COS318: Midterm 2008 Solutions

I. Short Answer Questions [15 points]

a. [3 points] Do device drivers typically run in kernel mode or user mode? Why? What might you have to do in order to run them in the other mode? What would the advantage be?

b. [2 points] DMA seems like a very good idea. When is DMA not a good idea? In what circumstances would you have the CPU do the work and not include a DMA controller in a computer you design?

c. [2 points] When DMA is done from disk to memory, the disk first reads data into a local internal buffer before the DMA controller starts to move the data to main memory. Why does it not read directly from the disk to memory, without using a buffer?

d. [4 points] Describe the necessary steps that a kernel must take in the course of processing a system call.

e. [4 points]

- **i. Briefly define the following terms, using diagrams where helpful:**
  - **A. monolithic operating system**
  - **B. microkernel operating system**
- **ii. List two advantages and two disadvantages of each of the above (monolithic kernels and microkernels)**

II. Deadlocks

a. [4 points] List and define four conditions necessary for deadlock to occur in an operating system.

b. [8 points] Given the following function:

```
function f (lock A, lock B) { acquire A; ...; acquire B; ...; release A; ...; release B }
```

If f can be run by multiple threads using the same locks, is there a potential deadlock? Explain. If so, how would you fix the code?

c. [8 points] Consider the following snapshot of a system:

	Allocation				Request				Available			
	U	V	W	X	U	V	W	X	U	V	W	X
A	0	0	1	0	2	0	0	1				
B	2	0	0	1	1	0	1	0	2	1	0	0
C	0	1	2	0	2	1	0	0				

Is the system in a deadlock state? Explain why or why not? If so, which are the processes that are involved in a deadlock?

III. Locks and Semaphores [20 points]

a. [4 points] Implement the P and V functions associated with semaphores. Make sure your P and V operations do not busy wait.

b. [16 points] In this part, you are asked to solve a standard problem that is known as the "Cigarette-Smokers Problem." Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it, rolls a cigarette and then smokes it, and so on. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has only paper, another has only tobacco, and the third has only matches. Each smoker has an infinite supply of their ingredient. The agent has an infinite supply of all three materials. The agent places two of the three ingredients on the table. The smoker who has the remaining ingredient then rolls and smokes a cigarette, signaling the agent on completion. The agent waits until that smoker is done, so no two smokers are smoking at the same time. The agent then puts out another two of the three ingredients and the smoker who has the remaining ingredient rolls and smokes a cigarette, signaling the agent on completion, and the cycle repeats. Write a program to synchronize the agent and the smokers. First, think about what data structures you need and how many semaphores and/or locks. Then write the pseudo-code. Please try to make your code modular and look like real code and functions as far as possible.

sem t_p, m_p, t_m; sem done; void smoker(sem ingredients_needed) { while(1) { P(ingredients_needed); smoke(); V(done); } } void agent() { while(1) { sem* new_ingredients = pick_ingr_pair(); // picks t_p, m_p, or t_m V(new_ingredients); P(done); } }*

Three threads run smoker with t_p, m_p, and t_m. Another thread runs agent.

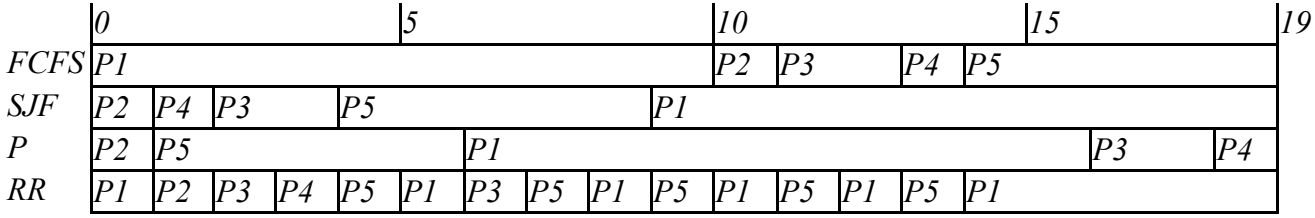
IV. Scheduling [10 points]

Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

a. Draw four timelines illustrating the execution of these processes using First-Come-First-Served, Shortest Job First, a non-preemptive priority (a smaller priority number implies a higher priority), and Round Robin (quantum=1) scheduling.



b. What is the turnaround time of each process for each of the scheduling algorithms in part (a)?

	P1	P2	P3	P4	P5
FCFS	10	11	13	14	19
SJF	19	1	4	2	9
P	16	1	18	19	6
RR	19	2	7	4	14

c. What is the waiting time of each process for each of the scheduling algorithm in part (a)?

	P1	P2	P3	P4	P5
FCFS	0	10	11	13	14
SJF	9	0	2	1	4
P	6	0	16	18	1
RR	9	1	5	3	9

d. Which of the schedules in part (a) results in the minimal average waiting time (over all processes)?

	Average Waiting Time
FCFS	*9.6*
SJF	*3.2*
P	*8.2*
RR	*5.4*

Shortest Job First has the least average waiting time.

v. Monitors [10 points]

An important operation in a database server is to atomically transfer money from one account to another. The goal is to have a highly concurrent implementation that allows multiple transfers between unrelated accounts in parallel. The following code is an attempt to implement the atomic transfer primitive:

monitor Account { int balance; procedure void withdraw(int amount) { if(amount > balance) throw new Exception("Insufficient Funds"); balance = balance - amount; } procedure void deposit(int amount) { balance = balance + amount; } procedure void transfer(int amount, Account dest) { self.withdraw(amount); dest.deposit(amount); } }

a. [3 points] Is a.transfer(10,b) atomic? Explain briefly. If not, provide an atomic implementation.

Yes. While it may be preempted and a deposit made to dest, there is no way to observe that self has made a withdrawal until the transaction is complete. In other words, there is no way to observe that the total wealth of all accounts has decreased during the transaction.

b. [7 points] Does the procedure always work correctly (i.e. succeed when it should and fail when it should)? If so, explain why; if not, show how to fix it.

No; there is a potential deadlock if, for example, a.transfer(10,b) and b.transfer(10,a) are called in parallel. In general, it is bad practice to access other monitors from inside a monitor. This can be fixed by creating a global lock around the transaction (bad for liveness):

procedure void transfer(int amount, Account dest) { acquire(transaction_lock); self.withdraw(amount); dest.deposit(amount); release(transaction_lock); }

or by moving the transaction outside of the monitor and acquiring the locks in some order:

procedure void transfer(int amount, Account source, Account dest) { if(source < dest) { acquire(source); acquire(dest); } else { acquire(dest); acquire(source); } source.withdraw(amount); dest.deposit(amount); release(source); release(dest); }

Incidentally, there is another bug in this monitor. Because it uses signed integers and does not check for insufficient funds upon deposit, transfer (and deposit) may be used to leave an account with a negative balance.

COS 318: Operating Systems
Princeton University
Department of Computer Science