UCLA CS 111 Winter 2009
Practice Midterm Example Answers

I Processes

In the following questions, circle a *minimal* set of system calls that can implement the following command lines in a shell implemented like Lab 1's ospsh. This shell *does not* support advanced features like job control or after. Don't include system calls required only for error reporting or cleaning up zombie processes. If you think you might have circled more system calls than we expect, write down why.

command &
A. read D. fork G. signal/sigaction J. open
B. write E. dup2 H. exit K. close
C. pipe F. wait/waitpid I. execvp L. chdir

D, I

command1 | *command2*
A. read D. fork G. signal/sigaction J. open
B. write E. dup2 H. exit K. close
C. pipe F. wait/waitpid I. execvp L. chdir

C, D, E, I, K maybe F The wait/waitpid system call might be required to wait for this foreground command to complete.

command <file
A. read D. fork G. signal/sigaction J. open
B. write E. dup2 H. exit K. close
C. pipe F. wait/waitpid I. execvp L. chdir

D, E, I, J, K, maybe F, H The exit system call might be required if the input file doesn't exist.

command1 && *command2*
A. read D. fork G. signal/sigaction J. open
B. write E. dup2 H. exit K. close
C. pipe F. wait/waitpid I. execvp L. chdir

D, F, Il

cd *dir1* || cd *dir2*
A. read D. fork G. signal/sigaction J. open
B. write E. dup2 H. exit K. close
C. pipe F. wait/waitpid I. execvp L. chdir

D, F, H, L

II Scheduling

Optimizatron decides to design a variant of Shortest-Job-First scheduling that will have slightly worse average wait time, but eliminate starvation. His idea is this:

- Use a preemptive version of strict priority scheduling.
- When a request arrives, set its priority value to its amount of work. (We assume work amounts are integers >= 1.)

• Use *priority aging* to give requests higher priority the longer they wait. Specifically, every 5 time units that a job waits, reduce the job's priority value by 1, down to a minimum of 1.

His friend Bert offers to implement this strategy. Unfortunately, Bertha's implementation still suffers from starvation!

15 What did Bert do wrong? Is this a fundamental problem with Optimizatron's design, or is there a different implementation that would meet his goals?

Eventually, Optimizatron's *system will force every long-lived request to have priority value 1, the highest possible priority. Bert's implementation, which suffers from starvation, must thus indefinitely delay at least one highest-priority process. The only reason a strict-priority scheduler would not schedule a highest-priority process is because there are* other *highest-priority processes to run. The problem is analogous to round-robin scheduling: To avoid starvation, the priority scheduler must run processes with the same priority level* in FCFS order*. It looks like Bert's system runs newer processes before older ones. Optimizatron*'s *design works in general; if Bert serviced same-priority processes in FCFS order, his implementation would prevent starvation.*

The following requests arrive at a scheduler at the times indicated. The scheduler begins to execute requests at time 0. Assume that the context switch overhead is also 0.

Request	**Arrival time**	**Amount of work**
A	−2	1
B	−1	3
C	**???**	**???**

16 Running these jobs with two different cooperative schedulers gives the following average metrics.

Scheduler X Average waiting time = 4
Scheduler Y Average turnaround time = 8

Figure out schedulers and characteristics of C that produce these results. Note that C's characteristics must be the same under both Scheduler X and Scheduler Y. You may assume that C's arrival time and amount of work are integers, and C arrives at a different time than A or B. *Hints:* This restricts the possible scheduling orders, making your work easier. One of the equations you'll generate has only one variable in it: we'd advise solving that equation first.

– **Scheduler X**: *SJF* – **C's arrival time**: −4
– **Scheduler Y**: *FCFS* – **C's amount of work _** : 4

We found these values by noting that in either *SJF or FCFS, request Awill precede request B. That leaves us with three possible orders, ABC, CAB, and ACB. Average waiting time for the order ABC generates an equation with only one unknown (no other equation does). As advised, we solve this equation for W* = 4*, giving C's arrival time* −4*. This shows that the ABC order is SJF. The FCFS order corresponding would be CAB, which generates the following scheduling orders, average wait times, and average turnaround times.*

Order Wait time Turnaround time
SJF *ABC* (2 + 2 + 8)/3 = 4 (3 + 5 + 8)/3 = 16/3
FCFS *CAB* (4 + 6 + 6)/3 = 16/3 (8 + 7 + 9)/3 = 8

c) List one advantage and one disadvantage of preemptive round-robin scheduling relative to other scheduling algorithms, in terms of metrics such as average waiting time, average turnaround time, possibility of starvation, and average

utilization.

Advantage: *Very low average waiting time, no starvation (if implemented correctly), robustness.*
Disadvantage: *Lower utilization due to more context switches.*

III Synchronization

You have been hired by NASA to study Operating Systems issues in their latest Mars probe. Consider
the following situations:

a) The probe has several interconnected fuel tanks. The tanks are connected in a ring and fuel can
be pumped between adjacent tanks by any of the processes running on the probe. To avoid race conditions, processes use the following algorithm for moving fuel: lock the tank from which fuel is being moved; lock the tank to which fuel is being moved; move the fuel between tanks (increment the contents of one and decrement the contents of the other); release the locks in the reverse order. What is the most serious problem this solution has? Give an example illustrating the problem (2 pts). Suggest a way to fix it.

Answer: The solution can deadlock. Consider 2 processes, p1 & p2, p1 is moving fuel from tank 1 ->2 and p2 from 2 ->1. p1 acquires lock1; context switch; p2 acquires lock 2; context switch p1 waits for lock 2; p2 waits for lock 1 - deadlock. You can fix it by numbering the tanks and acquiring them in order.

b) This problem can be reduced to another problem that was discussed in class. Name the problem
(2 pts), and show how the two problems are equivalent.
Answer: This is the dining philosophers' problem. The processes are philosophers and the forks are tanks.

c) Because we can reduce the fuel problem to the class problem (the one discussed in class), we can use the known solution for that problem. Other than avoiding the problem you identified above, name an advantage that using the known solution has.
Answer: There are known solutions to the dining philosophers problem that get maximum concurrency - that is as many fuel transfers as possible will occur using the dining phils solution, as opposed to the numbering that may not.

d) This probe is a self-contained system being sent far from its controllers into a dangerous and unpredictable environment with the intent of reporting interesting data back to its owners. Discuss 3 ways in which the **operating system** of this probe must differ from the operating system of a general purpose earthbound computer.

Answer:
• It's not connected to a reliable, effectively infinite amount of power. Power becomes a resource to be managed, like CPU time or disk space.
• It is not intended for interactive use, so optimizations for interactive use, like aging non-interactive
jobs are unlikely to be helpful.
• Various control tasks on the probe may have hard deadlines. Scheduling mechanisms tuned for real-time deadlines are more appropriate.
• Either crises or interesting events may need to be reacted to quickly. A scheduling mechanism that allows certain tasks to have absolute priority is important.
• It's difficult to reboot a remote probe. The system must recognize that it's deadlocked and reboot itself. In general it needs to run with much less user intervention than a desktop computer.
• Probably not much use for sophisticated screen drivers or graphical interfaces.

• System components are likely to be fixed - new processes will be rare, as will new memory allocations. Schedulers and allocators can be simplified.