CSC 33200 (L) - Operating Systems – Fall 2023

Lab 3: Process Management System Calls
Date: 02/24/2023

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 which 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 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. Subsequentlychanging 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 process child ends, the operating system changes the int variable status to represent how child passed away (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 avalue of -1

This handout describes the **exec** family of functions, for executing a command. You can use these functionsto make a child process execute a new program after it has been forked. The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in theheader file **'unistd.h'.**

execv (char *filename, char *const argv[])

The execv function executes the file named by filename as a new process image.

The argv argument is an array of null-terminated strings that is used to provide a value for the argv argument to the main function of the program to be executed. The last element of this array *must* be a null pointer.

execvp (char *filename, char *const argv[])

The execvp function is similar to execv, except that it searches the directories listed in the PATH environment variable to find the full file name of a file from filename if filename does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. **Shells use** execvp **to run the commands** that users type.

execl (char *filename, const char *argo, ...)

This is similar to execv, but the argv strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

execlp (char *filename, const char *argo, ...)

This function is like execl, except that it performs the same file name searching as the execvp function.

Lab 3. Marks: 30 **Submission Deadline: 03/09/2023**

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.

Marks: 5

2. Consider a series, S1 = 2 +4 +6+,..+10 (sum of even positive numbers upto 10) and another series S2=1+3+5+..+9 (sum of odd positive numbers upto 10) and
Another series S3 = 1+2+3+...+10. (sum of all positive number upto 10)

We know that S1+S2 = S3.

Now write a program, where a parent process creates 2 child process and computes S1 and S2. And Parent process computes S3. The input argument for program will be the end of series number for S3.

For example, if the execution file name is series.exe then, the argument will be
./series.exe 10

Child 1 will compute the series from 1 to upto 10 with difference 2. So, it would be 1+3+5+7+9
Child 2 will compute the series from 2 to upto 10 with difference 2. So, it would be 2+4+6+8+10

Parent will compute the series from 1 to 10 with difference 1. So, it would be,
1+2+3+4+5+6+7+8+9+10

Make sure your program can compute S1, S2 and S3 for any given number.

Marks: 10

3. Write a program with 2 children process, where one child is created to execute command that tells you the date and time inUnix.

Use execl(...).

Note, you need to specify the full path of the file name that gives you date and time information.

Announce the successful forking of child process by displaying its PID.

And another child is created to execute a command that shows all files (including hid-den files) in a directory with information such as permissions, owner, size, and when last modified.

Use execvp(...).

For the command to list directory contents with various options, refer the handout on Unix filesystem sent to you in the first class.

Announce the successful forking of child process by displaying its PID.

Marks: 5

4.

[Step 1] Prcs_P1.c: Create two files namely, destination1.txt and destination2.txt with read, write permissions.

[Step 2] Prcs_P2.c: Copy the contents of source.txt into destination1.txt and destination2.txt as per the following procedure.

1. Read the first 100 characters from source.txt, and among characters read, replace each char-acter '1' with character 'L' and all characters are then written in destination1.txt
2. Then the next 50 characters are read from source.txt, and among characters read, replace each character '3' with character 'E' and all characters are then written in destination2.txt.
3. The previous steps are repeated until the end of file source.txt. The last read may not have100 or 50 characters.

Once you're done with successful creation of executables for the above two steps do the following. Write a C program and call it Parent_Prcs.c. Execute the files as per the following procedure using execv system call.

[Step 3] Using fork create a child process, say Child 1 that executes Prcs_P1. This will create two destination files according to Step 1.

[Step 4] After Child 1 finishes its execution, use fork to create another child process, say Child 2 and execute
Prcs_P2 that accomplishes the procedure described in Step 2.

Note that: source.txt is already provided in the previous lab. Use that for task 4.

Marks: 10

Submission Instructions

- You should use *only* file management system calls for file manipulation
- Use the given source.txt
- All the programs MUST be clearly indented and internally documented
- Make sure your programs compile and run without any errors
- Only include c files or txt files for submission. Do not include any executables.
- Save all your programs with meaningful names and zip into a single folder as: Lab3_[your last name here].zip (e.g., Lab3_Xyz.zip)
- Email your code with the subject line, "Lab3-CSC33200–Class#G-*lastname*" (e.g., Lab3 - CSC33200-Class#G-Xyz)
- Email: sdebnath@ccny.cuny.edu

******