d KBytes\n",  p_mem, p_mem * 4 );
224         }
225         else if (strcmp(comm, "-pt") == 0) {
226                 if (strcmp (argv[3], "-o") == 0) {
227                         order.code = GET_OUTER_TABLE;
228                         order.p1 = pid;
229                         memcpy ((void *) buf, (void *) &order, sizeof (order));
230                         printf ("calling the write\n");
231
232                         ret = write (fd, buf, sizeof(order));
233                         if (ret != 0)
234                                 error_exit ("write failed\n");
235
236                         ret = read (fd, buf, PAGESIZE);
237                         if (ret != 0)
238                                 error_exit("read failed\n");
239                         print_table (buf, INDEX_REFERRING_OUTER_TABLE);
240                 }
241                 else if (strcmp (argv[3], "-i") == 0) {
242                         index = atoi (argv[4]);
243
244                         order.code = GET_INNER_TABLE;
245                         order.p1 = pid;
246                         order.p2 = index;
247                         memcpy ((void *) buf, (void *) &order, sizeof (order));
248
249                         ret = write (fd, buf, sizeof(order));
250                         if (ret != 0)
251                                 error_exit ("write failed\n");
252                         ret = read (fd, buf, PAGESIZE);
253                         if (ret != 0)
254                                 error_exit("read failed\n");
255                         print_table (buf, index);
256                 }
257         }
258         else if (strcmp(comm, "-f") == 0) {
259                 order.code = GET_FRAME;
260                 order.p1 = pid;
261                 order.p2 = atoi (argv[3]);
262                 order.p3 = atoi (argv[4]);
263                 st_vaddr = (order.p2 * 4 * ONEMEGABYTE) + (order.p3 * 4096);
264
```

```
265                 memcpy ( (void *) buf, (void *) &order, sizeof (order));
266
267                 ret = write (fd, buf, sizeof(order));
268                 ret = read (fd, buf, PAGESIZE);
269
270                 dump_frame (buf, PAGESIZE, st_vaddr);
271
272         }
273
274         close (fd);
275 }
```

Below is a Makefile that will compile the tool and the driver and an application whose memory usage will be analyzed using the tool.

```
1  obj-m += driver.o
2
3
4  all:
5          make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
6          gcc -o tool -g tool.c
7          gcc -o app app.c
8
9  clean:
10         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
11         rm -fr info *~ app
```

Below is the sample application program (app.c) whose page tables will be retrieved and printer using the tool. You can monitor (retrieve and print) the pages and page tables of other processes created in your system. Not just this application. Use the 'ps aux' command to see the list of processes created in your machine.

```
1  /* -*- linux-c -*- */
2  /* $Id: app.c,v 1.1 2010/04/14 12:16:40 korpe Exp korpe $ */
3
4  #include <stdio.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <malloc.h>
8
9
10 int x = 32;
11 int y = 100;
12 int a[5] = {1,2,3,4,5};
13 int z = 200;
14
15 int
16 main()
17 {
18
```

```
19
20          int i;
21
22          for (i = 0; i < 5; ++i)
23                  if (malloc (4096) == NULL) {
24                          printf ("can not alloc\n");
25                          exit (1);
26                  }
27
28          printf ("finished allocating memory\n");
29
30
31          printf ("data segment end = 0x%x\n", (unsigned int) sbrk(0) );
32
33          while (1) {
34                  sleep (10);
35          }
36
37          return 0;
38  }
```

And finally, below is our driver code (driver.c). This is a kernel level code. Note that this a virtual driver. There is no hardware device that it is driving. We just call it a driver since it is written like a driver. A hardware driver has similar functions (like read, write) implemented.

```
1   /* -*- linux-c  -*- */
2   /* $Id: driver.c,v 1.1 2010/04/14 12:16:40 korpe Exp korpe $ */
3
4   #include <linux/kernel.h>
5   #include <linux/module.h>
6   #include <linux/fs.h>
7   #include <asm/uaccess.h>/* for put_user */
8   #include <linux/highmem.h>
9   #include <linux/init.h>
10  #include <linux/sched.h>
11  #include <linux/mm.h>
12  #include <asm/page.h>
13
14
15  #define DEVICE_NAME "mydevice"/* Dev name as it appears in /proc/devices   */
16  #define MAJOR_NUM 210
17
18  #define PAGESIZE       4096
19  #define ENTRYSIZEBYTES    4
20  #define ENTRYSIZEBITS    32
21  #define NUMENTRIES     (PAGESIZE/ENTRYSIZEBYTES)
22
23
24  struct order_struct  {
25          int code;
26          int p1;
27          int p2;
```

```
28          int p3;
29  };
30  #define INVALID_CODE -1
31  #define GET_FRAME 1
32  #define GET_OUTER_TABLE 2
33  #define GET_INNER_TABLE 3
34  /* .... */
35
36
37
38  static int     device_open(struct inode *, struct file *);
39  static int     device_release(struct inode *, struct file *);
40  static ssize_t device_read(struct file *, char __user *, size_t, loff_t *);
41  static ssize_t device_write(struct file *, const char __user *, size_t, loff_t *);
42
43  static int Major;  /* Major number assigned to our device driver */
44  static struct order_struct  order;
45
46  struct file_operations device_fops = {
47          .open = device_open,
48          .release = device_release,
49          .read = device_read,
50          .write = device_write
51  };
52
53
54
55  /*
56     A Note: Inside kernel, you can dynamically allocate some amount of
57     memory using kmalloc() if you need and wish.
58  */
59
60
61  int init_module(void)
62  {
63          int retval;
64
65          retval = register_chrdev (MAJOR_NUM, DEVICE_NAME, &device_fops);
66
67          if (retval < 0) {
68                  printk ("can not register\n");
69                  return retval;
70          }
71          else {
72                  printk ("registered the module, retval=%d\n", retval);
73
74          }
75
76          order.code = INVALID_CODE;
77          order.p1 = 0;
78          order.p2 = 0;
79          order.p3 = 0;
80
81          return 0;
82  }
83
```

```
84
85
86  void cleanup_module(void)
87  {
88          printk ("cleanup called\n");
89
90          unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
91  }
92
93
94  static int device_open (struct inode *ino, struct file *filp)
95  {
96          printk ("\n device open function called \n");
97          return 0;
98
99  }
100
101 static int device_release (struct inode *ino, struct file *filp)
102 {
103         printk ("\n device close function \n");
104         return 0;
105 }
106
107
108 static ssize_t
109 device_read(struct file *filp, char  *buf, size_t len, loff_t * offset)
110 {
111         struct task_struct *task;
112         unsigned int framenum, framenum2;
113         unsigned int va, va2;
114         unsigned int entry, entry2;
115         pte_t *pagetable_ptr;
116
117
118         printk ("\n device read function \n");
119         printk ("%d\n", order.code);
120
121         task = &init_task;
122         do {
123                 if (task->pid == order.p1)
124                         break;
125         }  while ( (task = next_task(task)) != &init_task );
126
127         if (task == &init_task)
128                 return -1;
129
130
131         if (order.code == GET_OUTER_TABLE) {
132                 va = (unsigned int) (task->mm->pgd);
133                 copy_to_user (  (void *) buf, (void *) va, (size_t) PAGESIZE);
134                 return 0;
135
136         }
137         else if (order.code == GET_INNER_TABLE) {
138                 if ( (order.p2 >= 0) && (order.p2 <= (NUMENTRIES-1) ))
139                 {
```

```
140
141                     entry = pgd_val(task->mm->pgd[order.p2]);
142
143
144                     if ( (entry % 2) == 1) {
145                             framenum = entry >> 12;
146                             va  = (unsigned int) kmap ( mem_map + framenum);
147                             copy_to_user (  (void *) buf,
148                                             (void *) va,
149                                             (size_t) PAGESIZE);
150                             kunmap ( mem_map + framenum);
151                             return 0;
152                     }
153                     else {
154                             return -1;
155                     }
156             }
157             else {
158                     return -1;
159             }
160     } else if (order.code == GET_FRAME) {
161             if ( (order.p2 >= 0) && (order.p2 <= (NUMENTRIES-1) ) )
162             {
163
164                     entry = pgd_val(task->mm->pgd[order.p2]);
165
166                     if ( (entry % 2) == 1) {
167                             framenum = entry >> 12;
168                             va  = (unsigned int) kmap ( mem_map + framenum);
169                             pagetable_ptr =  (pte_t *) va;
170
171                             entry2 = pte_val (pagetable_ptr[order.p3]);
172
173                             if  ( (entry2 % 2 ) == 1) {
174                                     framenum2 = entry2 >> 12;
175                                     va2 = (unsigned int)
176                                             kmap ( mem_map + framenum2);
177
178                                     copy_to_user (  (void *) buf,
179                                                     (void *) va2,
180                                                     (size_t) PAGESIZE);
181                                     kunmap (mem_map + framenum2);
182                                     kunmap (mem_map + framenum);
183                                     return 0;
184                             }
185                             else {
186                                     kunmap (mem_map + framenum);
187                                     return -1;
188                             }
189                     }
190                     else {
191                             return -1;
192                     }
193             }
194             else {
195                     return -1;
```

```
196                 }
197         }
198         return 0;
199 }
200
201
202 static ssize_t
203 device_write  (struct file *filp, const char *buf, size_t len, loff_t * offset)
204 {
205         printk ("\n device write function \n");
206
207         copy_from_user ( (void *) &order,  (void *) buf, len);
208         printk ("order code = %d\n", order.code);
209
210         return 0;
211 }
212
213
214
```

We will now compile and run our tool and driver.

To compile everything, we just type make.

Then we need to load the driver (i.e. module). But before that we need to create a special device file in the /dev directory of our filesystem. This is called creating a device node. The association between our tool and driver will be established with the help of that device file. Basically, our tool will open that device file to operate on the corresponding virtual device using the driver that we have written. That means the driver will be accessed and triggered to execute using this device file. The file is actually a special file corresponding to a virtual device. There is no content stored on disk for the file. That means, when we open the device file (by calling the open system call), for example, in our tool, we will invoke (trigger to execute) the open() routine implemented in our driver. As another example, when we call the read() or write() system call on the opened file in our tool, we trigger the execution of our read() or write() routine in our driver. In this way, the operations that are implemented in our driver (open, read, write, close operations) are called from the tool via using the device file that is created. The kernel ties these system calls that we call from our tool and the respective routines that we implemented in our driver.

To create such a special device file, we first need to select a major number that is unused. Type `cat/proc/devices`. It will give you the major numbers currently used. Also check the major numbers used by the device special files sitting on the `/dev/` directory. For that you type `ls -al /dev/*`. You will see the assigned major numbers at the 5th column of the output. By checking these two outputs (output of 'cat /proc/devices' and 'cat /dev/*'), select a major number (between 0 and 255) that is unused. For example, in our machine, 210 was not used. Actually, if you look to the driver code above, you will see that 210 is hardcoded there as the value of the MAJOR_NUM macro. If you select

a different number than 210, you need to change the macro value in driver.c as well. Be careful about this.

Decide on a name for your special virtual device that you will create in /dev. We selected, for example, a name "mydevice". If you look to the code of the driver, and code of our tool, you will see that name appearing in those C files. If you select another name, make sure you change the respective names in the driver.c and tool.c.

Now we can create a special device node in the /dev directory. For that we type:

```
sudo mknod /dev/mydevice c 210 0
```

Here, c denotes that this is a character-oriented device. That means that data flowing into and out of the device is just a stream of bytes (not blocks as in the block-oriented devices like hard-disks). Here, 0 is the minor number. 210 is the major number that we selected.

You can change into the directory /dev and see that the file mydevice has been created.

```
$ ls -la /dev/* | grep my
crw-r--r--  1 root root    210,   0 2010-04-13 14:54 /dev/mydevice
```

Now, we need to load our driver. After compilation, if successful, we should have obtained a file called `driver.ko`. This is the module/driver code that has to be loaded and become part of the kernel. We load a module using the `insmod` command. To load our module/driver, we type:

```
sudo insmod driver.ko
```

If we want to remove the driver/module, we type:

```
sudo rmmod driver
```

Then you can reload it again by using insmod. You can load and remove a module as many times as you wish. You can load and remove a module whenever you wish. Of course, if the module/driver is used at that moment by someone (i.e. it is opened and not closed), the system may not allow you to remove it until that someone (that process) releases the module. The module must be loaded (and not removed) to be used by a process.

Now, after you have loaded the module using insmod, you applications/processes may start using it. To check if the module is really loaded, type `cat /proc/modules` and search for the name of your driver in the output. If you see it, that means it is loaded and became part of the kernel. The output will also tell you where in kernel it is sitting.

```
$ cat /proc/modules | grep driver
```

```
driver 2444 0 - Live 0xf80ad000 (P)
```

Now, we can run our memory tool (`tool`) that will use the driver to get information about the page tables and frames of a process. The tool expects some parameters. See what it expects from its source code (tool.c).

Before running our memory tool, let us first start a sample application whose page table will be printed using our tool and driver. We have such a sample application called `app`. You can examine its source code. It is a very simple application. Now let us start that by typing:

```
./app &
```

We want that to be running in the background, therefore we use the '&' sign.

To find out the pid our our application, we type "ps aux" and look for the name "app" in the output. We will see the pid assigned to the process in the output. It was 3587 in our case.

```
$ps aux
...
korpe     3587  0.0  0.0   1664   420 pts/1    S    23:11   0:00 ./app
...
```

Now we can run our tool to see, for example, the content of the outer page table of the process. For that we type:

```
sudo ./tool 3587 -pt -o
```

We can see such an output. Only the entries have the corresponding inner tables in RAM are printed out. That means only the entries for which the Present flag is TRUE are printed out.

```
1 calling the write
2 pte[   0]: it=1      f=7de06  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
3 pte[   1]: it=2      f=7da7b  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
4 pte[   2]: it=3      f=7dc32  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
5 pte[  32]: it=4      f=7dc3b  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
6 pte[  37]: it=5      f=7dc30  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
7 pte[ 734]: it=6      f=7dc31  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
8 pte[ 767]: it=7      f=7dc2f  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
9 pte[ 768]: kt=1      f=8a9    av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
```

```
pte[ 769]: kt=2      f=373f2  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 770]: kt=3      f=800    av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 771]: kt=4      f=c00    av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 772]: kt=5      f=1000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 773]: kt=6      f=1400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 774]: kt=7      f=1800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 775]: kt=8      f=1c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 776]: kt=9      f=2000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 777]: kt=10     f=2400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 778]: kt=11     f=2800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 779]: kt=12     f=2c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 780]: kt=13     f=3000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 781]: kt=14     f=3400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 782]: kt=15     f=3800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 783]: kt=16     f=3c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 784]: kt=17     f=4000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 785]: kt=18     f=4400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 786]: kt=19     f=4800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 787]: kt=20     f=4c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 788]: kt=21     f=5000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 789]: kt=22     f=5400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 790]: kt=23     f=5800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 791]: kt=24     f=5c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 792]: kt=25     f=6000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 793]: kt=26     f=6400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 794]: kt=27     f=6800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 795]: kt=28     f=6c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 796]: kt=29     f=7000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 797]: kt=30     f=7400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 798]: kt=31     f=7800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 799]: kt=32     f=7c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 800]: kt=33     f=8000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 801]: kt=34     f=8400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 802]: kt=35     f=8800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 803]: kt=36     f=8c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 804]: kt=37     f=9000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 805]: kt=38     f=9400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 806]: kt=39     f=9800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 807]: kt=40     f=9c00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 808]: kt=41     f=a000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 809]: kt=42     f=a400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 810]: kt=43     f=a800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 811]: kt=44     f=ac00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 812]: kt=45     f=b000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 813]: kt=46     f=b400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 814]: kt=47     f=b800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 815]: kt=48     f=bc00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 816]: kt=49     f=c000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 817]: kt=50     f=c400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 818]: kt=51     f=c800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 819]: kt=52     f=cc00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 820]: kt=53     f=d000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 821]: kt=54     f=d400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 822]: kt=55     f=d800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 823]: kt=56     f=dc00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 824]: kt=57     f=e000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
```

```
pte[ 825]: kt=58     f=e400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 826]: kt=59     f=e800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 827]: kt=60     f=ec00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 828]: kt=61     f=f000   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 829]: kt=62     f=f400   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 830]: kt=63     f=f800   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 831]: kt=64     f=fc00   av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 832]: kt=65     f=10000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 833]: kt=66     f=10400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 834]: kt=67     f=10800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 835]: kt=68     f=10c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 836]: kt=69     f=11000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 837]: kt=70     f=11400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 838]: kt=71     f=11800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 839]: kt=72     f=11c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 840]: kt=73     f=12000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 841]: kt=74     f=12400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 842]: kt=75     f=12800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 843]: kt=76     f=12c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 844]: kt=77     f=13000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 845]: kt=78     f=13400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 846]: kt=79     f=13800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 847]: kt=80     f=13c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 848]: kt=81     f=14000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 849]: kt=82     f=14400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 850]: kt=83     f=14800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 851]: kt=84     f=14c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 852]: kt=85     f=15000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 853]: kt=86     f=15400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 854]: kt=87     f=15800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 855]: kt=88     f=15c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 856]: kt=89     f=16000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 857]: kt=90     f=16400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 858]: kt=91     f=16800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 859]: kt=92     f=16c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 860]: kt=93     f=17000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 861]: kt=94     f=17400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 862]: kt=95     f=17800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 863]: kt=96     f=17c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 864]: kt=97     f=18000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 865]: kt=98     f=18400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 866]: kt=99     f=18800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 867]: kt=100    f=18c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 868]: kt=101    f=19000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 869]: kt=102    f=19400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 870]: kt=103    f=19800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 871]: kt=104    f=19c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 872]: kt=105    f=1a000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 873]: kt=106    f=1a400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 874]: kt=107    f=1a800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 875]: kt=108    f=1ac00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 876]: kt=109    f=1b000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 877]: kt=110    f=1b400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 878]: kt=111    f=1b800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 879]: kt=112    f=1bc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 880]: kt=113    f=1c000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
```

```
pte[ 881]: kt=114    f=1c400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 882]: kt=115    f=1c800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 883]: kt=116    f=1cc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 884]: kt=117    f=1d000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 885]: kt=118    f=1d400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 886]: kt=119    f=1d800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 887]: kt=120    f=1dc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 888]: kt=121    f=1e000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 889]: kt=122    f=1e400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 890]: kt=123    f=1e800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 891]: kt=124    f=1ec00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 892]: kt=125    f=1f000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 893]: kt=126    f=1f400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 894]: kt=127    f=1f800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 895]: kt=128    f=1fc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 896]: kt=129    f=20000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 897]: kt=130    f=20400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 898]: kt=131    f=20800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 899]: kt=132    f=20c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 900]: kt=133    f=21000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 901]: kt=134    f=21400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 902]: kt=135    f=21800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 903]: kt=136    f=21c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 904]: kt=137    f=22000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 905]: kt=138    f=22400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 906]: kt=139    f=22800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 907]: kt=140    f=22c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 908]: kt=141    f=23000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 909]: kt=142    f=23400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 910]: kt=143    f=23800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 911]: kt=144    f=23c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 912]: kt=145    f=24000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 913]: kt=146    f=24400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 914]: kt=147    f=24800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 915]: kt=148    f=24c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 916]: kt=149    f=25000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 917]: kt=150    f=25400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 918]: kt=151    f=25800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 919]: kt=152    f=25c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 920]: kt=153    f=26000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 921]: kt=154    f=26400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 922]: kt=155    f=26800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 923]: kt=156    f=26c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 924]: kt=157    f=27000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 925]: kt=158    f=27400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 926]: kt=159    f=27800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 927]: kt=160    f=27c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 928]: kt=161    f=28000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 929]: kt=162    f=28400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 930]: kt=163    f=28800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 931]: kt=164    f=28c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 932]: kt=165    f=29000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 933]: kt=166    f=29400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 934]: kt=167    f=29800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 935]: kt=168    f=29c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 936]: kt=169    f=2a000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
```

```
pte[ 937]: kt=170    f=2a400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 938]: kt=171    f=2a800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 939]: kt=172    f=2ac00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 940]: kt=173    f=2b000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 941]: kt=174    f=2b400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 942]: kt=175    f=2b800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 943]: kt=176    f=2bc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 944]: kt=177    f=2c000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 945]: kt=178    f=2c400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 946]: kt=179    f=2c800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 947]: kt=180    f=2cc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 948]: kt=181    f=2d000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 949]: kt=182    f=2d400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 950]: kt=183    f=2d800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 951]: kt=184    f=2dc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 952]: kt=185    f=2e000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 953]: kt=186    f=2e400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 954]: kt=187    f=2e800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 955]: kt=188    f=2ec00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 956]: kt=189    f=2f000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 957]: kt=190    f=2f400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 958]: kt=191    f=2f800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 959]: kt=192    f=2fc00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 960]: kt=193    f=30000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 961]: kt=194    f=30400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 962]: kt=195    f=30800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 963]: kt=196    f=30c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 964]: kt=197    f=31000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 965]: kt=198    f=31400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 966]: kt=199    f=31800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 967]: kt=200    f=31c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 968]: kt=201    f=32000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 969]: kt=202    f=32400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 970]: kt=203    f=32800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 971]: kt=204    f=32c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 972]: kt=205    f=33000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 973]: kt=206    f=33400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 974]: kt=207    f=33800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 975]: kt=208    f=33c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 976]: kt=209    f=34000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 977]: kt=210    f=34400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 978]: kt=211    f=34800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 979]: kt=212    f=34c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 980]: kt=213    f=35000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 981]: kt=214    f=35400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 982]: kt=215    f=35800  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 983]: kt=216    f=35c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 984]: kt=217    f=36000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 985]: kt=218    f=36400  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 986]: kt=219    f=3691d  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 987]: kt=220    f=36c00  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 988]: kt=221    f=37000  av=0 g=1 ps=1 d=1 a=1 cd=0 wt=0 us=0 rw=1 p=1
pte[ 989]: kt=222    f=7      av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
pte[ 991]: kt=223    f=37005  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
pte[ 992]: kt=224    f=37010  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
pte[ 993]: kt=225    f=36b66  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
```

```
234 pte[ 994]: kt=226    f=36538  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
235 pte[ 995]: kt=227    f=36baf  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
236 pte[ 996]: kt=228    f=368df  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
237 pte[ 997]: kt=229    f=3694b  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
238 pte[ 998]: kt=230    f=367c4  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
239 pte[ 999]: kt=231    f=36553  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
240 pte[1000]: kt=232    f=369cd  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
241 pte[1001]: kt=233    f=34c56  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
242 pte[1002]: kt=234    f=367e8  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
243 pte[1003]: kt=235    f=367a1  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
244 pte[1004]: kt=236    f=367ef  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
245 pte[1005]: kt=237    f=369c9  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
246 pte[1006]: kt=238    f=369c8  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
247 pte[1007]: kt=239    f=36ac9  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
248 pte[1008]: kt=240    f=34e0a  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
249 pte[1009]: kt=241    f=367d7  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
250 pte[1010]: kt=242    f=35fb5  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
251 pte[1011]: kt=243    f=34e08  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
252 pte[1012]: kt=244    f=34e1e  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
253 pte[1013]: kt=245    f=34e92  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
254 pte[1014]: kt=246    f=35f92  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
255 pte[1015]: kt=247    f=34e36  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
256 pte[1022]: kt=248    f=13     av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
257 pte[1023]: kt=249    f=81d    av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
```

Note that all the virtual addresses used by a process in a 32 bit machine is between 0x00000000 and 0xc0000000 (the first 3 GB of the virtual address space). The virtual addresses between 0xc0000000 and 0xffffffff (the last 1 GB of the virtual address space) are used by the kernel. Note also that kernel code may reference an address falling into range (0x00000000, 0xc0000000), but this means that the kernel is trying to access appliction address space (for example, an application variable). This is OK; kernel can do that. But all data and instructions belonging to kernel space has to be addressed with virtual addresses in range (0xc0000000, 0xffffffff).

Therefore, in the outer page table, we see that only the first 768 entries are used to map process pages to its allocated frames. The remaining entries are used to map the kernel pages. Hence a process is only using the first 768 (out of 1024 entries) entries in the outer page table. However, in an inner page table pointed by such an entry of the outer page table, of course, all the entries are used by the process.

The output shows that in the outer table, the entries with indices 0, 1, 2, 32, 37, 734, 767 are used by the process and the corresponding inner tables are in RAM now. Hence there are 7 inner tables loaded into memory at the moment. We can access the content of one them, lets say the inner table with index 32. For that we type:

```
sudo ./tool 3587 -pt -i 32
```

We get the following output. Only the entries corresponding to pages that are loaded into memory are printed. That means the entries whose Present flag is TRUE are printed out.

```
pte[  72]: pg=32840  f=5a6c5  av=0 g=0 ps=0 d=0 a=1 cd=0 wt=0 us=1 rw=0 p=1
pte[  73]: pg=32841  f=64af9  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=0 p=1
pte[  74]: pg=32842  f=5d292  av=0 g=0 ps=0 d=1 a=1 cd=0 wt=0 us=1 rw=1 p=1
```

The output says that 3 pages of the process pointed by this inner table are loaded into memory at the moment. These 3 pages are loaded into 3 different frames that does not have to be adjascent. The output tells which frames are these. The frames of RAM into which those pages are loaded are (number are in hex): 5a6c5, 64af9, 5d292.

Lets us now dump the frame 5d292. The frame is reached using the index1=32, index2=74 (hence it is the physical frame where virtual page $32 * 1024 + 74$ is loaded). Hence, we can invoke our tool with those values of the indices. We type the following to get the frame content of frame 5d292. Note that the content of frame 5d292 is the virtual page that has the pager number: $32 * 1024 + 74$. That virtual page starts at virtual address $32 * (1024 * 4096) + (74 * 4096)$.

```
sudo ./tool 3587 -f 32 74
```

The output that we get is the following. It is the frame content in hex (there are 4096 byte vaues printed to the screen).

```
dumping frame
0804a000: b2830408 701a5500 30f55f00 501f5800 10a85a00 00175d00 10875900 22840408
0804a020: 00000000 00000000 20000000 64000000 01000000 02000000 03000000 04000000
0804a040: 05000000 c8000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a060: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a080: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a0a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a0c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a0e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a100: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a120: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a140: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a160: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a180: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a1a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a1c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a1e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a200: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a220: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a240: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a260: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a280: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a2a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a2c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a2e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a300: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a320: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a340: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
0804a360: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a380: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a3a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a3c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a3e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a400: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a420: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a440: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a460: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a480: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a4a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a4c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a4e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a500: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a520: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a540: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a560: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a580: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a5a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a5c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a5e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a600: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a620: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a640: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a660: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a680: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a6a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a6c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a6e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a700: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a720: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a740: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a760: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a780: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a7a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a7c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a7e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a800: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a820: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a840: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a860: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a880: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a8a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a8c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a8e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a900: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a920: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a940: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a960: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a980: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a9a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a9c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804a9e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aa00: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aa20: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aa40: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
0804aa60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aa80: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aaa0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aac0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aae0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ab00: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ab20: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ab40: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ab60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ab80: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aba0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804abc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804abe0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ac00: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ac20: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ac40: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ac60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ac80: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aca0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804acc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ace0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ad00: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ad20: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ad40: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ad60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ad80: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ada0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804adc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ade0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ae00: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ae20: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ae40: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ae60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804ae80: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aea0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aec0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804aee0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804af00: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804af20: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804af40: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804af60: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804af80: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804afa0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804afc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0804afe0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```