Lecture 25
Systems Programming
Process Control

A *process* is defined as an instance of a program that is currently running. A uni processor system or single core system can still execute multiple processes giving the appearance of a multi-core machine. A call to an executable program spawns a process. For examples, if a mail program is called by n users then n processes or instances are created and executed by the unix system. Many operating systems including windows and unix can execute many processes at the same time. When a program is called, a process is created and a process ID is issued. The process ID is given by the function getpid() defined in <unistd.h>.

The prototype for pid( ) is given by

```
#include <unistd.h>
pid_t getpid(void);
```

In a single core machine, each process takes turns running for a short duration in time interval called a timeslice. The unix command ps can be used to list all current process status in your shell.

- **ps**

```
PID      TTY       TIME   CMD
10150    pts/16    00:00:00 csh
31462    pts/16    00:00:00  ps
```

The command ps lists the process ID (PID), the terminal name(TTY), the amount of time the process has used so far(TIME) and the command it is executing(CMD). Ps command only displays the current user processes. But we can get all the processes with the flag (-a) and in long format with flag (-l)

- **ps –a**
- **ps -l**
- **ps -al**

Information provided by each process may include the following.

PID	The process ID in integer form
PPID	The parent process ID in integer form
STAT	The state of the process
TIME	CPU time used by the process (in seconds)
TT	Control terminal of the process
COMMAND	The user command that started the process

Each process has a process ID that can be obtained using the getpid() a system call declared in unistd.h. Here is an example of using the getpid function.

```
printf("The current process %d \n",getpid());
printf("The parent process is %d \n",getppid());
printf("The owner of this process has uid %d \n",getuid());
sleep(1);
```

Each process has a parent process and the parent process ID can be obtained by the function getppid(). A process can be terminated using the kill command

- **kill -9 pid**

or if the process is in the foreground then it can also be killed by cntrl-c.

Background Processes

A long running program can be controlled by introducing a process running in the background. For example, we can run a C program in the background by typing

> ./a.out &

Background processes continues to run even when you logout. You can check the status of a background process by typing > ps at the login.

Process related Commands

The process related system calls in UNIX include **fork( ), exec( )** [many variations of this], **wait( )** and **exit( )** system calls. Using exec, an executable binary file (eg: a.out) can be converted into a process. An example of using exec is implementing a shell program or a command interpreter. A shell program takes user commands and executes them. We will see an example of a simple shell soon.

Forking new Processes

A new process can be created using the command fork. The fork( ) function creates a child process which is exactly identical to the parent process except for the return value of fork( ). The value zero gets returned to the child and PID gets returned to the parent. An example of using fork( ) is

```
if (fork() == 0) { printf("This is a message from the child\n");}

else { printf("This is a message from the parent\n");}
```

if the fork process is failed, no child process is created and fork returns -1.

```
int PID = fork();
if (PID == -1)  printf("the process creation failed\n");
```

Let us consider the following program that uses the child to compute partial sums and parent to compute the partial products of an array of integers.

```
int A[]={1,2,3,4,5,6};
int sum=0, pdt=1, PID, i;
if ((PID=fork())==0){
   for (i=0;i<6;i++) sum += A[i];
   printf("This is child process computed sum %d \n", sum);
}
if (PID <0) {
    fprintf(stderr,"problem creating a process \n");
}
if (PID >0) {
    for (i=0;i<6;i++) pdt *= A[i];
    printf("The parent process completed the product %d \n", pdt);
}
```

Executing another process

There are many variations of the exec system call that can be used to execute a binary executable file. Exec programs do not return. It replaces the current process with the new process. If the exec command fails, then it returns 1. We will consider some useful forms of exec commands.

execl --- takes the path name of a binary executable as its first argument, the rest of the arguments are the command line arguments ending with a NULL.

Example: `execl("./a.out", NULL)`

execv – takes the path name of a binary executable as its first argument, and an array of arguments as its second argument.

Example:
```
static char* args[] = {“ “, "cat.txt", "test1.txt", NULL};
execv("/bin/cp", args);
```

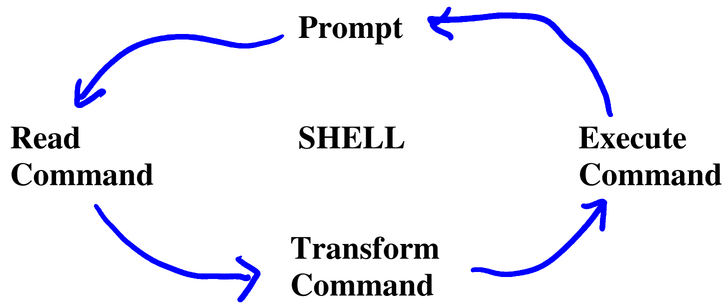
The empty string “ “ can be replaced by the command itself (eg “ls”) or leave it as “ “.

execlp --- same as execl except that we don’t have to give the full path name of the command.

```
execlp("ls", NULL)
```

The UNIX Shell

After login, we interact with unix through a program called “shell”. There are number of shells in use, csh(Berkeley unix C-shell), bsh (bourne Shell) etc. You can find the current shell by typing ps. The shell program that is invoked upon login is called the “login shell”. The shell is a command interpreter as well as a command programming language with many modern programming constructs. We will learn how to write shell programs (or shell scripts) later. Shell takes commands such as ls, cp and calls the correct binaries to execute the command. Shell can also take a binary executable code of your C program, and start a process to execute that code. Shell continues to provide a prompt for the next command until shell is terminated. All shells are programmed in C. The command interpretation cycle of a shell is given as follows.



A simple form of a command looks like this

> command [arg(s)]

Examples: **> ls**
> cp file1 file2

After reading the command, the shell must interpret the command to understand its form. It needs to break down the command and command line arguments (if any). The shell then invokes a child process and calls the child process to execute the command by using one of the exec system calls. The binaries associated with the command are interpreted and executed. Let us now look at a program that implements a simple shell. This shell can only execute commands with no arguments such as **ls** or **who**.

```
int PID;
char cmd[256];
while (1) {
  printf("cmd: ");
  scanf(“%s”,cmd);
  if ( strcmp(cmd,"e")==0) /* loop terminates if type 'e'*/
       exit(0);
 /* creates a new process. Parent gets the process ID. Child gets 0 */
  if ((PID=fork()) > 0)
      wait(NULL);
  else if (PID == 0) /* child process */
     {
      execlp (cmd,cmd,NULL);
```

```
          /* exec cannot return. If so do the following */
        fprintf (stderr, "Cannot execute %s\n", cmd);
          exit(1); /* exec failed */
       }
    else if ( PID == -1)
         {
           fprintf (stderr, "Cannot create a new process\n");
           exit (2);
         }
}
```

Exiting a process

System function exit allows termination of a process. The prototype for exit is

```
#include <stdlib.h>
void exit(int status);
```

The value of the status may be EXIT_SUCCESS(0), EXIT_FAILURE(1) or any other value that is available to a parent process.

Waiting for a Process

wait, waitpid - wait for a child process to stop or terminate

```
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

It returns the PID of the child and the exit status gets placed in status.

```
main() {
  int child_status, pid, pidwait;
  if ((pid = fork()) == 0) {
    printf("This is the child!\n");
  }
  else {
    pidwait = wait(&child_status);
    printf("child %d has terminated\n", pidwait);
  }
  exit();
}
```

Questions about this Lecture

1. Why does not exec return (they have to maintain the same environment variables? See http://www.opengroup.org/onlinepubs/000095399/functions/exec.html
2. Do fork processes share the same variable space? Answer is no. But I need to know why.

Exercises

1. Modify the program (shell.c) so it can execute any shell command with upto 2 arguments. Eg: cp file1 file2 , mv file1 file2, ls –l
2. Write a program that will read a directory containing a set of files, create a new subfolder called “backup” on same folder and copy all the files from directory to the new sub directory “backup”. You are supposed to use exec() with “cp” command. Create a child process for each file that can execute each of the cp commands to copy the file from directory to subfolder “backup”
3. Why would you use fork() to create new processes in a uni-processor machine? Discuss a scenario where this can be useful.