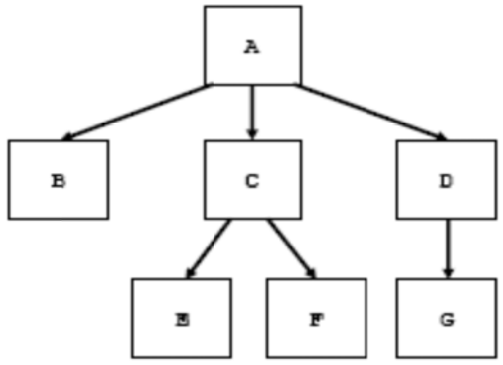
Lab 4: Processes and Signals (Week of 9.2.15)

Exercise 1

Consider the family tree of processes shown below (where A is the initial parent process):



- This particular family tree can be generated using exactly **SIX** `fork()`calls.
- This family tree can *also* be generated with exactly **FIVE** `fork()`calls.
- This family tree can *also* be generated with exactly **FOUR** `fork()`calls.
- This family tree can *also* be generated with exactly **THREE** `fork()`calls.

Your task is to code the four scenarios. You should explain which processes should fork a child and in what order, to create the tree in the specified number of steps. Use tables similar to that provided below to organize your thoughts. If using the table, you should put in each cell, the PID (corresponding letter) of the process created by the particular parent at that fork call. If a particular process does not create a child, place an X in the appropriate cell. For example, let's assume that only *Process A* creates *Process D* during the first fork, and then only *Process D* creates *Process G* during the second fork. The first two columns of the table should look like the example shown below.

Process	Fork 1	Fork 2	Fork 3	Fork 4	Fork 5	Fork 6
A	D	X				
B	X	X				
C	X	X				
D	X	G				
E	X	X				
F	X	X				
G	X	X				

Caveats: Calling `fork()`inside of a loop counts as multiple fork calls, even though it only appears once in the code. Solutions using forks within loops may be correct, but you need to be very careful in your approach.

Exercise 2

You can see the currently running processes in the system with the **ps** command.

- Study the **ps** command by executing the man ps command.
- Try the **ps** command in your system. Find the total number of processes running on your system.
- You can terminate a process using the **kill** command. Study this command by executing the man kill command. Experiment with this command by starting a process and terminating it with **kill**. (You can start a program as a background process by "*program* &".)
  You may use the following program for experiment:

  ```
  int main()
  {
     while ( 1 );
     return 0;
  }
  ```

  To **kill** a process, first use "ps -l" to find out the process id (*pid*) of the process. Then issue the command "kill *pid*". Use "ps -l" to check again to see if the process has been terminated.

Exercise 3

Write a program that executes the "cat -b -v -t filename" command. Call your executable myfork. The call to your program will be made with the following command:

% myfork filename

- Your code will fork()
- The child will use the execl to call cat and use the filename passed as an argument on the command line
- The parent will wait for the child to finish
- Your program will also print from the child process:
  - The process id
  - The parent id
  - The process group id

  and print from the parent process:

  - the process id
  - the parent id

- the process group id

Comment out the execl call and add instead a call to execv. Add any necessary variables to do that.

Exercise 4

A simple shell can be written like this:

```
while (1) {
     printf("%% ");
     fgets(command, 80, stdin);
     system(command);
}
```

Write a program that does exactly this, but with your own implementation of the system() function. Use the execv() function to start an external program. (The execv() function is just a frontend for the execve() function so you should read both manpages to understand what this function does.)

To do an ls -l in your shell, you actually need to call /bin/sh -c "ls -l", so -c and ls -l are parameters to /bin/sh. Please note (as you can read from the manpage), that the list of arguments **must** be terminated by a NULL pointer, and this pointer must be cast (char *) NULL.

With your program you should be able to create a session like this:

(Note that the % is the prompt of your program).

```
$ ./myshell

% ls

Makefile myshell myshell.c

% abc

/bin/sh: abc: command not found

% ˆD (Ctrl-D)

$
```

Exercise 5

Signals are software interrupts. We can handle signals using the signal library function:

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

```
void (*signal(int sig, void (*func)(int)))(int);
```

This rather complex prototype means that **signal()** is a function that takes two parameters: *sig* and *func*. The function to be caught or ignored is specified by argument *sig* and *func* specifies the function that will receive the signal; this function must be one that takes a single **int** argument and is of type **void**. The **signal()** function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of the two special values

SIG_IGN	Ignore the signal.
SIG_DFL	Restore default bahavior.

Learn more about **signal** using the "man" command. Now try the following program.

```
void func( int sig )
{
  printf("Oops! -- I got a signal\n");
}

int main()
{
  signal(SIGINT, func);        //catch terminal interrupts

  for ( int i = 0; i < 20; ++i ) {
    printf("IT 215 lab on signals\n");
    sleep ( 1 );
  }
  return 0;
}
```

Run the program and hit ^C for a few times. What do you see? Why?

Exercise 6

Consider the following C program.

```
int val = 10;
void handler(sig) {
  val += 5;
  return;
}
```

```
int main() {
   int pid;
   signal(SIGCHLD, handler);
   if ((pid = fork()) == 0) {
      val -= 3;
      exit(0);
   }
   waitpid(pid, NULL, 0);
   printf("val = %d\n", val);
   exit(0);
}
```

Run this program. What do you see? Why?

Exercise 7

Consider the following program.

```
pid_t pid;

int counter = 0;

void handler1(int sig)

{

   counter ++;

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

   fflush(stdout); /* Flushes the printed string to stdout */

   kill(pid, SIGUSR1);

}
```

```
void handler2(int sig)

{

    counter += 3;

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

    exit(0);

}

main() {

    signal(SIGUSR1, handler1);

    if ((pid = fork()) == 0) {

        signal(SIGUSR1, handler2);

        kill(getppid(), SIGUSR1);

        while(1) {};

    }

    else {

        pid_t p; int status;

        if ((p = wait(&status)) > 0) {

            counter += 4;

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

        }
    }

}
```

Run this program. What do you see? Why?