Problem 1 (0 Points)
In each case below, provide a script that provides the output described 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).

a) 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'.

b) Extend the script in Part a so that an '*' character indicates a "don't care" sequence. Example: 'scripta * .c' prints out all files ending in '.c' (Note the quoting of '*').

c) 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.

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 (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 (3 Points)
On the course Web page is a shell script called `p6.sh` that is related to the script in the preceding problem.

a) What is the purpose of the script?

b) Submit output from the script and how the output supports your answer in Part a.

See `sh(1)`{file name expansion, `for, case or if` and `test, $$`}.

Submit a source listing of your script, an example of test output, and an explanation of why the output demonstrates that your script is operating correctly.
Problem 7 (4 Points)

Write a C/C++ program whose executable is called \texttt{t\_proc} to measure the average time in microseconds over N iterations for a parent process to execute the system call sequence \texttt{fork-execvp-waitpid}. The command-line argument \texttt{-n N} (N is a positive integer) indicates the number of iterations and is an optional argument with a default value of 1,000. The usage is:

\texttt{t\_proc [-n N]}

\begin{itemize}
  \item a) Submit a C/C++ source program listing and the output for the simple test case of N=2 and where the child displays its PID before exiting and the parent displays the pid and exit status of the child process after the child terminates.
  \item b) Submit the measurement results when the child process just exits (no output) for the cases N = 1,000, 2,000 and 4,000 where each case is repeated three times.
  \item c) How do the times in Part b compare to the time to do a system call from the preceding homework? Explain why this result makes sense.
\end{itemize}

See \texttt{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 \texttt{xssh0}. \texttt{xssh0} allows only one command per line and supports the following \texttt{builtin} (internal) commands:

\begin{itemize}
  \item \texttt{chdir <Pathname>}: Change the current directory to \texttt{<Pathname>} which can be an absolute or relative Unix pathname. If \texttt{<Pathname>} is not given, change the current directory to the path given by the environment variable \texttt{HOME}. The current directory should be maintained in an environment variable called \texttt{PWD}.
  \item \texttt{environ}: Display all of the environment variables; i.e., the name-value pairs, one per line just like \texttt{printenv} does. There are only three environment variables: \texttt{HOME}, \texttt{PWD} and \texttt{PATH}. \texttt{PATH} has the same format as in any other shells; i.e., ':' separates directory pathnames.
  \item \texttt{echo <Word> <Word> ...}: Display the arguments followed by a newline. Multiple spaces/tabs may be reduced to a single space.
  \item \texttt{quit}: Quit the shell.
  \item \texttt{"bg ..."}: The remainder of the line should be run in the background with the word following \texttt{bg} treated as a command.
  \item \texttt{"wait"}: The shell should wait for all backgrounded processes to complete.
  \item \texttt{"pause <pid>"}: The process with PID \texttt{<pid>} should be put to sleep.
  \item \texttt{"resume <pid>"}: The process with PID \texttt{<pid>} should be allowed to continue to run.
  \item \texttt{"status"}: Display the status of child processes. The display should show for each process the PID, the PID of the parent process, and the run state ("RUNNING" or "PAUSED");
  \item \texttt{"set"}: Set a variable to a value. A variable name begins with a letter and consists of letters, digits and the underscore character. For example, \texttt{set XYZ 32} sets the variable XYZ to the string 32. The value of XYZ is denoted by \texttt{$XYZ}'; i.e., $ denotes "the value of". Note that the variable name is the longest possible name.
\end{itemize}
Here are the other features of \texttt{zsh0}:

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 pathname of the current directory followed by the three character sequence '>>' (i.e., $>$, $>$, space).

c) Define the \textit{root shell} to be the first instance of \texttt{zsh0} and all other instances of \texttt{zsh0} to be \textit{subshells}. The root shell inherits \texttt{HOME}, \texttt{PWD} and \texttt{PATH} from its parent process.

d) $XY$ is the string resulting from the concatenation of the values of the variables XY and Z.

e) Curly braces are used to disambiguate variable names. So, $XYZ$ is the string resulting from the concatenation of the value of the variable XY with the character Z.

f) A non-built-in command is assumed to be a Unix executable that can be found in a directory listed in the \texttt{PATH} environment variable.

g) The exit status of the latest process is stored in the variable $?$. 

Note that there is simple variable substitution, but there is no filename substitution nor command substitution.

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) For each built-in command, give code sketches for processing the command. Note that it is not expected that the code fragments will compile at this point. You can assume whatever utility functions are required to simplify the code, and you can assume that the input contains only valid commands (i.e., you don't have to check for syntax errors). But if the function is not a standard Unix system call or library function, you must indicate in the right hand margin that you are using such a function by providing the name enclosed in square brackets (e.g., \texttt{[getTokens]}, \texttt{[isBuiltIn]}). 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 function listed in the right margin in Part b.

d) Give the code fragment for processing non-built-in commands.