CSc 332 - Operating Systems

Process Management System Calls

September 18, 2020

Process

- A process is basically a single running program.
- Each running process has a unique number - a process identifier, **pid** (an integer value).
- Each process has its own address space.
- Processes are organized hierarchically. Each process has a **parent** process that explicitly arranged to create it. The processes created by a given parent are called its **child** processes
- The C function **getpid()** will return the **pid** of the process.
- A child inherits many of its attributes from the parent process.
- The UNIX command ps will list all current processes running on your machine and will list the **pid.**
- **Remark:** When UNIX is first started, there is only one visible process in the system. This process is called **init** with pid 1. The only way to create a new process in UNIX is to duplicate an existing process, so init is the ancestor of all subsequent processes.

Process Creation

- Each process is named by a process ID number
- A unique process ID is allocated to each process when it is created
- Processes are created with the **fork()** system call (the operation of creating a new process is sometimes called forking a process)
- The fork() system call does not take an argument
- If the fork() system call fails, it will return -1
- If the fork() system call is successful, the process ID of the child process is returned in the parent process and a 0 is returned in the child process
- When a fork() system call is made, the operating system generates a copy of the parent process which becomes the child process

- Some of the specific attributes of the child process that differ from the parent process are:
  - The child process has its own unique process ID
  - The parent process ID of the child process is the process ID of its parent process
  - The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa
- When the lifetime of a process ends, its termination is reported to its parent and all the resources including the PID is freed.

Example of fork() **system call structure**

Here a process P calls fork(), the operating system takes all the data associated with P, makes a brand-new copy of it in memory, and enters a new process Q into the list of current processes. Now both P and Q are on the list of processes, both about to return from the fork() call, and they continue.

The fork() system call returns Q s process ID to P and 0 to Q. This gives the two processes a way of doing different things. Generally, the code for a process looks something like the following:

```
int child = fork();
if(child == 0) {
   //code specifying how the child process Q is to behave
}else {
   //code specifying how the parent process P is to behave
}
```

Process Identification

- You can get the process ID of a process by calling getpid()
- The function getppid() returns the process ID of the parent of the current process
- Your program should include the header files unistd.h and sys/types.h to use these functions

waitpid() **System Call**

- A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions
- The waitpid() system call gives a process a way to wait for a process to stop. It's called as follows.

```
pid = waitpid(child, &status, options);
```

In this case, the operating system will block the calling process until the process with ID child ends

• The options parameter gives ways of modifying the behavior to, for example, not block the calling process. We can just use 0 for options here.

• When the child process ends, the operating system changes the int variable status to represent how child process ended (incorporating the exit code, should the calling process want that information), and it unblocks the calling process, returning the process ID of the process that just stopped.

• If the calling process does not have any child associated with it, **wait** will return immediately with a value of -1

TASK 2

DUE: October 1, 2020, 11:59 PM – 15 Points

Part 1 Write a program children.c, and let the parent process produce two child processes. One prints out "I am child one, my pid is: " PID, and the other prints out "I am child two, my pid is: " PID. Guarantee that the parent terminates after the children terminate (Note, you need to wait for two child processes here). Use the getpid() function to retrieve the PID.

Part 2 Consider the parent process as P. The program consists of fork() system call statements placed at different points in the code to create new processes Q and R. The program also shows three variables: a, b, and pid - with the print out of these variables occurring from various processes. Show the values of pid, a, and b printed by the processes P, Q, and R.

```
//parent P
int a=10, b=25, fq=0, fr=0
fq=fork() // fork a child - call it Process Q
if(fq==0){ // Child successfully forked
  a=a+b print values of a, b, and process_id
  fr=fork() // fork another child - call it Process R
  if(fr!=0){
    b=b+20
    //print values of a, b, and process_id
   }else{
    a=(a*b)+30
    //print values of a, b, and process_id
  }
}else{
  b=a+b-5;
```

```
print values of a, b, and process_id
```

Submission Instructions:

Part 1: Save your program with given name.

Part 2: Type and run the code to get the values of a, b, and process id. Briefly write your interpretation (in a text file) on working of the code to explain how your code arrives at the values you printed out. Note, run your program for Part 2 multiple times and see whether there are any changes in the order of execution before writing your report.

Save your responses to parts 1 and 2 in a single folder. Make sure your program compiles and run without any errors. Follow the same procedure as of the previous lab for the submission.

Next Class: Having several processes run the same program is only occasionally useful. But the child can execute another program using one of the exec functions/ system calls. We'll see various **exec** system calls next week. Also, there may be instances where a parent dies before its child. We shall see how such cases are handled.

******