

CSE 120 Fall 2007 Midterm Examination

Name _____

Student ID _____

This midterm has four questions and is worth 60 points. Please answer all questions.

1	/10
2	/15
3	/20
4	/15
total	/60

1 (10 points) Some very short questions, each worth one point.

- a) True/**False**: The wait/signal operations of a semaphore must be executed with interrupts disabled.
- b) True/**False**: If a set of threads don't deadlock, then no thread can starve.
- c) When a Java thread executes notify(), always/**sometimes**/never exactly one thread is placed on the ready queue.
- d) Using the *Shortest Job Next* scheduling policy **always**/sometimes/never minimizes the average job waiting time.
- e) True/**False**: the timer interrupt should never be masked.
- f) Consider the *Dining Philosopher's* problem. Each chopstick is numbered 0 through 4. Each philosopher needs to pick up two chopsticks. Suppose each philosopher first picks up the lower-numbered chopstick, and then the higher-numbered chopstick. Will this solution deadlock? (Yes/**No**)
- g) The monitor invariant must **always**/sometimes/never hold when a thread waits on a condition variable.
- h) The Unix fork command **always**/sometimes/never creates a new stack.
- i) True/**False**: If a thread ever executes wait(s) on a mutex semaphore s, and it doesn't eventually execute a signal(s) on that same semaphore, then there's a bug.
- j) The module of the OS that interacts with a device is called a device driver.

2 (15 points) Three short problems, each worth five points:

a) Consider the following atomic instruction *FetchAndAdd*:

```
<< function FetchAndAdd(address location) {  
    int value := *location  
    *location := value + 1  
    return value  
} >>
```

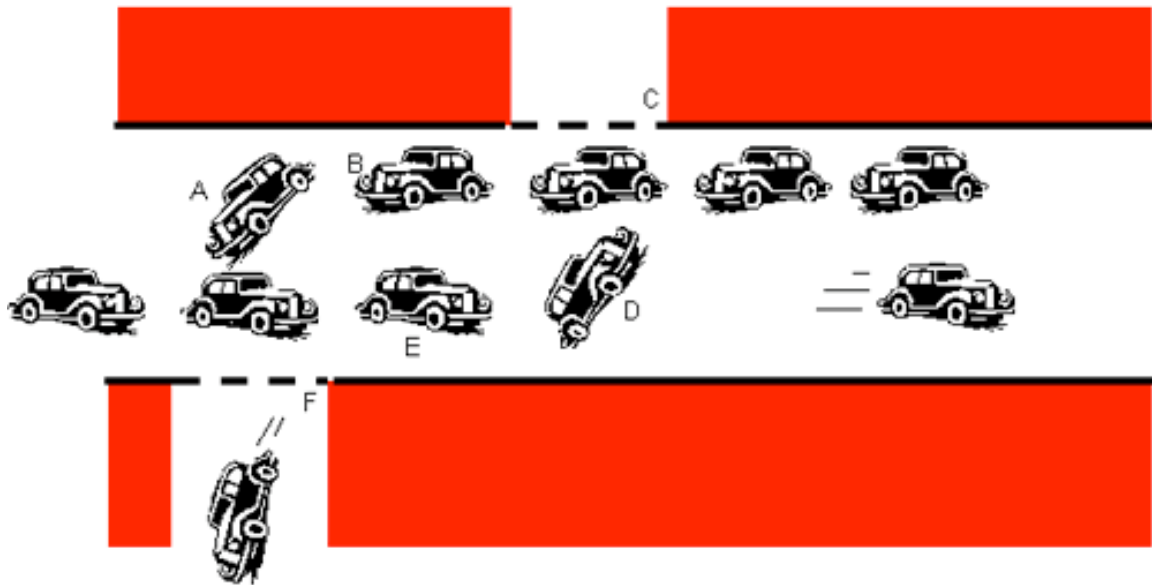
Give pseudocode for acquiring and releasing a spinlock based on *FetchAndAdd*. *Hint*: Unlike the solution, use two integers rather than one, and don't spin continuously executing *FetchAndAdd*.

Answer:

```
int in = 0;  
int lock = 0;  
  
void acquire () {  
    int me = FetchAndAdd (lock);  
    while (me != in) ;  
}  
  
void release () {  
    FetchAndAdd(in):  
}
```

Some people noted that there's no need for using the atomic instruction for releasing because mutex is already held on in. It's true. But, in practice, FetchAndAdd would also be a memory barrier, and so would be the better choice.

b) Consider the following example of *gridlock* (inspired by a leisurely pleasure drive from EBU 3B to Peet's in La Jolla Village Square):



Explain how the four necessary conditions for deadlock indeed hold in this example.

Answer:

- **Mutual Exclusion:** Each position can only hold at most one car.
- **Hold and Wait:** Each car occupies one position and wants to go forward, waiting for the car in front.
- **No preemption:** No car can be put aside or thrown out from the road.
- **Circular Wait:** A waits for F. F waits for E. E waits for D. D waits for C. C waits for B. B waits for A

c) What advantage is there in having different time-quantum sizes on different levels of a multilevel queuing system?

Having different time-quantum sizes gives several benefits. It improves fairness: a process at a lower level queue has less chance to be scheduled. But once it is scheduled, it can be executed for a longer time-quantum. It reduces context switch overhead as well. The higher priority processes are usually interactive (that's why they have the higher priority) and so run for shorter bursts. The lower priority processes would then be more batch like, and thus have longer bursts. Once a process at a lower level queue gets scheduled, the higher level queues have no ready processes to schedule. In this case, increasing the time-quantum sizes reduce the number of context switches, thus making the system to work more efficiently.

3 (20 points) Consider a *priority mutex* class *PrioMutex*. A thread obtains mutex on an object *m* of class *PrioMutex* with the method *m.lock(int p)* and releases the mutex with the method *m.unlock()*. The priority *p* can be either 1 or 2. The mutex is preferentially given to thread with higher priorities (where 2 is higher than 1). Thus, if the mutex is free then a thread calling lock will obtain it, but if the mutex is not free when a thread calls *m.lock*, then the next thread to acquire it should be one with the higher priority among those that are waiting.

Give a Java monitor that implements this class. We've helped by giving you the class outline below. You can assume that all clients give a priority *p* of 1 or 2 when they invoke *m.lock(p)*. We won't deduct points if you forget to embed any *wait()* in a *try* block. Finally, you don't have to worry about fairness in acquiring the lock, even among threads with the same priority.

```
public class PMutex {

    int held = 0;           // Lock
    int p2wait = 0;         // Number of waiting priority 2 threads

    public synchronized void lock (int p) {

        if (p==2) p2wait++;

        while (held !=0 || (p==1 && p2wait != 0))
            wait();

        if (p==2) p2wait--;
    }

    public synchronized void unlock () {

        held = 0;

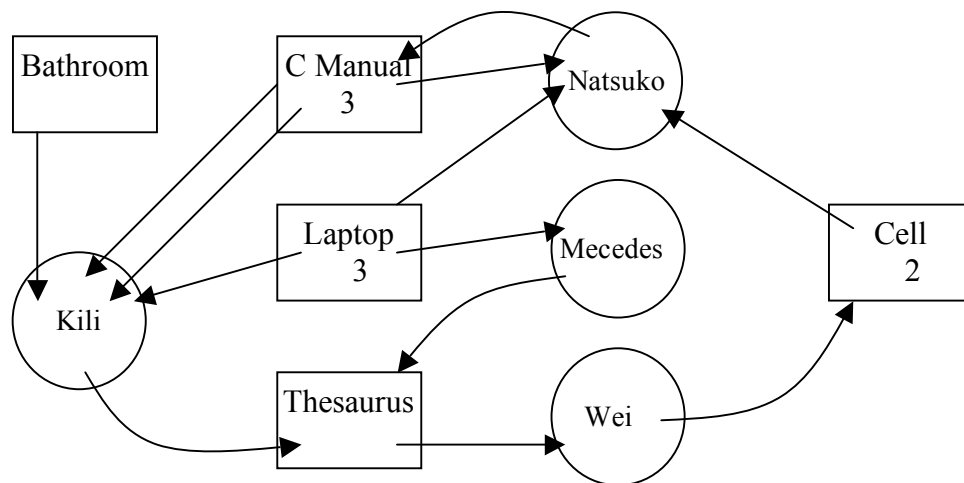
        notifyAll ();
    }
}
```

4 (15 points) Four students in CSE 122: *Operating Systems For Fun and Profit* are working as a team on the first programming project. They have found a small room in the basement of EBU 3B in which to work. The room has three laptops, three C programming manuals, two cell phones, a thesaurus, and a bathroom (which can hold only one person). Each of the four students is working on his or her piece of the project:

- Wei (who is working on the writeup) is using the thesaurus and needs a cell phone.
- Natsuko (who is working on the drivers) is using a cell phone, a laptop, a C programming manual, and needs another C programming manual because she's getting lost flipping between two different sections.
- Mercedes (who is working on the API) is using a laptop and needs the thesaurus to come up with more evocative procedure names.
- Kili is in the bathroom with a laptop and two C programming manuals (don't ask), and he needs the thesaurus before he's done with whatever he's doing in there.

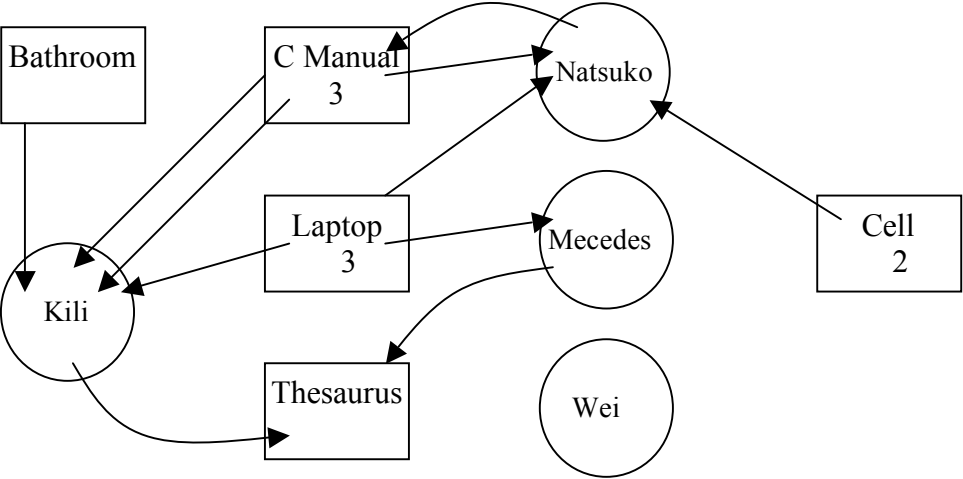
a) Draw a resource allocation graph for the above situation and use it to show whether or not the four teammates are in a deadlocked state.

Answer:

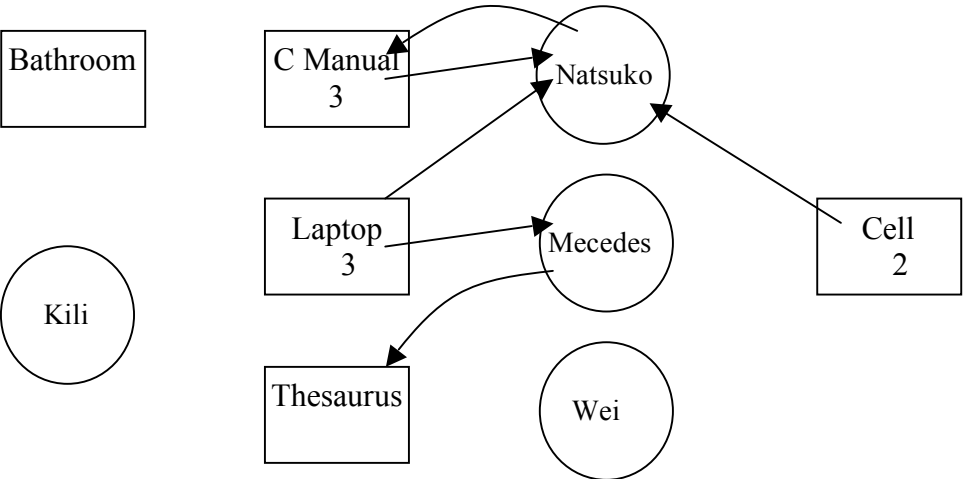


They are not in a deadlocked state. Wei only needs a cell phone which is free at that time. His request can be served. Once Wei is done, the thesaurus is free, which means Kili can proceed. Once Kili is done, the thesaurus will be free again, together with the Programming manual. Therefore, both the requests from Matsuko and Mercedes can be served.

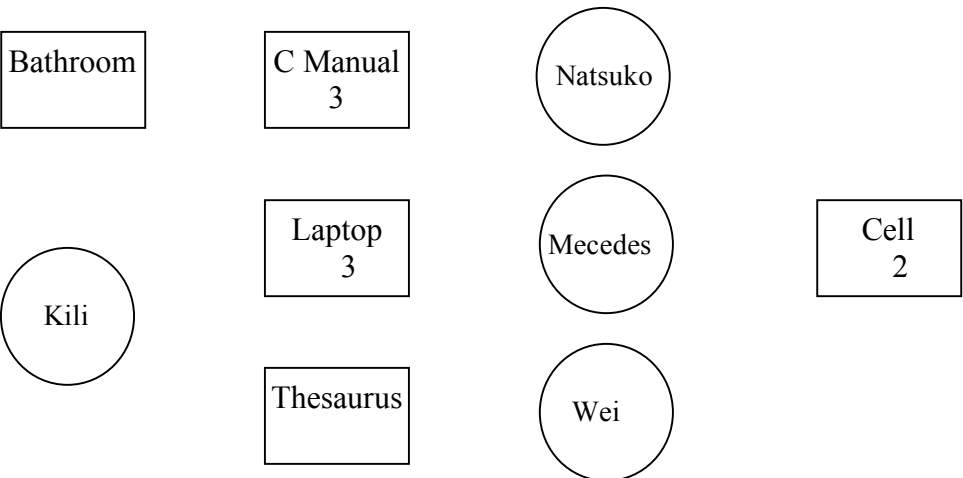
(Problem 4a continued)



Wei completed his work.



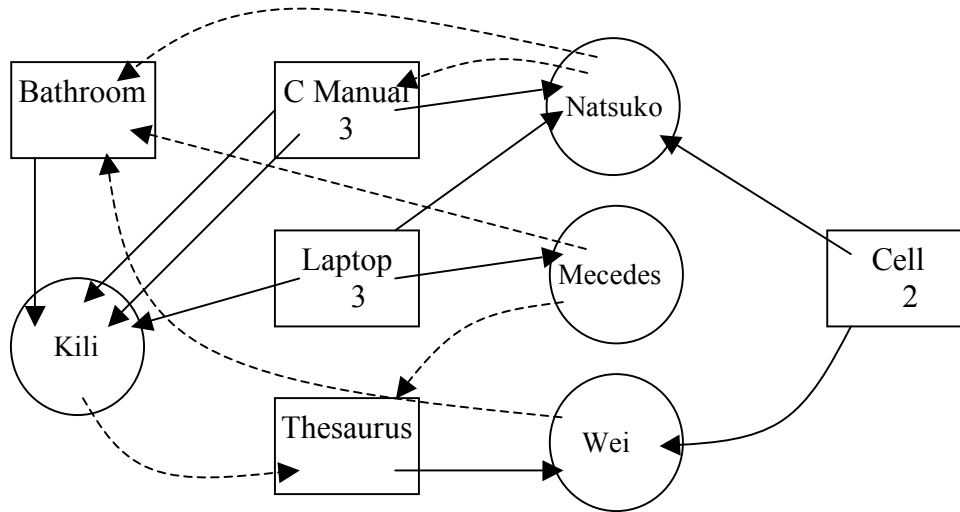
Kili completed his work.



Natsuko and Mercedes completed their work.

b) Before heading to the lab, our team stopped at one of the coffee stands on the way to EBU 3B, and each had a bowl of extra-spicy beef chili with fresh onions and a triple-shot pumpkin pie soy cappuccino. So, there's a high probability that each of them may need to bolt for the bathroom at a second's notice, carrying their work with them. Using a maximum claims graph, explain whether or not the above state is safe or unsafe.

The above state is unsafe: the maximum claims graph is not fully reducible.



Wei, currently holding the thesaurus, may want to use the bathroom. However, bathroom is occupied by Kili, who is waiting to use the thesaurus. This would be a deadlock state.