Problem 1 (0 Points)
Consider the following code fragment:

```c
printf("mypid = \%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 2 (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 3 (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 4 (0 Points)

The Linux man 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 5 (4 Points)

A C++ program called `forkit.c` is available from the course Web page. You should compile and run the program, and then, answer the following two questions.

a) Even though the child process sleeps for 2 seconds before printing out the value of `x`, it doesn’t see the final value of `x` seen by the parent. Why?

b) What does the output indicate about which process gets control after a call to `fork(2)`?

Submit both the output and your explanation.

Problem 6 (6 Points)

Write a C/C++ test program whose executable is called `t_exec` which will time fork-exec commands entered on stdin. The program should be able to fork-exec any program found in your PATH environment variable. You can use your own parser or the one supplied on the course Web page. The optional command-line argument `-v` indicates verbose mode:

```
t_exec [-v]
```

In non-verbose mode, your program should just execute the command and output to stderr the elapsed time of the command. In verbose mode, your program should output to stderr, the pid of the child process and the arguments being passed into the `execvp(2)` call. See `fork(2)`, `waitpid(2)`, `execvp(2)`, `sh(1)`, `gettimeofday(2)`, `exit(3)`.

Submit a C/C++ source program listing, the output for some test cases when in verbose mode, and an explanation of why the output makes sense.
**Problem 7 (4 Points)**

Homework 4 describes the shell language `xshA`. `xshA` 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 `xshA` shell scripts X, Y, and Z that will create the following process hierarchy:

- The `xshA` process that interprets the script "X" is the parent of the `xshA` process that interprets the script "Y".
- The `xshA` process that interprets the script "Y" is the parent of two instances of `xshA` 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 `xshA` 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/xshA`).

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 `xshA`. The biggest difference is the placement of the `&` symbol denoting backgrounding. Run your scripts in Part a using the Bourne or bash shell and determine how killing of the `xshA` 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.

**Extra Credit**

**Problem 8 (6 Points)**

Write an interesting homework problem and its solution. Explain why you think the problem is interesting. I will interpret interesting here to mean some of the following about the problem:

- Can be done in a short amount of time.
- Exposes something interesting about the material covered in the above problems (i.e., processes, process creation, shell scripts, shells).
- Requires deep thinking.
- The solution has useful implications with respect to learning about an OS.