UNIX Systems Programming

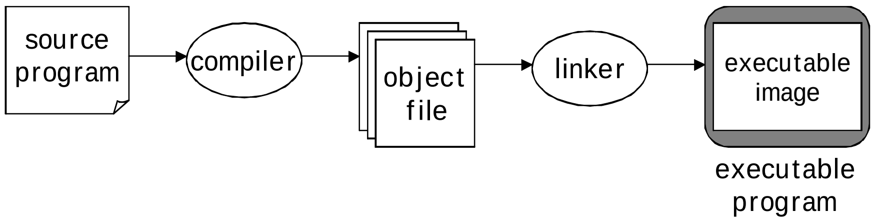
Processes

(Curry, chp.11)

Dr. Kivanç Dinçer
CENG-332 Lecture Notes
Spring 2000

Processes and Programs

- A <u>process</u> is a basic active entity in most O/S models.
  - an instance of a program whose execution has started but has not yet terminated.
  - each instance has its own address space and execution state.



A process is a program in execution.

Program to Process

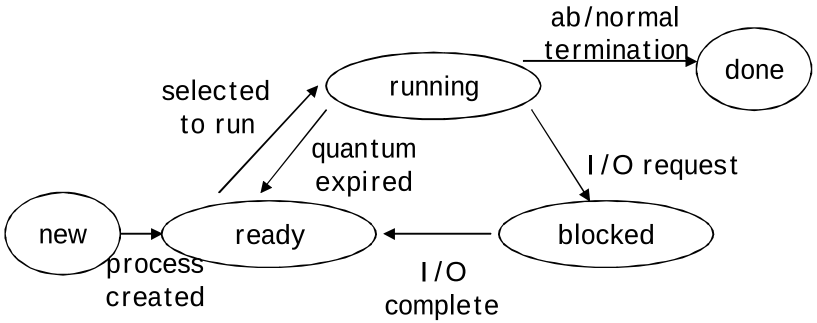
- When does a program become a process?
  - O/S (loader) reads the program into memory
    - assigns a process ID
    - assigns a process state (i.e.,execution status of an individual process)
    - determines required system resources:
      - CPU, memory, user and system stacks, file handles, I/O devices

Process Table

- The O/S maintains info about each process in a process table.
  - Entries of this table are called <u>process control blocks</u> (PCB) and contain information about:
    - process state: registers, stack pointer, PC, process id, etc.
    - memory state: memory areas used by the process
    - resource state: files, etc.

The O/S keeps tracks of the process Ids and corresponding process states and uses the information to allocate and manage resources for the system.

Process States



Running: executing instructions.

Ready : waiting to be assigned to a processor.

Blocked: waiting for some event to occur.

Context Switch

Context switch: the act of removing one process from the running state and replacing it with another.

Context of a process: information that is needed about the process and its environment in order to restart it after a context switch.

- E.g., executable, stack, registers, program counter, memory used for static and dynamically allocated variables.
- all info kept in PCB

Process Operations

- create a process
- destroy a process
- run a process
- suspend a process
- get process information
- set process information

Process Creation

- A new process will be created using a new one:
  - synchronous: the new one must complete execution before the old one can resume
  - asynchronous: the new process is created asynchronously, then the two processes may be run in pseudo-parallel.

Parent: When a new process is created, it may use the old one as "parent."

- No parent exists in Windows NT.

Spawning a new process

In general, *spawning* a (new) process should involve:

a. creating the process
b. setting the process' context
c. allocating resources to the process
d. loading memory space with program to execute
e. starting execution of program

Note: a-d are one step in UNIX, d-e constitute another step.

system

```
#include <stdlib.h>
int system(char *command)
```

creates a new synchronous process.

fork

```
#include <sys/types.h>
pid_t fork(void)
```

creates a new asynchronous process.

- splits the current process into two almost identical copies.
  - new process is the *child*
  - process initiating the *fork()* is the *parent*
- A PID of 0 is returned to child, and PID of child is returned to parent.

Child and parent have same ..

a. file descriptors (e.g. standard input, standard output)
b. execution priority
c. memory image (though child's is a copy)
d. register contents (e.g. PC value!)
e. signal handling
f. etc.

result of c and d is:
both child and parent will be executing (at least initially) the **same program at the same point** (i.e. machine instructions after the call to *fork()*.

Child and parent have different ...

- PID and PPID
- return value from *fork()*
- child gets 0
- parent gets child's PID
- typical coding logic:

```
if( (result=fork()) == 0 ) {/*child code*/ ... }
else if( result > 0 ) { /* parent code */ . . . }
else { /* error */ . . . }
```

- executing new program
- memory image of parent and child are initially the same, until one (typically the child's) is overwritten by a new memory image (copy-on-write semantics)

exec after fork

- It is common to replace one of these processes (usually the child) so that it uses a different program.
  - exec overlays the image of the calling process with the image of a new program.
  - *exec* does not create a new process, and other than the process' memory image, nearly every other attribute of the process' context remains the same
  - if exec succeeds, it never returns.

Ex: suppose that a process creates a file that it wants printed
  - it does not have access to printer device, but **`lpr`** has.

- *exec* overlays (replaces) the address space of the calling process with that of a new program
- six variants (e.g. *execl()*, *execve()*, *execl()*) having different arguments and performing different preprocessing
  - (the six are collectively referred to as *exec*)
- on successful "return" from *exec*, the process resumes execution at the entry point of the new program

Six versions of exec

- **execl**(char *pathname, char *arg0, ...,(char*) 0);
- **execv**(char *pathname, char *argv[]);
- **execle**(char *pathname, char *arg0, ..., (char*) 0, char *envp[]);
- **execve**(char *pathname, char *argv[], char *envp[]);
- **execlp**(char *filename, char *arg0, ..., (char*) 0);
- **execvp**(char *filename, char *argv[]);

Process Suspension

- wait

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* status)
```

  – Waiting for a child
  – If there is more than one child, wait() returns on termination of any children
  – Ex: suppose that parent wants to delete the temp file printed by child after printing.
- sleep

```
unsigned int sleep(seconds)
```

  – A process may suspend for a period of time using the sleep command.

- A process that calls wait() can:
  – block (if all of its children are still running)
  – return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
  – return immediately with an error (if it doesn't have any child processes)

Example: `wait.c`

```
#include <sys/types.h>
#include <sys/wait.h>
void main( void )
{
    int status;

    if( fork() == 0 ) exit( 7 ); /* normal exit */
    wait( &status ); prExit( status );

    if( fork() == 0 ) abort(); /* generates SIGABRT */
    wait( &status ); prExit( status );

    if( fork() == 0 ) status /= 0; /* generates SIGFPE */
    wait( &status ); prExit( status );
}
```

SIGABRT: Abort.
SIGFPE: Arithmetic exception.

- **waitpid**
  – can be used to wait for a specific child pid
  – waitpid also has an option to block or not to block

```
pid_t waitpid( pid, &status, option );
```

  – `pid == -1` waits for any child
  – `option == NOHANG` non-blocking
    `option == 0` blocking
- `waitpid(-1, &status, 0)` equivalent to `wait(&status)`

Process Removal

- exit

```
int exit(status)_
```

  – when a process executes the “exit” command it terminates
  – performs various cleanup operations, such as flushing output buffers
- _exit
  – calls *exit()* kernel function which causes the termination of the calling process
  – The exit status is by convention:
    - 0: success
    - nonzero: error, with value being error code

- kill
  – one process can send simple messages to another using the “kill” command

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig)
```

- signal
  – a process can catch certain signals by installing a signal handler which is a function invoked when the signal arrives.

Orphan Processes

- a process whose parent is the init process (pid 1) because its original parent died before it did
- **Every normal process is a child of some parent, a terminating process sends its parent a SIGCHLD signal and waits for its termination code status to be accepted**
  – **The C shell stores the termination code of the last command in the local shell variable status**

Zombie processes

- a process that is “waiting” for its parent to accept its return code
- a parent accepts a child's return code by executing wait()

A terminating process may be a (multiple) parent; the kernel ensures all of its children are orphaned and adopted by init

Process Environment

- A process has an entry in a table of processes. This table contains all sort of information:
- Process ID

```
#include <sys/types.h>
pid_t getpid(void)
```

- User ID

```
#include <sys/types.h>
uid_t getuid(void)
```

- User name

```
char*  getlogin(void)
```

- Current directory

```
char* getcwd(char* dir, int size)
```


Error Handling

- **All system calls return -1 if an error occurs**
  - **errno:** global variable that holds the numeric code of the last system call
  - **perror():** a subroutine that describes system call errors
    - Every process has errno initialized to zero at process creation time
    - When a system call error occurs, errno is set
    - See /usr/include/sys/errno.h
    - A successful system call never affects the current value of errno
    - An unsuccessful system call always overwrites the current value of errno

- **perror()**

```
void perror( char *str )
```

  - perror displays str, then a colon (:), then an english description of the last system call error, as defined in the header file /usr/include/sys/errno.h

Protocol:

- check system calls for a return value of -1
- call perror() for an error description during debugging (see example on next slide)

perror() example

```
#include <stdio.h>
#include <errno.h>

int main( void )
{
  int returnVal;
  printf( "x2 before the execlp, pid=%d\n",getpid());
  returnVal = execlp( "nonexistent_file", (char *)0);
   if (returnVal == -1)
      perror( "x2 failed" );
   return( 1 );
}
```

Signals

- Signals are unexpected/unpredictable events:
  - floating point error
  - interval timer expiration (alarm clock)
  - death of a child
  - control-C (termination request)
  - control-Z (suspend request)
- Events are called interrupts
  - When the kernel recognizes such an event, it sends the corresponding process a signal
  - Normal processes may send other processes a signal, with permission (useful for synchronization)

Race conditions

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run
  - This is a situation when using forks: if any code after the fork explicitly or implicitly depends on whether or not the parent or child runs first after the fork
    - A parent process can call wait() for a child to terminate (may block)
    - A child process can wait for the parent to terminate by polling it (wasteful)
  - Standard solution is to use signals