Due: February 15

Purpose:
In this lab you will become familiar with UNIX processes. In particular you will examine processes with the ps command and terminate processes with the kill command. You will also see how you can run a program in the background. Finally you will write a C program to spawn a process.

Instructions:

1. Remote login to kant and place copies of the files sequencer.c and generator.c into your working directory. (You can find the code for both programs at the end of the lab. I can show you a way to transfer files between Windows and kant using 'psftp' in lab today.) Compile and run the program sequencer.c.

   First modify the program sequencer.c to introduce a 1 second delay in the loop with the statement
   ```
   sleep(1);
   ```
   Now introduce an infinite loop by replacing the for statement with a while (1). You will need to pull the initialization and update of the loop control variable into separate statements. What happens when you run the program? (Note: Ctrl-C will halt a runaway program.)

2. Open a second terminal window (how you do this will depend on the windowing environment you are running). In one terminal window start the program and in the second terminal window enter the command

   ```
   ps -ef
   ```

   This gives a complete list of the processes. Here are some additional uses of ps:

   ```
   ps -ef | grep <username>
   ps U <username>
   ps -U <username>
   ps -f -p <pid>
   ```

   Find the line for the process that is running your program. How are you able to identify the process? What is the pid and ppid of this process? What do pid and ppid represent?

   You can terminate a process with the kill command:

   ```
   kill <pid>
   ```

   Terminate your program by terminating the process. Verify that it is gone by examining the list
of processes.

Can you terminate the first terminal window (by terminating the shell)? Do so if you can and describe how you did it.

3. Repeat the situation you had in Step 2, that is, open two terminal windows and run the program from the first one.

Review the process list in the second terminal window. Record the pid’s for the first terminal window and the program. Record the ppid of the program. What connection do you see among these id’s?

Now terminate the first terminal window. (You may need to give the command
\[ \text{kill} -9 <\text{pid} > \]
to really kill it.) What do you notice about the process list after terminating the first window?

4. Now run the program in the background by entering the command
\[ .;/a.out & \]

Review the process list. What is the pid of the program? What process is the parent of the program? Can you terminate the program without going to a second terminal?

Run three copies of the program from the first terminal window in the background. Record the pid’s of each of the three processes. What do you observe about their ppid’s? What process spawned the three processes? What happens when you terminate the spawning process?

Terminate the three processes.

5. Compile and run the program \textit{generator.c}. Note that the program requires a filename as a command line argument. Describe the output.

Modify the code to generate the upper case letters ‘A’ – ‘J’. Compile and run the program to test your modification.

6. Now we want to write a program that spawns another process. In C one spawns another process with the \textit{fork()} system call. The new process executes the code immediately following the \textit{fork()} call. The effect is there are two processes executing the same code (and how cool is that!).

In order to have the new process execute different code you need to branch on the \textit{pid} returned by the \textit{fork()} call. If the value returned is 0 then it is the new (child) process. If the value returned is not 0 then it is the original (parent) process and the value returned is the \textit{pid} of the newly spawned process.
The code typically has the form:

```c
int pid;
if ((pid = fork()) == 0)
{
    // child code
    ...
}
else
{
    // parent code
    ...
}
```

Modify the program `generator.c` so that a child process is spawned. Further have the child generate the lower case letters and have the parent generate upper case letters.

When you modify the program be sure the buffer declaration and the opening of the file precede the code for the `fork()`. The child code should call `exit(0);` at the end of its block. Also, for this to work properly the parent will need to wait for the child to terminate before closing the file and terminating. The following statement will do this `wait(&status);` where `status` is an `int` variable.

Run the program and describe the results.

7. Modify the program `generator.c` to print out the value of `pid` both in the child code and the parent code (appropriately labeled). Record your results.

How can we know the non-zero value printed is the `pid` of the child process? Add additional statements to print out the `pid` and `ppid` of each process by calling the system functions `getpid()` and `getppid()`.

Run the program and record your results.

8. As a last modification of the program introduce a one-second pause in each loop by using the system call `sleep(1)`.

Run the program with this modification. Does the program still have the same behavior?

Run the program again and while it is running go to a different terminal window and examine the process list to find the `pid` and `ppid` of the two processes. Record your observations.

9. For the last step of the lab reflect on what you have learned about the relationship of a process with a process it has spawned. In particular examine the code in `generator.c` with the modifications you have made. Address the following questions:
   - Does the child (spawned) process inherit the resources owned by the parent?
   - Do the two processes share the same address space (memory)?
- Can the parent terminate independent of the child?
Refer to your code and observations made in the lab and use them as evidence when addressing these questions.

Hand In:

Hand in a write-up that includes answers to questions posed in the steps above and results of executions along with observations you have made. Turn in the programs `sequencer.c` (Step 1), and `generator.c` (Step 8). E-mail the two source files to your instructor as well.
/* sequencer.c 
Programmer: ska 
Date: 8 Feb 2018 */

#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("%d\n", i);
    }
    return 0;
}

/* generator.c 
Programmer: ska 
Date: 8 Feb 2018 */

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

#define PERMS 0666

int main(int argc, char * argv[]) {
    int fd, i;
    char buf[2];
    buf[0] = 'a';
    buf[1] = '\n';

    if ((fd = creat(argv[1], PERMS)) == -1)
    {
        printf("generator: can't create %s, mode %o.\n", argv[1], PERMS);
        exit(1);
    }

    for (i = 0; i < 10; i++)
    {
        write(fd, buf, 2);
        buf[0]++;
    }

    close(fd);
    return 0;
}