

CSC 262

Project 4: Processes

1 Overview

In this lab you will be messing around with process creation. You'll learn how to spawn processes and execute completely different programs. However, first **a word of caution**. As you begin to develop your program, while you are still debugging you may accidentally make a program that creates processes uncontrollably over and over, eventually causing your machine to crash (this little gem is often called a fork bomb or fork bunny). If this happens, you will probably need to reboot and fix your program before running it again. Because of this possibility, you should not work on this project using any of the shared CS machines (i.e., beowulf or grendel). Use the lab computers instead. Also, a buggy program may create extra processes that persist after the parent process has finished. Use the `ps` command to detect this situation, and `kill` to deal with it and clean up.

2 Linux Processes

A process, also known as a task under Linux, is a running instance of a program. This means that if 10 users on a server are all using the same program, such as emacs, there are 10 emacs processes running on the server, although they all share the same executable code (also called program text).

The processes on a UNIX system can be viewed using the `ps` command, like so:

```
1 ?          00:00:01 init
2 ?          00:00:00 kthreadd
3 ?          00:00:00 migration/0
4 ?          00:00:01 ksoftirqd/0
5 ?          00:00:00 watchdog/0
6 ?          00:00:00 events/0
7 ?          00:00:00 khelper
59 ?         00:00:02 kblockd/0
63 ?         00:00:00 cqueue
65 ?         00:00:00 ksuspend_usbd
70 ?         00:00:00 khubd
73 ?         00:00:00 kseriod
116 ?        00:00:01 pdflush
117 ?        00:00:00 pdflush
118 ?        00:00:02 kswapd0
163 ?        00:00:00 aio/0
301 ?        00:00:00 kpsmoused
...
1901 pts/0    00:00:00 bash
8586 pts/1    00:00:01 bash
```

```

8698 pts/2    00:00:00 bash
9188 ?          00:00:00 notification-da
9745 ?          00:00:00 gvfsd-computer
10814 pts/0     00:00:00 ps
29384 ?          00:00:00 gconfd-2

```

On linux, you need to type `ps -e` to see processes not attached to your current terminal. The PID column list the process id number for a process, while TTY shows you what terminal the process is attached to (? means that it has no terminal), and last CMD is the name of the program actually running.

The process ID (PID) is a unique identifier for a process. The operating system uses a 32-bit counter `last_pid` to keep track of the last PID assigned to a process. When a process is created, the counter is increased and its value becomes the PID of the new process. Because the counter may wrap around at some point, the kernel needs to check if the value of `last_pid++` already belongs to a task, before it can assign it to a new process.

More information about the process list can be displayed using the `-l` option of the `ps` command.

```

F S  UID    PID  PPID  C  PRI  NI  ADDR  SZ  WCHAN  TTY          TIME CMD
0 R   500   1901  1898  0   80   0   -    1233  -      pts/0       00:00:00 bash
0 R   500  10820  1901  0   80   0   -    1136  -      pts/0       00:00:00 ps

```

The most important thing here is the S column, which indicates the state of a process. Processes can be in the following states:

code	name	C name
D	uninterruptible sleep	TASK_UNINTERRUPTIBLE
R	runnable (on run queue)	TASK_RUNNING
S	sleeping	TASK_STOPPED
T	traced or stopped	TASK_STOPPED
Z	defunct (zombie)	TASK_ZOMBIE

If you run `ps -el`, you'll notice that most processes are sleeping. That is, they're waiting for some kind of input. `ps -l` also shows the user ID of the owner (UID), the PID, and the PID of the parent process (PPID). The parent process is the process that launched this process. In the example above, you can see that the `ps` commands PPID is the PID of `bash`, which makes sense because I launched `ps` from my shell prompt.

Because of the child-parent relationship among processes, you can also view processes as a tree. And the `pstree` command will show you all processes in the system organized as a tree. The first thing you may notice about `pstree`'s output, is that the tree is rooted at `init`. That's because `init` is the initial process in the system that gets everything going (hence the name). Processes whose PPID is `init` (i.e. 1) and that do not have a controlling terminal (and are not zombies) are called *daemons*. A daemon is just UNIX speak for a background server process. For instance, the login window, a web server, and the `ssh` server are all daemons. Usually, the executable for a daemon is suffixed with the letter 'd' (e.g. `sshd`) to let the system administrator know that it is the server program.

Q1: Give examples of at least 3 daemons running on your computer.

Q2: Draw the relationship tree for the following process listing. Annotate each node of the tree with the executable name and PID (remember that init is always there and has a PID of 1).

UID	PID	PPID	CPU	PRI	NI	WCHAN	STAT	TT	TIME	COMMAND
88	86	1	0	63	0	-	Ss	??	4:59.77	WindowServer
501	281	86	0	46	0	-	S	??	0:00.11	iTunes
501	282	86	0	47	0	-	S	??	0:08.06	Terminal
0	287	282	0	31	0	-	Ss	p1	0:00.02	login
501	461	293	0	57	0	-	S	??	0:46.52	firefox-bin
501	531	86	0	97	0	-	S	??	37:12.33	Safari
501	726	86	0	62	0	-	S	??	3:12.02	Mail
501	751	293	0	46	0	-	S	??	0:32.22	Aquamacs Emacs
501	293	287	0	31	0	-	S	p1	0:00.27	-bash

3 Signals

In UNIX, signals are an asynchronous communication channel to a process. Software signals perform for software what interrupts do for hardware. A signal interrupts the normal execution of a process in order to deal with some kind of event. Standard signal examples are a segmentation fault (the kernel sends the program a SIGSEGV), the user typing CTRL-C at the terminal (sends SIGINT to the process). A complete list of signals can be seen by consulting the signal man page. From the command line, you can use the `kill` binary to send signals to a program. For example,

```
csdhcp042:~ turk$ sleep 200 &
[1] 934
csdhcp042:~ turk$ ps
  PID  TT  STAT      TIME COMMAND
  293  p1  S+      0:00.29 -bash
  878  p2  S       0:00.17 -bash
  934  p2  S       0:00.01 sleep 200
csdhcp042:~ turk$ kill -s SIGKILL 934
csdhcp042:~ turk$
[1]+  Killed                  sleep 200
csdhcp042:~ turk$
```

This example sends the `KILL` signal to the `sleep 200` (which just sleeps for 200 seconds and then exits) process (the trailing ampersand, by the way, puts the process in the background). `kill` is very useful for terminating unwanted processes (they've either ground to a halt or are going crazy). The two signals for doing that are `SIGKILL` and `SIGTERM`. `SIGTERM` kills the process "nicely", the process has a chance to handle the signal and clean up a bit, whereas `SIGKILL` just immediately kills it dead (cleanup is handled by the OS).

There are three ways of handling a signal. It can be ignored (not all signals can be ignored), it can be sent to a default handler, or it can be sent to a custom handler. In order to install a

custom signal handler in your code you need to use the `signal` function (which is included in `signal.h`). `signal` takes an integer (which is the signal number) and an address of a function to handle that signal. The signal handler function itself needs to return void and accept an int as its one and only arg. For example, the following code registers a handler for SIGINT (CTRL-C):

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

int loop_forever = 1;

void on_sigint(int signo){
    printf("CTRL-C pressed\n");

    loop_forever = 0;
}

int main(void){
    loop_forever = 1;
    signal(SIGINT, on_sigint);
    while(loop_forever){ }
}
```

Q3: Modify the above code to also catch the SIGTERM and SIGUSR1 signals as well. The behavior should be the same except that the output should reflect the received signal (e.g. if you send the program SIGTERM, it should output something like “SIGTERM received”)

Q4: `signal` is actually the old-style way of handling signals. New code should all be using `sigaction`. `sigaction` has a more complicated signature, but the big difference is that a handler installed by calling `signal` is uninstalled after the signal is received (and replaced with the default handler). That is, your SIGINT handler has to be manually RE-INSTALLED after a SIGINT is received. A handler installed with `sigaction` is not replaced after a signal is received. In the old days, if you wanted a signal handler to always be valid, you’d have to insert a call to `signal` in your signal handler to re-install itself. The following code reinstalls a handler for SIGINT (note that this code runs forever):

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

void on_sigint(int signo){
    signal(SIGINT, on_sigint);
    printf("CTRL-C pressed\n");
}

int main(void){
```

```

    signal(SIGINT, on_sigint);
    while(1) { }
}

```

With the call to `signal` as the first line in the handler, is it still possible to miss a `SIGINT` (i.e. it gets sent to the default handler rather than `on_sigint`)? If so, how? If not, why not?

Q5: `SIGKILL` cannot be overridden with a user-installed handler. Consult the signal man page to discover which other signals cannot be caught or ignored. Why do you think this is?

4 Process Creation

In UNIX, process are created by first duplicating the parent process. This is called forking, after the `fork` system call. In C, you invoke the `fork` system call with the `fork()` function:

```

#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

```

And yes, you do need both header files. After forking the parent and child process share the same code, but effectively have different stacks and heaps (the child has its own separate address space). The `fork` function returns a PID, this is used to identify (in the code) which process you are. In the child process, `fork` will return 0, but in the parent `fork` will return the PID of the newly created child (if `fork` returns a negative value, then an error occurred). Consider the following code:

```

#include<stdlib.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main() {
    pid_t pid;

    pid = fork(); /* the process is split here !! */

    /*
     * Now we have two processes (child and parent) running on the
     * same executable code. The way we tell who we are is by
     * looking at the return value of fork(): 0 for the child
     * process and > 0 for the parent.
     */
}

```

```

if (pid == 0) {
    /* CHILD PROCESS */
    printf("Child process!\n");
} else {
    /* PARENT PROCESS */
    printf("Parent process!\n");
}
exit(0);
}

```

The `wait` and `waitpid` functions can be used to make a parent process wait until a child process terminates. For example, the following code makes the parent wait until the child is done before printing:

```

#include<stdlib.h>
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main() {
    pid_t pid;

    pid = fork(); /* the process is split here !! */

    if (pid == 0) {
        /* CHILD PROCESS */
        printf("Child process!\n");
    } else {
        /* PARENT PROCESS */
        wait(NULL); /* wait until the child terminates */
        printf("Parent process!\n");
    }
    exit(0);
}

```

Q6: Consider the following code:

```

#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>

int main() {
    pid_t pid;

```

```

int num_coconuts = 17;

pid = fork(); /* the process is split here !! */

if(pid == 0) {
    /* CHILD PROCESS */
    num_coconuts = 42;
    exit(0);
} else {
    /* PARENT PROCESS */
    wait(NULL); /*wait until the child terminates */
}

printf("I see %d coconuts!\n", num_coconuts);
exit(0);
}

```

What will this program print out? Why?

5 Exec

`fork` lets you make a copy of the current program, but what if you wanted to run a completely different program? In UNIX, the `exec` system call causes a process to invoke another executable. There are many functions that act as front ends to the `exec` system call (see the `exec` man page). These functions replace the image of the current process' with the executable image specified as an argument to `exec`. For example, the following code tries to launch `emacs`:

```

#include<stdlib.h>
#include<unistd.h>

int main(){
    execl("/usr/bin/emacs", "emacs", NULL);

    /* We can only reach this code if execl returned with
    * an error
    */
    perror("execl");
    exit(1);
}

```

`execl` accepts the executable name as well as the command line arguments. It is a variable argument function so it may accept multiple strings. For example if you wanted to execute the equivalent of “`emacs foo.c`”, then the `execl` call would look like: `execl(“/usr/bin/emacs”, “emacs”, “foo.c”, NULL);`.

With `fork` and `exec` you now have all the tools to launch programs on UNIX. First, you need to fork off a new process from the current one. Then, in the child process you invoke `exec` to launch the new program.

- Q7:** Windows creates processes differently. On windows the `CreateProcess` function is used to create a new process. The first argument to `CreateProcess` is the path to the executable. Therefore, to achieve the same effect as `fork`, `CreateProcess` would be called with the same executable (`argv[0]`, usually). What are the relative advantages and disadvantages of UNIX style `fork` and `exec` compared to Windows-style monolithic `CreateProcess`? Think especially in terms of programmer convenience and efficiency of process creation.
- Q8:** You're familiar with so-called shell programs; these are the programs that run in a terminal window and accept commands typed by the user. Implement a small shell that repeatedly asks for the a program name and then executes that program using `fork()` and `execlp()`. The shell must terminate when the user types `exit`. The shell shouldn't prompt for a new program to execute until the previous one has terminated. You can use the `wait()` function to make the parent process wait for the child to terminate. (Recall that you can learn more about this function by typing `man 2 wait` at a terminal prompt. Also note that `execlp()` differs slightly from `execl()` as shown in the example program above; read the corresponding `man` page to learn more about the difference.)

Your program should also respond to signals as follows: If `SIGINT` (`CTRL-C`) is typed while a child process is executing, your program should pass along the signal to that process (and keep itself running). If no child process is currently running, your program should catch the signal and print an informative message explaining how to terminate the program (by typing `exit` at the prompt). On the other hand, if your program receives a `SIGTERM` signal, it should first send the `SIGTERM` signal to any child processes and then terminate itself as well. (Remember for testing that you can use the `kill` command from another terminal window to send signals to your shell program.)