Problem 1 (0 Points)
The Process item in the course Syllabus contains a link to the section of The Gnu C Library Reference Manual which discusses Processes. Read this section. Follow the Top link to the Main Menu and familiarize yourself with what is documented in the other sections.
Consider the following code fragment:

```c
#define max(\ldots)

mymap[\ldots]
```

Problem 2 (0 Points)
Consider the following code fragment:

```c
printf("myid = %d\n", getpid());
for (int i=0; i<2; i++) {
    pid_t p = Fork(); // never returns an error
    printf("i = %d (pid = %d), fork returned = %d\n",
            i, getpid(), p);
    ... code goes here ...
}
```

a) Complete the code fragment so that it will produce a chain of three (3) processes where the original process is the parent of a child which is the parent of the another child.

b) If we assume that process IDs start at 1000 and assigned consecutively for each new fork call, what will be the output of the code in Part a? Explain.

c) Will the output be different if I replace the call to printf with a call to "fprintf (stderr, ...)" and redirect stdout and stderr to the file xxx? Explain.

Problem 3 (0 Points) (From Tanenbaum, modified)
When a new Unix process is created by forking, it must be assigned a unique integer as its PID. Typically, a Unix kernel assigns a new PID using a 16-bit unsigned integer counter that indicates the PID of the newest process.

a) The kernel can't just increment this counter and assign the value as the PID of the next new process. Why not?

b) Give an algorithm that uses the counter for properly assigning a unique PID.

c) Where is the PID of a Unix process stored?

Problem 4 (0 Points)
The Linux man page boot(7) describes five steps involved in booting a Unix system.

a) Summarize what is done in the first three steps.
b) When booting most operating systems, the bootstrap loader in sector 0 of the boot disk first loads a boot program which then loads the operating system. Why is a multi-step boot procedure used instead of a single-step one?

**Problem 5 (0 Points)**

The Linux manual page `environ(5)` describes environment variables.

a) What is a login shell?

b) What is an environment variable and how are they different than other shell variables?

c) How does Linux determine the value of the environment variables `HOME` and `SHELL` in a login shell?

d) How would you set the value of the variable `TRACE_OPTS` to `-g -r` and put it in the environment when using the Bourne shell `sh`? The Bash shell `bash`? The C shell `csh`?

**Problem 6 (4 Points)**

The last problem describes the shell language `xsh`, `xsh`, like all other shells (e.g., `sh`, `csh`, `tcsh`, `bash`), interprets `$` as the PID of the current process. The `kill` command (see `kill(1)`) sends a signal to a process. For example, `kill -STOP $` puts the current process to sleep by sending a STOP signal to itself. The `-CONT (continue) signal will resume the process. Other signals are described in `signal(7)`.

See `sh(1)`, `kill(1)`, `chmod(1)` and `signal(7)`. Specifically in `sh(1)`, the `sh` symbol & (background); `sh` parameter `$` (current process PID); the interpreter specification `#!`.

a) Write the simplest `xsh` shell scripts X, Y, and Z that will create the following process hierarchy:

- The `xsh` process that interprets the script "X" is the parent of the `xsh` process that interprets the script "Y".
- The `xsh` process that interprets the script "Y" is the parent of two instances of `xsh` processes that each interprets the script "Z".
- All processes display on `stdout` their process ID ($ is the PID of the process) as their first action.
- All processes put themselves to sleep as their last action.

Note that you can assume that `xsh` is executed within a standard shell and therefore, if the first line begins with "#!", the remainder of the line contains the pathname of the interpreter (e.g., `#!/usr/home/kenw/bin/xsh`).

Submit a listing of the three scripts X, Y and Z. Note that although this is a paper and pencil exercise, you should still submit a printed solution.

b) The Bourne and bash shells have a syntax that is almost identical to `xsh`. Run your scripts in Part a using the Bourne or bash shell and determine how killing of the `xsh` process that interprets the Y script effects the process hierarchy in a real shell. (Note: If process Y has PID 7777, `kill 7777` terminates process Y. The `ps` command indicates the process hierarchy.) Submit an explanation of what happens to the process hierarchy.
Problem 7 (6 Points)

The course Web page contains the file npipe3-puzzle.c and its header file stdinc.h. At one time, the program solved the npipe problem described below but for the limited case of $N = 3$ processes (i.e., a pipeline of two pipes). But the student dropped the npipe3.c source code on the floor, and some lines got scrambled.

Here is the description of the more general npipe program It is called like this:

\[ \text{npipe [-n N]} \]

where $N$ indicates the number of instances of npipe. Basically, when run it forms a pipeline with $N - 1$ pipes connecting $N$ instances of the npipe process: "npipe | npipe | ... | npipe" The $k$th process creates the pipe to the $(k+1)$th process and forks the $(k+1)$th process. As a verification that each process outputs the same bytes, each process computes the sum of all bytes treating each byte as an unsigned integer value. This checksum is printed on stderr by the main routine before it exits.

The program npipe3 is just npipe with $N=3$. You are to unscramble the code so that it will compile and run properly. We know the following:

- See the comments in the beginning of the npipe3-puzzle.c code for the usage syntax.
- In its correct order, the program compiled using "g++ -g -o npipe3 npipe3.c" and ran correctly on a Linux 2.6 system. There may be minor changes that are needed to get it to compile on other systems.
- The file stdinc.h is correct.
- The only lines that got scrambled are in the following functions: pipe stdout, pipe stdin, do_pipe, and do_jo.
- Debugging output can be enabled by uncommenting the line of code debug = 1; in the main routine.

Submit the following:

- The source program listing for the working version
- The output for some test cases when in debug mode
- An explanation of why the output indicates that the program is running correctly
- The output of the command "diff npipe3-puzzle.c npipe3.c" where npipe3.c is the working code and npipe3-puzzle.c is the original code.
Problem 8 (10 Points)

The course Web page has a link to the source code for the test harness for xsshA. xsshA is a very simple shell language which is a subset of the language xssh which will be implemented in Project A. It is a test harness in the sense that the command sequence is hardcoded into the simple two-dimensional array cmd[][i] where cmd[i] points to the ith command and cmd[i][j] points to the jth word of command i.

In the description below, a proper word indicates a metasymbol, square brackets ([ ]) indicate an optional word(s), and "..." indicates 0 or more words. xsshA supports the following builtin (internal) commands:

- **echo [word] . . .** Display the arguments followed by a newline. Multiple spaces-tabs should be reduced to a single space.
- **quit N** Quit the shell with an exit status of N.
- **wait N** The shell should wait for process N to terminate.
- **set Name Value . . .** Set a variable name to a value. A user-defined variable name is a sequence of letters, digits and underscore. There are three special variable names described later: question mark (?), dollar ($), and exclamation (!). The value of a variable is indicated by preceding the name with the dollar sign. For example, 'set XY 32' sets the variable XY to the string 32. The value of variable XY is denoted by $XY. If there is more than one Value argument, the values are concatenated together to form a single value; i.e., 'set X 32 ABC' is equivalent to 'set X 32ABC'. 'set X ABC $3' sets the value of X to the concatenation of the string "ABC" with the value of the variable 3.

All other commands are assumed to be executables in a directory listed in the PATH environment variable.

Here are the other features of xsshA:

- a) The command line prompt should be the three character sequence '>> ' (i.e., >, >, space).
- b) A non-builtin command is assumed to be a Unix executable that can be found in a directory listed in the PATH environment variable.
- c) $? is the exit status of the latest process. $$ is the process number of the shell. $! is the process number of the last backgrounded process.
- d) All undefined variables have a value of the null string.
- e) The $ symbol only has a special meaning when it is the first character of a word and it is interpreted as meaning that the value of the variable is desired. So, $X$YZ is the value of the variable X$YZ because $ has no special meaning unless it is the first character of a word.
- f) There are no explicit environment variables available to the user.
- g) An ampersand character (&) at the end of a line indicates that the command should be run in the background.

Note that there is very simple variable substitution, but there is no filename substitution nor command substitution. See fork(2), waitpid(2), execvp(2), sh(1), gettimeofday(2), exit(3). The course Web page also contains links to source code that might be useful to you.
You should fill in the test harness `xsshA.c` so that it can interpret the `xsshA` language. Note that the code recognizes two flags: `-x` and `-d`. The `-x` flag indicates that the command line AFTER variable substitution is displayed. The `-d` flag indicates that debugging output should be displayed on stderr. **When debugging is turned on with `-d`, the values returned from each major system call (e.g., fork, wait, exec) should be displayed even if the value is returned in the parameter list (e.g., waitpid) and the input parameters to every call to an `exec` function should be displayed.** The debug output should be labeled with the variable names when appropriate so that it is clear what variables are associated with what values. Choose a format that is simple but easy to read.

Submit the following:

a) Your source code.

b) The output of the test harness when run with the `-x` and `-d` flags.

c) For each command, indicate whether your code is working properly or not. If not, indicate what is wrong and what needs to be done to fix the bug(s).