CS2211a Lab No. 3
Introduction to UNIX
Tuesday September 30, 2014 (sections 3 and 2),
Wednesday October 1, 2014 (sections 6 & 7), and
Thursday October 2, 2014 (sections 4 and 5)

Location: ***MC10*** lab

The objective of this lab is:

- to get familiar with Unix Processes and Job Control, as well as
- to practice Unix Shell Environments

If you would like to leave, and at least 30 minutes have passed, raise your hand and wait for the TA.
Show the TA what you did. If, and only if, you did a reasonable effort during the lab, he/she will give you the lab mark.

==========

1. Use the `ps` Unix command to show the process ID of your current shell.
   Trace all parent process IDs of your current shell to identify the only living ancestor of it.

2. Use the top Unix command to display and update information about the top cpu processes ***sorted by***
   a. memory usage
   b. CPU usage
   c. time

3. Use the top Unix command to display the ***full command line*** and update information about your ***own*** top cpu processes. Identify the process ID of the top command that you are running right now. Using top command, kill this top process.

4. Use the top Unix command to display and update information about the ***root*** top cpu processes.
   Do you recognize any of these processes? Hmm, you should ☺ **!!**

5. Execute the command `sleep 100` (which will suspend execution for 100 seconds).
   How do you *terminate* the execution of this command?
   How do you *stop* the execution of this command for a while and then *resume* it again?

6. The following is a simple Unix Bourne shell script (program). This program displays the shell script filename (`$0`), the iteration number (`$i`) followed by a colon (`:`), and the date/time of displaying this information. The program will do so three times--one second a part (`sleep 1`). At the end, it will display a message saying that the program finished execution.

```
#!/bin/sh
i=1
while [ $i -le 3 ]; do
  echo $0 $i ":"   `date`
  sleep 1
  i=`expr $i + 1`
done
echo $0 ": finished execution."
```

   a. Type in (or download) the above script to a file called prog-A
   b. Copy prog-A to prog-B and copy prog-A to prog-C
   c. Change the permission of prog-A, prog-B, and prog-C files to give execution permission to the user owner.
   d. What will be the output if you execute the following commands (below)?
   e. Draw a time chart to show when, approximately, each program will start and end.
   f. How do you stop *each* of these programs separately during their execution? You may want to change "`sleep 1`" to "`sleep 20`" in all three programs to give you more time to stop each program.

```
   i. prog-A;  prog-B;  prog-C
  ii.  prog-A;  prog-B   prog-C
 iii.  prog-A;  prog-B;  prog-C&
  iv.  prog-A;  prog-B   prog-C&
   v.  prog-A&  prog-B;  prog-C
  vi.  prog-A&  prog-B   prog-C
 vii.  prog-A&  prog-B;  prog-C&
viii.  prog-A&  prog-B   prog-C&
  ix. (prog-A;  prog-B)& prog-C
   x. (prog-A   prog-B)& prog-C
  xi. (prog-A;  prog-B)| prog-C
 xii. (prog-A;  prog-B)| tee prog-C
xiii.  (prog-A   prog-B)| prog-C
 xiv.  (prog-A   prog-B)| tee prog-C
```

7. The following C program demonstrates the fork() function. In this program, the main process forks a single child process. After a successful fork(), two processes are now running in parallel, with the new child being a copy of the parent. The fork() call returns the child's pid (process ID) to the parent process, and returns 0 to the child process. This is how the parent gets to know the child's pid, and is also used to separate the program code of the parent and of the child.

   Before the child process exits, it asks for an 8-bit number from the user which it uses as its return code. The parent waits for the child to exit. It retrieves the child's exit code, and displays it. The parent then exits shortly thereafter.

   The C function and UNIX system calls are declared in the included header files at the start of the code.
   The following C functions are used in this program:

   fork() ➔ creating a copy of the calling process

   sprintf() ➔ placing output in consecutive bytes, followed by the null byte (\0)

   fwrite() ➔ writing, from the provided array pointed, up to n items elements whose size as specified, to the file stream

   strlen() ➔ returning the number of existing bytes in a null string

   getpid() ➔ returning the process ID number of the current process

   getppid() ➔ returning the parent process ID of the calling process

   sleep() ➔ suspending the caller from execution for the number of seconds specified by the argument

   scanf() ➔ reading bytes from the standard input, interpreting them according to a format, and storing the results in its arguments

   exit() ➔ terminating process with status

   wait() ➔ waiting for child process to stop or terminate

   a) Read and understand the code

   b) Download the program to you gaul account and name it fork_example.c

   c) Compile the program using the following Unix command:
      `gcc -Wall fork_example.c -o fork_example`

   d) Run the code by executing the generated “fork_example” executable code

   e) Re-run the code few times and compare the outputs.

   f) Explain why you got the outputs in various order when you run the code several times.

```
#include <unistd.h>   /* Symbolic Constants */
#include <stdio.h>    /* Input/Output */
#include <sys/wait.h>   /* Wait for Process Termination */
#include <stdlib.h>   /* General Utilities */
#include <string.h>   /* General Utilities */

int main()
{
  char buf[100];
  int  childpid; /* variable to store the child's pid */
  int  retval;   /* child process: user-provided return code */
  int  status;   /* parent process: child's exit status */

  /* only 1 int variable is needed because each process would have its own
     instance of the variable.  Here, 2 int variables are used for clarity */

  /* now create new process */
  childpid = fork();

  if (childpid >= 0) /* fork succeeded */
  {
    if (childpid == 0) /* fork() returns 0 to the child process */
    {
      sprintf(buf, "<> <> CHILD: I am the child process!\n");
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "<> <> CHILD: Here's my PID: %d\n", (int) getpid());
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "<> <> CHILD: My parent's PID is: %d\n", (int) getppid());
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "<> <> CHILD: The value of my copy of childpid is: %d\n", childpid);
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "<> <> CHILD: Sleeping for 1 second...\n");
      fwrite(buf, strlen(buf), 1, stdout);

      sleep(1); /* sleep for 1 second */

      sprintf(buf, "<> <> CHILD: Enter an exit value (0 to 255): ");
      fwrite(buf, strlen(buf), 1, stdout);

      scanf(" %d", &retval);

      sprintf(buf, "<> <> CHILD: Goodbye!\n");
      fwrite(buf, strlen(buf), 1, stdout);

      exit(retval); /* child exits with user-provided return code */
    }
    else /* fork() returns new pid to the parent process */
    {
      sprintf(buf, "  #   PARENT: I am the parent process!\n");
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "  #   PARENT: Here's my PID: %d\n", (int) getpid());
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "  #   PARENT: The value of my copy of childpid is %d\n", childpid);
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "  #   PARENT: I will now wait for my child to exit.\n");
      fwrite(buf, strlen(buf), 1, stdout);

      wait(&status); /* wait for child to exit, and store its status */

      sprintf(buf, "  #   PARENT: Child's exit code is: %d\n", status/256);
      fwrite(buf, strlen(buf), 1, stdout);

      sprintf(buf, "  #   PARENT: Goodbye!\n");
      fwrite(buf, strlen(buf), 1, stdout);

      exit(0);  /* parent exits */
    }
  }
  else /* fork returns -1 on failure */
  {
    fprintf(stderr, "Error: unsuccsseful fork\n"); /* display error message */
    exit(0);
  }
}
```

8. There are many shells available under Unix. These shells include `csh`, `tcsh`, `sh`, and `bash`. The default login shell for all students is `tcsh`. To change your current shell, you just type the shell name and hit enter. From this moment till you execute `exit`, your Unix environment will be treated according to this shell governing rules. To determine your current shell, use `echo $0` command.
   In this lab, we will attempt to *explore* these four shells.
   a. Which shell of them correctly accepts arrow keys at the command-line levels?
   b. Which shell of them can execute the following Unix commands?
      Is there any difference in the output of these Unix commands across various Unix shells?
      i. `set`
      ii. `unset`
      iii. `export`
      iv. `setenv`
      v. `unsetenv`
      vi. `printenv`
      vii. `alias`
      viii. `unalias`
      ix. `history`

9. Under your default shell (`tcsh`), create a regular variable called **`abc`**.

10. Under your default shell (`tcsh`), create an environment variable called **`DEF`**.

11. Under your default shell (`tcsh`), display *all current environment variables*.

12. Under your default shell (`tcsh`), display *all current regular (local) variables*.

13. Change your current shell to `sh`,
    a. Create a regular variable called **`ghi`**
    b. Create an environment variable called **`JKL`**
    c. Display *all current environment variables*
    d. Display *all current regular (local)* and *environment variables*,
    e. Exit `sh` shell.