

## Homework 3

Reading: Tanenbaum, Sections 2.1, 2.2 and various man pages

Due: Thu, Sep. 27, 2007

**Problem 1** (0 Points) [ Gnu C Library ]

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.

**Problem 2** (0 Points) [ fork ]

Consider the following code fragment:

```
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 ...
}
```

- 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.
- 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.
- 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) [ PIDs ]

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.

- The kernel can't just increment this counter and assign the value as the PID of the next new process. Why not?
- Give an algorithm that uses the counter for properly assigning a unique PID.
- Where is the PID of a Unix process stored?

**Problem 4** (0 Points) [ boot ]

The Linux man page `boot(7)` describes five steps involved in booting a Unix system.

- Summarize what is done in the first three steps.
- 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) [ environ ]

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 6** (4 Points) [ Shell Language ]

The last problem in this homework describes the shell language `xsshA`. `xsshA`, 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 `xsshA` shell scripts `X`, `Y`, and `Z` that will create the following process hierarchy:
  - The `xsshA` process that interprets the script "X" is the parent of the `xsshA` process that interprets the script "Y".
  - The `xsshA` process that interprets the script "Y" is the parent of two instances of `xsshA` 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 `xsshA` 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/xsshA`).

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 `xsshA`. Run your scripts in Part a using the Bourne or `bash` shell and determine how killing of the `xsshA` 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** (10 Points) [ The Simple Shell `xsshA` ]

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[] []` where `cmd[i]` points to the  $i$ th command and `cmd[i][j]` points to the  $j$ th word of command  $i$ .

We use metasympols below to describe the syntax of the shell language. For example, "W" stands for a single word. A non-whitespace character (SPACE or TAB) or a newline character terminates a word. "... " indicates a continuation of 0 or more repetitions of the preceding symbol. For example, "W ..." means one or more W's. "I" stands for integer.

`xsshA` supports the following *builtin* (internal) commands:

- `echo W ...`: Display the arguments followed by a newline. Multiple spaces/tabs should be reduced to a single space.
- `quit I`: Quit the shell with an exit status of  $N$ .
- `wait I`: The shell should wait for process  $N$  to terminate.

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) `#!` should be treated as the process number of the last backgrounded process and has an initial value of the null string.
- d) An ampersand character (&) at the end of a line indicates that the command should be run in the background.

Note that there is almost no variable substitution except for `#!`, and there is no filename substitution nor command substitution. See `fork(2)`, `waitpid(2)`, `execvp(2)`, `sh(1)`, `gettimeofday(2)`, `exit(3)`.

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 should be displayed after replacing `#!`. 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).