

Homework 3*Reading: Tanenbaum, Sections 2.1, 2.2, 10.1-10.3**Due: Feb. 15, 2006***Problem 1** (0 Points)

In each case below, write a shell script that produces the output described below and an example of its execution. In all cases, input is from stdin and output is to stdout. You can assume that all command-line arguments are valid; i.e., you do not have to check the validity of command-line arguments.

See sh(1).

- List all files in the current directory beginning with a user-specified character sequence and ending in another user-specified character sequence. The beginning and ending sequences are given as the first and second command-line arguments. Example: `"scripta xx .c"` prints out all files beginning with `"xx"` and ending in `".c"`.
- The script in Part a can also be used when either of the arguments is the `"*"` metacharacter if it is properly quoted (the metacharacter indicates a "don't care" sequence). Example: `"scripta '*' .c"` prints out all files ending in `".c"` (Note the quoting of `"*"`). Explain why the script in Part a still works.
- Extend the script further to allow 0, 1 or 2 arguments. When there is just 1 argument, it should be assumed that the beginning sequence argument is `"*"`. Example: `"scripta .c"` prints out all files ending in `".c"`; i.e., is equivalent to `"scripta '*' .c"`.

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.

- 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 3 (0 Points)

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 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 (3 Points)

Shells (e.g., `sh`, `csh`, `tcsh`, `bash`) interpret `$$` as the PID of the current process. The `kill` command (see `kill(1)`) sends a signal to a process. For example, `'kill -STOP $$'` 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 Bourne or bash shell scripts that will create the following process hierarchy:
 - A is the parent of B.
 - B is the parent of two instances of C.
 - All processes record their process id in the file `pid.$$` (`$$` is the PID of the process) as their first action.
 - All processes put themselves to sleep as their last action.

Submit a listing of the three scripts A, B and C.

- b) Run the script in the background and determine the process IDs of each process. What is the relationship of the PIDs?
- c) How does the killing of the B process effect the process hierarchy? (Note: If process B has PID 7777, `'kill 7777'` terminates process B.)

Problem 6 (2 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 explanations.

Problem 7 (4 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 [-x]
```

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.

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.

See `fork(2)`, `waitpid(2)`, `execvp(2)`, `sh(1)`, `gettimeofday(2)`, `exit(3)`.

Problem 8 (8 Points)

We consider the design of an interpreter for the simple shell language `xssh0`. In the description below, a proper word indicates a metasymbol, square brackets (`[]`) indicate an optional word(s), and `"..."` indicate 0 or more words. `xssh0` allows only one command per line and supports the following *builtin* (internal) commands:

- `chdir [Pathname]`: Change the current directory to *Pathname* which can be an absolute or relative Unix pathname. If *Pathname* is not given, change the current directory to the path given by the environment variable HOME. The current directory should be maintained in an environment variable called PWD.
- `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*. If *N* is omitted, the exit status is that of the last command executed.
- `"bg Cmd . . ."`: The remainder of the line should be run in the background with the word following `bg` treated as a non-builtin command.
- `"wait"`: The shell should wait for all backgrounded processes to complete.
- `"pause Pid"`: The process with PID *Pid* should be put to sleep.
- `"resume Pid"`: The process with PID *Pid* should be allowed to continue to run.
- `"set Name Value"`: Set a variable name to a value. A user-defined variable name is a sequence of letters, digits or underscores. There is one special single-character variable name described later: question mark (?). The value of a variable is indicated by preceding the name with the dollar sign. For example, 'set XYZ 32' sets the variable XYZ to the string 32. The value of XYZ is denoted by \$XYZ. Note that the variable name is the longest possible name (e.g., if there are variables X and XY, \$XY refers to the variable XY).
- `"export [Name] . . ."`: The names are exported to the current environment (and subsequent children). If there are no arguments, list the exported names and their values.
- `"include Name"`: Read and execute commands from the file *Name*. The search path \$PATH is used to find the directory containing *Name*.

Here are the other features of `xssh0`:

- a) Multiple spaces/tabs are reduced to a single space during the substitution and line scanning phase.
- b) The command line prompt should be the three character sequence '>>' (i.e., >, >, space).
- c) A child process inherits all of its parent's environment variables.
- d) `XYZ` is the string resulting from the concatenation of the values of the variables `XY` and `Z`.
- e) A non-builtin command is assumed to be a Unix executable that can be found in a directory listed in the `PATH` environment variable.
- f) The exit status of the latest process is stored in the variable `?`.
- g) All undefined variables have a value of the null string. By *convention*, we use `?` as a special variable whose value is always the null string and is used to terminate a variable name. (e.g., `X.Y` is the value of the variable `X` concatenated with the string "Y").

Note that there is 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)`, `getenv(3)`, `putenv(3)`, `chdir(2)`, `kill(2)`, `fgets(3)`.

The main processing loop of the `xssh0` interpreter looks like this:

```
Initialize;
Process command-line args;
while ( (line=getcmd()) != EOF ) { // prompt&get cmd until EOF
    (nwords, word[]) = parse(line); // word[i] points to ith word
    word[] = do_subst(word[]); // do var substitution
    quit = eval_builtin(nwords,word[]) if (is_builtin(word[0]));
    status = eval_nonbuiltin(nwords,word[]) if (!is_builtin(word[0]));
} until (quit);
Cleanup;
```

Submit the following:

- a) A summary of the primary abstract data types that will be needed to implement the interpreter. For each abstract data type, provide a description of the valid operations.
- b) The pseudo-code for the two functions `eval_builtin` and `eval_nonbuiltin`. You can assume whatever utility functions are required to simplify the code. Furthermore, assume that there is a wrapper function that catches and handles errors for each Unix system call or library function, and its name is identical to the supplied function except that the first letter is uppercase (e.g., `Fork` for `fork`). The course Web site contains the beginnings of a solution and a set of abstract data types that you can assume are available to you.
- c) Provide a short description of each user-defined function in Part b. that is not a Unix system call or library function.