**CMSC 345 Operating Systems**

**Homework: Processes in Unix/Linux**

**Assigned: Thursday, January 30, 2014**

**Completion: Wednesday, February 5**

*Goals:*

* *To develop experience with process creation and process control.*
* *To use the C programming language for system development work, reviewing system calls: ps, kill, and fork.*
* *To identify process state information provided by system calls.*
* *To gain experience with dynamic experimentation.*

**Turn in your report on Canvas before class time. 35 points.**

Preparing Your Report:

Summarize what you have learned through your explorations suggested for each exercise, using a technical writing style (full sentences, paragraph structure) supported by examples.

For each exercise, you should experiment beyond the leading questions given. Summarize your explorations within each exercise (did you have a thesis, how did you approach experimentation, what was your discovery, can it be replicated?). Capture illustrative runtime activity with a screenshot or appropriate typescript file and include that in your writeup for each exercise.

Do NOT execute programs on beast or beauty for any project using “fork” commands. VPN to a campus machine and follow up with an SSH session to one of the lab machines *selma, poochie, wiggum, simpsons, homer, sideshowmel, drnick, snake, krusty, nelson, scratchy, itchy, cletus, bart, flanders, slh, gabbo, duff, moe, smithers, hibbert, mrburns, milhouse, apu, lenny*

There are a large number of C tutorials online and introductions to Unix/Linux commands.

Please review how to use “ps” and how to kill processes based on their pid values before you do any fork programming! You should read the **man pages for the system calls to ps and kill**.

1. **Pid and Kill**

As you know, every process has a process id. You can inspect the pid values for the processes you are running at the terminal window command prompt by issuing a system call **ps u**. How many processes do you have? Open several more command windows. How many processes are there now? Note the change in the results of the

**ps u**.

I have at least 100 active processes. Opening more adds processes to the list.

Create the following program lab0.c using a text editor (such as vi, vim, nano…)

#include<stdio.h>

main()

{

int i;

for (i=0; i <=1000000; i++)

printf("%d\n",i);

}

Compile the program cc lab0.c –o lab0.out *(or use gcc)*

This will create an executable file with the default name lab0.out

Execute the program with a call to

**./lab0.out** (this tells the system to look in the current directory for a file called lab0.out. The . before the / is a shorthand notation for current directory. Similarly, .. is shorthand for the parent directory (up one level in the directory hierarchy). The shell program looks in the PATH environment variable for where to locate files. It can’t find lab0.out without you indicating where it is located. See the FAQ below:

Assuming you are using the Bash (Bourne again shell) (substitute your favorite shell):

Why can’t I simply execute by calling lab0.out ? Bash tries to execute commands based on your PATH. It's an environment variable that tells Bash where to find programs. Usually your PATH points to /bin/ and /usr/bin/ and perhaps a few other directories. Bash can't find a.out in those directories, so it throws an error.  
  
By prefacing a command with **./** you're telling Bash to include the current directory you're working in when it looks for the executable named "a.out". ( Source: <http://www.linuxforums.org/forum/linux-programming-scripting/117351-solved-bash-out-command-not-found.html>)

What shell program are you executing? Bin/bash

Issue the command **echo $SHELL** to find out.

If you would like, you may change your shell program. See <http://www.linfo.org/change_shell.html>

(Over the years, I’ve changed my shell, so I’m not sure what the “standard” assigned shell is on the lab machines.)

Start executing the lab0.out code in one window, and look at the process list in another window. What is the Pid of the lab0.out process? While it is executing, terminate the process from the other window by issuing **kill -9 ##...# (where ##...# is the pid of lab0.out)** Feel free to experiment with the max value for the counter variable i. What’s the largest int in this architecture? Try this: modify the program to

I ran past 12,000,000 with no sign of stopping, so at least 12,000,000, I would guess if I left it running, it would be the equivalent of a long long, which I believe is 64 bits on this platform.

#include<stdio.h>

#include<limits.h>

main()

{

int i;

for (i=0; i <=INT\_MAX; i++)

printf("%d\n",i);

}

**What is the value of INT\_MAX? Why?**

**The value of INT\_MAX is the largest supported integer in a given archacture. Because the operating system is 64-bits.**

**How to terminate the program: First, I usually try a control-c**.

Then, I will look for the pid in another window and issue the system call to kill on the command line. Try it without any flags.

Some people do not advocate using the -9 flag. Read more about the signals that get sent at <http://superuser.com/questions/49024/kill-9-programs-but-they-still-hang-on>. So, I take drastic approach and risk a memory leak, according to this article. I use the -9 flag when I’m forking processes and suspect I have a runaway approach. But, experience tells me though I always try it, that approach often is ignored by a runaway process. There are less drastic approaches to try as detailed in <http://superuser.com/questions/107543/bash-man-page-kill-pid-vs-kill-9-pid>. If I’m worried about a reboot scenario, it’s easy to remember the -9 flag.

Next, open a third command window. In two of the windows, **execute** the lab0.out code. **Can you detect any scheduling switching?** (Do you see any delays in the I/O in either or both windows? If so, why do you see the delay?) In the third window, look at the process list. **How can you tell which lab0.out belongs to which window?** **Kill** one of the processes (from the third window where no code is executing). What happens?

I see no delay. I ran them both out of the same window using & and writing to file so I was given the pid for each.

1. **Read about the system command fork. Create and execute the following C program; then explain the observed execution.**

milhouse[99]% more hello.c

#include<stdio.h>

main ( )

{

printf("hello \n");

fork();

printf(" I am a process \n");

}

milhouse[100]% cc hello.c

milhouse[101]% a.out

hello //why do you see only one hello?

I am a process //why do you see two statements?

I am a process

milhouse[102]%

because hello is not being forked, printf is being forked. Because printf is forked once, therefore there are two streams doing the output.

1. **Execute the following C program and explain your observed execution. Will execution order always be the same? Explain.**

milhouse[109]% more lookpid.c

#include <stdio.h>

#include <stdlib.h>

main ( )

{

int pid;

printf("hello \n");

pid = fork ( );

if (pid == -1)

{

perror("bad fork");

exit(1);

}

if (pid == 0)

{

printf("I am the child process \n");

}

else

{

printf(" I am the parent process \n");

}

}

milhouse[110]% cc -o mylook.out lookpid.c *{notice the order of arguments can vary}*

milhouse[111]% mylook.out

hello

I am the child process

I am the parent process

milhouse[112]% ^C

milhouse[112]%

yes, it will remain the same because the child process is always pid 0.

1. **Execute the following C program and explain your observed execution. Read about the call to wait and the use of status. (Possible source:** [**http://man7.org/linux/man-pages/man2/wait.2.html**](http://man7.org/linux/man-pages/man2/wait.2.html) **)**

**Describe what you have learned in your discussion.**

milhouse[118]% more waiting.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

main ()

{

int status;

int pid;

pid = fork();

if (pid == 0)

{printf("hello ");

}

else

{wait(&status);

printf(" world\n");

}

}

I learned that the wait is used to hold a child process so that a parent process or an otherwise critical process can be executed and used.

**4. Explain the following C program and execution. Be certain to experiment with the sleep parameters. Do the parameters to sleep serve to synchronize or merely relinquish the CPU? (In general, it is not sufficient to make assumptions about processor speed for synchronization.)**

milhouse[15]% more child.c

#include <stdio.h>

#include <stdlib.h>

int main (void){

/\* the child process's new program

this program replaces the parent's program \*/

printf("process[%d]: child in execution\n", getpid());

sleep(1);

printf("process[%d]: child terminating...\n", getpid());

exit(25);

}

milhouse[16]% ^C

milhouse[17]% more parent.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void)

{

int pidvalue;

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

else /\* parent code here \*/

{ printf("process[%d]: parent in execution...\n", getpid());

sleep(4);

if (wait(NULL)> 0) /\* child terminating \*/

printf("process id [%d]: parent process detects terminating child\n",

getpid());

printf("process[%d]:parent terminating....\n",getpid());

}

}

the services are synched with the CPU.

**5.** **Explain the following C program after observing the execution.**

milhouse[113]% more syscall.c

main()

{

system("date");

system("ls");

}

milhouse[114]% ^C

milhouse[114]%

**How is calling system different from using an exec command in the example in Figure 3.9 of Silberschatz?**

**It is indirect, the system function takes an argument and translates it to the appropriate shell script for a given system.**

**6. Write a C program with 2 child processes to print out their pids. Do they always execute in the same order? Try it with 10 child processes, with all children answering to the same parent. What do you find? Are you certain? Can we rely on assumed timings? Your code must be well documented and included in the body of your report (not in an appendix).**

**Everything appears to execute in the same order, however, when running 10 child processes, the parent executed before the all of the children, despite this, the rest of the children still executed and terminated. I assume this to be true, as I have run it several times, always with the same result. No we cannot rely on assumed timings, as the parents executed before the children.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void)

{

int pidvalue;

//bank of 10 forked children

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process1\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process2\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process3\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process4\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process5\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process6\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process7\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process8\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process9\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

pidvalue = fork();

printf("pidvalue = %d\n",pidvalue);

if (pidvalue == 0) /\* this is the child process \*/

{

printf("entered child process10\n");

execlp("./child.out", "./child.out", NULL);

printf("oh no, got here\n");

exit(0); /\* should never get here, terminate\*/

}

else /\* parent code here \*/

{ printf("process[%d]: parent in execution...\n", getpid());

sleep(4);

if (wait(NULL)> 0) /\* child terminating \*/

printf("process id [%d]: parent process detects terminating child\n",

getpid());

printf("process[%d]:parent terminating....\n",getpid());

}

}

Turn in a report of your work on these exercises via Canvas. Please label the document file with your *name* in the title of the file. Make certain you have your name on the first page of the document as well.