Assignment 2

Read the article on pair programming (Williams and Kessler, 2000, available through the library or Blackboard) regardless of whether you wish to pair program in this class.

Part I – These questions should be answered individually. Questions are worth 10 points each. Note that we use the convention 0x to denote a hexadecimal number, e.g. 0x1000 is 4096 decimal. As always, answer in your own words.

1. Nosek's Temple University study on pair programming showed that:
    a. the pair programming teams outperformed *every* single programmer on a timed programming task.
    b. pair programmers took 15% longer to complete the task, but created programs with fewer faults.
    c. the learning curve for unfamiliar languages and technologies was significantly reduced when learned in a pair programming environment.
    d. All of the above.
2. The article argues that:
    a. as a result of increased productivity, pair programmers get to go home early (just like kindergartners ☺).
    b. both parties should take shared responsibility for any code defects.
    c. programmers should develop a set schedule of how often to change between the roles of typist and code reviewer.
    d. pair programming is frequently effective even when the participating programmers are opposed to the concept.
3. Williams and Kessler provide the following example of the importance of communication:
    a. A study by Donadey and Mendoza showed that programmers who spend time discussing their design before starting to program tend to have fewer code defects.
    b. After removing a soda machine from a computer lab due to complaints about people talking, people started complaining about a lack of consultants to answer questions.
    c. A pair of programmers had been working separately on similar difficult problems for several weeks. At the suggestion of their supervisor, they teamed up to work on the problems together and solved both problems within one week.
4. Pair programming comes from the eXtreme Programming (XP) paradigm. XP purists recommend "flushing" any code that is not written in the presence of their programming partner. Why?

Answers in this section of Part I should be concise and to the point.

5. Privileged instructions:
   a. When can a privileged instruction be executed?
   b. What happens when a process without privilege attempts to execute a privileged instruction?
6. When cache write through is not enabled, explain briefly what will happen with respect to the state of the cache and memory when the CPU attempts to write a location? Which cache issue does this affect?
7. A user calls subroutine analyze(data, elements) where data is a 64 bit pointer to data located at location 0x25500 and elements is a 64 bit unsigned integer with value 0x1600. Assume the following
   a. Calling convention is to push arguments in reverse order. The last item pushed onto the downward growing stack is at location 0x50000.
   b. The first instruction after the jump subroutine instruction and its datum is at location 0x4000.

   Draw a picture of the stack once the call has been set up but prior to the actual invocation. Be sure to include the addresses of each item.
8. How does a virtual machine operating system determine whether or not a privileged instruction originating from one of its guest operating systems should be executed or if a privilege violation should be generated?

Part II – This programming assignment may be done on your own or with ***one other student***.

In this assignment, you implement the My Underachieving Shell (MUSH) that is based on your previous assignment. The starting point for this assignment is to use your tokenizer to parse out lines read from the user or a redirected file. Your token list should then be analyzed for commands which are separated by command terminators (semicolon and pipe: ; |). Each command will be executed by a function in file command (.c, .cpp, .C, etc.) with signature:

```
void execute_commands(token *list)
```

where token may be substituted with whatever type used for your linked list of tokens. Commands will be executed according to the following criteria:

- Commands containing tokens for file redirections "<" or ">" will be stripped of the file redirect and its argument. Commands containing the background operator "&" will be stripped of the background operator. For example: [{cat}{bozo.tzt}, {>},{clowns.txt}] becomes : [{cat}{bozo.tzt}]. In both cases, the warning "IO redirection and background not implemented" will be produced.

- Commands starting with the token cd will accept exactly one argument, e.g. "cd .." or "cd cs570". If there is not exactly one argument, you should provide the error message "Accepts exactly one argument". Use the chdir system call in the man pages (man chdir) to change the directory to the requested directory. If chdir fails, write the message "Directory does not exist or is not accessible."
- Commands with the single token "pwd" will call the getcwd system call (man getcwd) to obtain the current directory. If successful, the current working directory will be printed. On error, write "Unable to obtain current directory".
- Otherwise, the command is assumed to be a program to be launched. For this you will need to execute a fork system call (man fork) to create a new process. The parent process will wait (man -s 2 wait)[1] for the child to complete and will then continue executing. The child will use execvp (man execvp) to attempt to load in the specified program and its arguments. The argv pointer in this system call is a pointer to an array of pointers that you will need to construct. argv[0] should point to your program to be executed with argv[1] pointing to the first argument, and so on. The last entry in argv must be NULL. It is acceptable to place a limit on the number of parameters in the argv array if you so desire (e.g. 100), but be sure to produce an error message if this is exceeded. Note that if you are using a symbolic debugger, you may find it easier to construct the argv command before you fork as the fork will create a new process and you would need to attach the debugger to that process.
    - If your fork fails, write "Unable to spawn program". In practice, you should not see this.
    - If the execvp does not work (e.g. it returns), print "Unable to execute %s" where %s is the program name.
    - Once the child has exited, use the return code from wait to execute either "Process exited with error" or "Process exited successfully".

For each line of input, this should be done, possibly executing multiple commands, e.g. "cd ..; pwd". As this shell is truly underachieving, when users connect commands with a pipe symbol "|", the commands should be treated as if they were separated by a ";" along with a warning "Pipe not implemented". If unlike the shell, you are an overachiever, you are welcome to implement additional functionality, but **do not include additional functionality as part of your submission**. It will create additional work for the grader who has his own programs to write for other classes.

What to turn in:

- As always, you will need to turn in a single or pair affidavit. PAIR PROGRAMMERS: Read the article **before** you start pair programming. When

[1] The -s specifies the manual section. Section two is the system programming manual. The bash shell also has a command wait in section 1 and that will be shown by default if we do not specify section 2.

you submit, submit both people's part I and one copy of part II as a single package.

- Printed submissions of your code and a test run showing functionality. Try to exercise your program reasonably well.
- Answers to the questions.

REFERENCES

Williams, L. A. and Kessler, R. R. (2000). All I really need to know about pair programming I learned in kindergarten. *Comm. ACM* **43**, 108-114.