**CS 410 (Park)** **Midterm** **Name:**

**QUESTION 1 (45 pts: 3pts each)** Give brief, but concise answers to the following questions.

• If you had to group *IEEE POSIX.1*, *ANSI C*, *4.3BSD*, *AT&T System V Release 4*, and *SunOS 5.3* into two nonoverlapping categories, what would they be? What was your criterion for forming the partition?

• Define *system buffering*. What are its main benefit and potential pitfall? Does UNIX allow the user to exert control over the latter? If so, how?

• How does UNIX know where the end of a file lies? Note, there are no special "end-of-file" characters that delimit a file.

• What technical feature of the UNIX file system allows it to be *hierarchical*? (Be as concrete as possible.) Is there a mechanism by which the user is allowed to violate the file system's *tree structure*?

• What does it mean for a system call to be *atomic*? What purpose does it serve?

• Define what it means for a function call to *block*. What is the default characteristic of most system calls?

• Assume some process $p$ is trying to access a file $f$. What *property* of process $p$ and which attribute of file $f$ are relevant when UNIX makes the access granting decision?

• What is the role of the *set user ID bit*? What happens when a process is `exec`ed whose binary file has the *set user ID bit* set?

• Define *canonical mode*? Which two terminal-generated inputs are of special importance to its operation? When is it useful to operate in a *non-canonical mode*?

• What is meant when one says that "`fork` returns twice?" What values are being returned?

• What is a benefit of not having `fork` and `exec` merged as a single system call? What is its main drawback?

• What is the role of `vfork`? Why can we say that this system call has become "obsolete?" (Excluding possible emotional attachments.)

• Assume you are running a single process and, among other things, it performs a bunch of `printf` and `fprintf` calls. What undesirable behavior can occur if `_exit` instead of `exit` is called to normally terminate the process?

• A *session* is organized into several *process groups* each of which consists of one or more *processes*. What feature allows this organization to achieve *terminal multiplexing* (i.e., terminal sharing)? Are all process groups allowed to read from the terminal?

• What is special about the signals *SIGKILL* and *SIGSTOP*? To whom does *SIGTSTP* get delivered if one triggers it with ^Z? (Be precise in the terminology.) What is the default action associated with *SIGTSTP*?

**QUESTION 2 (30 pts: 10 pts each)** Give a detailed sketch to the following questions.

• In homework II, `get_lock` with function definition

int `get_lock`(void)

was implemented using a file semaphore ".sem_file" as a *blocking* function call (actually "pseudo-blocking"), returning 1 if acquiring the semaphore was successful, and returning 0 if repeated trials at lock attempts failed. Reimplement `get_lock` using the UNIX file locking facility. As in homework III problem 1 (extension of `cleanup`), use the `fcntl` system call in conjunction with the *flock* structure. Make sure `get_lock` remains blocking but **not** through the rudimentary means of an endless loop.

• Assume you are asked to design a library function called `spawn` with function definition

int `spawn`(const char *pathname, char *const argv[])

On success, it will **not** return but `fork` another process, explicitly set the file descriptors 0, 1, and 2 to the control terminal, then immediately call `execvp` with arguments *pathname* and *argv*. Upon error, it is to return −1. Sketch the code of `spawn` which achieves these goals.

• Consider a *job control* shell such as *C shell*. Assume user *Jane* has logged in, and only her *login shell* is running with *pid* = 10 (process ID), *pgid* = 10 (process group ID), and *fg-pgid* = 10 (foreground process group ID). Assume the following sequence of events occurs next.

```
% ls -l
^Z
Suspended
%
```

Assume `ls` was assigned *pid* = 11 and *pgid* = 11. What signal gets generated as a result of typing *control–Z*? To whom does this signal get delivered? What default action is taken by the recipient? Which signal gets generated as a consequence of this default action? To whom does it get delivered? Upon receiving this signal, what sequence of actions must the second recipient have performed in order for the *prompt* % to be displayed again, implying further commands can now be accepted from the terminal?

**QUESTION 3 (25 pts)** The following questions refer to the code shown on the next page and its output, discussed previously in class. Give brief but concise answers.

• Why is the message "Child: about to die" not output? How can it be most easily fixed?

• Why is there a second output of the message "Parent: before fork" in the second run where the only difference from the first run lies in the redirection of the standard output to the file *junk*?

• Related to the second question, why is there a single output of the message "Parent: before fork" in the first run? In general, to avoid the confusing and unintended duplication of messages, what could be done in the parent **before** `fork`ing to solve this problem?