Homework 2 with SOLUTIONS
Operating Systems, V22.0202
Fall 2007, Professor Yap

Due: Mon Oct 1
SOLUTION PREPARED BY Instructor and T.A.s

- Please read questions carefully. When in doubt, please ask.
- The written homework is to be submitted in hardcopy during class, but the programming part sent to us in a single file (as detailed below) by midnite.

Question 1 (10 Points)

To understand Unix, you need to know its special characters. For each of the following, describe one situation where they are used. They are all related to processes.

ANSWER:

CHARACTER	NAME	ANSWER
cntl-D	control-D	to signal end of input file (e.g.. terminal)
cntl-Z	control-Z	to suspend a process
|	pipe	pipe between two processes
<, >	redirections	redirect input (<) or output (>)
&	ampersand	to run a process in the background

♢

Question 2 (28 Points)

For each of the following shell commands, (1) describe the meaning of the command, and (2) describe a situation where it is useful. Note that you can string several shell commands into one command if you separate them with a semicolon (;).

HINTS: We suggest you actually test these commands on your system. In Unix systems, you can get an online manual for a particular `COMMAND` by typing `man COMMAND`. For instance,

```
> man ls
```

will tell you all the options available for `ls`. Also try `man man`.

1. `ls -a`
   ANS: List all files in current directory, including hidden (dot-)files
2. `ls -tl /bin`
   ANS: List files in directory /bin in long format, most recent first
3. `ls /bin/m*`
   ANS: List files in directory /bin whose name begins with "m"
4. `ls -sF`
   ANS: list in current directory, indicating their size and also their type (executable, directory, links, etc)
5. `ps -s`
   ANS: show summary of processes
6. `ps -sW`
   ANS: as in previous item, but include processes related to Windows as well (not just cygwin).
7. `echo "Reading This File"`
   ANS: print the string "Reading This File" on the standard output

---

8. `which ls`
   ANS: print the full path name for the executable file of "ls"
9. `file hw1.c hw1.o hw1.exe`
   ANS: determine the file type of hw1.c, hw1.o, hw1.exe. "file" might figure that hw1.c is a C program, hw1.o is an object file and hw1.exe is an executable file.
10. `wc hw2.c`Makefile
    ANS: do a "word count" of the file hw2.c. It gives 3 numbers: number of lines, number of words, number of characters. This is useful for text files.
11. `pushd "/cygdrive/c/Program Files"; ls`
    NOTE: this will not work if you are not in Cygwin. Instead use `pushd /usr/bin; ls`.
    ANS: First, you push your current directory on to a stack of "working directories", then you cd to the directory "/cygdrive/c/Program Files" and push that on top of the stack of "working directories". Finally you list the files in the current directory.
12. `popd`
    ANS: You pop the top of the stack of "working directories" and make the current working directory to be the new top of the stack.
13. `alias ls="ls -sF --color=tty"`
    ANS: From now on, if you type "ls" command, it will be as if you had typed "ls -sF –color=tty"
14. `ln -s "/cygdrive/c/Program Files/Vim/vim71/gvim.exe" .`
    ANS: place a symbolic link in the current directory to the indicated file (/cygdrive/c/.../gvim.exe). The link is also named gvim.exe.

♢

Question 3 (5 Points Each)

- (i) Exercise 2.5, page 73 of Silberschatz. (File Management)
  What are 5 major activities of an OS with regard to file management?

  SOLUTION: Create files, delete files, open files, read/write files, close files. Note that files include directories. Also: copy and move files, rename files, set file attributes.

- (ii) Exercose 3.1. page 116 of Silberschatz. (scheduling)
  Describe the differences among short-term, medium-term, and long-term scheduling.

  SOLUTION: The jobs that are ready to run "immediately" are kept in the READY QUEUE. Scheduling or choosing processes from this queue is called **short-term scheduling** or **CPU scheduling**. There might also be a queue of jobs on the disk that need to be loaded into memory and placed in the READY QUEUE. Scheduling this transfer into the READY QUEUE is called **long-term scheduling** or **job scheduling**. Finally, some jobs may be moved out of the READY QUEUE into some intermediate queue (and possibly swapped out of memory as well). The purpose is to remove excess multiprogramming. Scheduling such jobs is called **medium-term scheduling** because, presumably, these processes have priority over jobs in the long-term queue.

- (iii) Exercise 3.2. page 116 of Silberschatz. (context-switching)
  Decribe the actions taken by a kernel to context-switch between processes.

  SOLUTION: The context of the current process must be saved. This information includes the values of: (1) process state (new, ready, running, etc), (2) program counter, (3) CPU registers, (4) scheduling info (5) memory-management info, (6) I/O status info (open files, allocated devices, etc). The information is stored in the PCB of the process. We then restore the context of the process that is to run next, and gives control to this process.

♢

Question 4 (70 points, Programming Part) Write a toy shell program called `tsh.c` that executes unix commands. It should know how to process command lines containing the special tokens "&" and ";", but not "|".

We will provide the basic routine called `parser` and also the skeleton of `tsh`. HINT: we suggest that you first implement "&" to make sure it works correctly before implementing ";".

Extra Credit Features:
(1) Allow the user can change the prompt string.
(2) Can you make "cd" work properly?

Enclosed is a tar file called "hw2.tar" that contains the files

`Makefile-hw2, parser.c, tsh.h, tsh.c`

When you have unpacked them, we suggest you rename `Makefile-hw2` to `Makefile` (why?) so that you can easily use it. You are encourage to modify and add to this Makefile to suit your needs.

You should not modify the parser files, but just use them in `tsh.c`. We have provided the bare skeleton in `tsh.c`, which you need to embellish in order to complete the assignment in this question.

We want you to do separate compilation of `parser.c` and `tsh.c` (see the Makefile to see how it is done).

SUBMISSION REQUIREMENTS: Up to 20 percent of this question may be deducted for non-compliance. These rules apply for all programming assignments (with the appropriate modifications).

- Make sure that your program is well-documented (use long and short comments as appropriate)
- Include a README file (you can copy it from previous homework!). As usual, the file should contain
  (1) your name,
  (2) NYU ID,
  (3) email,
  (4) phone Contact,
  (5) the operating system environment for your work,
  (6) acknowledgement of sources (otherwise it would be plagiarism),
  (7) any collaboration with others (if none, please say so),
  (8) any other information you like us to know,
  (9) and include the following statement: "*This submission represents my own work.*"
- To submit, you should send us one single tar file. Create a tar file called `h2.tar` containing
  `README, Makefile, parser.c, tsh.h, tsh.c`
  and any other necessary files you need. (Type "tar cvf h2.tar README Makefile ..." to create this tar file. Better still, make this task a target in your Makefile!)
- Send `h2.tar` to the grader (but cc to me). After we untar your file, we expect to type "make" to compile `tsh.c`, and to type "make test" to run the compiled program.
- If you need to resend for any reason, you must again send a single complete tar file. E.g., you forgot to attach a file `foo` the first time, DO NOT simply send us the file `foo`. You must re-package everything and re-send us the new tar file.

HINTS:

- Make sure that your shell will not get confused if we type an empty line.
- If you find that the output from your shell is not printed in the sequencing order you expect, try this: after each printf command, immediately issue the command `fflush(stdout)`. This is because the standard output is buffered, and fflush will force the buffer to be cleared.

- The most useful `exec*` version for this problem is `execvp`. You can use the arrays `args` returned by the `parser` function as follows: `execvp(args[0], args)`.
- For testing purposes, it is helpful to know that you can directly construct your own arguments for `execvp` as follows:

```
main() {
    char *prog = "ls";
    char *argv[] = { "ls", "/etc", ".", NULL };
    ...
    execvp(prog,argv);
    }
```

- In Cygwin, the wait() function appears to to get confused if you use the background command ('&'). Namely, if you issue a command such as `ls &`, then the subsequent output gets messed up.

  EXPLANATION: Normally, if a command is not to be executed in the background, you perform a wait(NULL) on the child process that is executing the command. If you issue a background command ('&'), then the parent process does not wait for the child.

  But it appears that after the first background command, the parent no longer waits for the child EVEN if you do not issue the background command! You see this because the prompt is immediately printed after each command, as if there is no wait.

  WHAT HAPPENS is this: in fact, the parent process did execute a wait() as it should. But it gets an immediate return from the previous child that was NOT waited for. E.g., suppose the child processes are denoted C1, C2, C3, etc. Assume C1 is run in the background but subsequently C2, C3, etc are run in the foreground. So you expect

```
C1, no wait
C2, wait (= wait2)
C3, wait (= wait3)
etc.
```

  Unfortunately, wait2 actually gets the return code of C1, not of C2. Similarly wait3 gets the return code of C2, not of C3. Since C1 has terminated before we executed wait2, in effect wait2 did not have to wait at all! Similarly, wait3 did not have to wait al all.

  SOLUTION: To avoid this problem, we remember the child's PID, and we explicitly wait on this PID. Note that wait() returns the process of the terminated child, and we check to see that this is the child we were waiting for.

♢