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

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

- What features make *system calls* fundamentally different from regular *function calls*?

- Define the term *copy-on-write* (COW). Is it a feature controllable via UNIX system calls?

- Explain the meaning of an *orphaned process* and *zombie process*, respectively. What happens to them ultimately?

- Is it meaningful to speak of a *stopped foreground process group*? If so, explain why and give an example where this is the case. If not, justify why not.

- Describe the *inheritance mechanism* of signal handlers across `fork` and `exec`.

• How is the `signal` system call fundamentally different between 4.3BSD based systems such as SunOS 4.1.3 and System V based systems such as Solaris 2.3? What system call should be used in the latter in place of `signal`?

• What are the two "messy" problems that arise due to *signal interrupts* being able to occur at *any* point in time? How are these problems resolved in modern UNIX systems?

• Say p_1, p_2, and p_3 are the UNIX processes generated by the following C shell command line:

% *prog1* | *prog2* | *prog3*

Indicate in a diagram the connections set up by the shell to facilitate the *command-line* induced communication between p_1, p_2, and p_3.

• Explain the differences between *pipes*, *FIFOs*, and System V *message queues*. What are their relative merits?

• If three processes are sharing a *shared memory segment* obtained by a single execution of `shmget`, how many copies of the memory segment does the kernel maintain? What is the role of `shmat`? Why is *shared memory* the fastest IPC mechanism?

• Assume several processes are write-sharing the same *pipe* or *FIFO* for enqueueing requests to a server. What determining factor guarantees that data belonging to separate `write` operations (constituting individual messages) do not get interleaved?

• Describe briefly the role played by each of the 7 layers of the *ISO/OSI model*. (If unsure, you may leave out the *presentation* and *session layers*.) *Socket programming* is a system call interface acting between which two layers?

• Define the terms *connection* (or *full-associator*) and *socket* (or *half-associator*). What are the effects of `socket` and `bind` in this context?

• Which of the following protocols—TCP, UDP, IP—carries information enabling the *unique identification* of a process on a remote host? What is this information called? Which protocol carries information to identify which host a packet is destined for?

• What are the main differences between TCP and UDP? Give two examples of possible applications where you would choose one over the other.

• Explain the difference between a *concurrent server* and an *iterative server*. What are some criteria that come into play when deciding which one to use in designing a server application?

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

• The UNIX `passwd` command, in the course of changing a user's password, needs to write to the file */etc/passwd* (to save the new password) which has *access permission -rw-r–r–* and belongs to *root*. How does UNIX handle this contradictory requirement? Give a detailed account of the mechanisms involved. (First, explain why this is a contradictory requirement to begin with.) Does the *read permission* for *group* and *others* of the */etc/passwd* file play a significant role in causing the dilemma in the first place? What other applications may potentially benefit from the preceding security scheme?

• Consider a *job control* shell such as *C shell*. Assume user *Bill* has logged in, and only his *login shell* is running with *pid* = 100 (process ID), *pgid* = 100 (process group ID), and *fg-pgid* = 100 (foreground process group ID). Assume the following sequence of events occurs next.

[1000] % ls &
[1] 200
[1001] % ^C
[1002] % fg
ls
^Z
Suspended
[1003] % fg
ls

First of all, don't be alarmed that *ls* keeps on running and running like the rabbit in the battery commercial. Assume it's just one of those days when the *file server* is real busy. What sequence of actions does the shell perform in response command line `[1000]`? Indicate which system calls are involved. Which signal gets generated at command line `[1001]`? To whom does it get delivered? (Be careful in your terminology.) What is the signal disposition of the recepient? What sequence of events transpires in the shell as a consequence of command line `[1002]`? Again, indicate which system calls are involved. What happens when ^Z is issued? Which signal gets generated and who receives it? What "chain-reaction" of events does this in turn evoke? Be as specific as possible. What happens as a consequence of command line `[1003]`? How is it different from command line `[1002]`? Explain what additional system call is issued in the latter which was absent in the former.

• We have seen *semaphores* implemented using `open` (with options O_CREAT | O_EXCL) as *file semaphores*, and directly supported by the kernel via `sem_get`, `sem_op`, and `sem_ctl`. A third possibility is through the use *FIFO*s. Explain how this can be accomplished. Describe in detail what the codes of `get_lock` and `release_lock` would need to do. (Sketching of code is optional if your description is clear enough.) What other function besides `get_lock` and `release_lock` is needed to facilitate the present scheme? What feature of FIFOs is an absolute requirement for rendering the present scheme correct?

QUESTION 3 (25 pts: 10 + 15)

• Sketch the relevant code of your *lsh* which would handle one level of *command-line piping*. For concreteness, assume the command-line was

% ls | sort

and the program names "ls" and "sort" were parsed and stored in the string pointers *from_filename* and *to_filename*, respectively. Make your program as complete as possible, showing the exact syntax where necessary. For example, the syntax of `exec` would not be relevant here. (*Hint: Note, command-line piping is very similar to command-line redirection.*)

• Sketch the code of a *concurrent server* that mimicks the behavior of *rsh*. *rsh* is used in the following manner

% rsh bigbird ls

where the second argument "bigbird" indicates on which remote host to execute, and the third argument dictates which command to execute. Your concurrent server is the *daemon process* that runs in the background on the machine *bigbird* and listens to, say, port 6565 for any incoming request. Naturally, there must also be a corresponding client program which submits the request, say from *csa* to *bigbird*, but you may assume its existence. Just note that the client process will contact the server at port 6565, passing along a string that represents a command without any options (for simplicity). Most importantly, your design should be *logically* correct, showing all the relevant system calls involved in your code sketch. There are parts where syntax is extremely relevant, and those should not be abbreviated. (*Hint: This is just like the* `barebones_shell` *example, except for some set-up operation needed to handle the networking part. The overall concurrent server structure was discussed in class.*)