Suggested Solutions to COS318 Midterm, Fall 2010

1. Basic Concepts [10 pts]

Answer each question *well* in no more than three sentences unless specified otherwise.

a) [4 pts] Of the following items, circle those that are stored in the thread control block (TCB). For each item, give a few words (no more than one sentence) why it is or is not in the TCB:

i. General purpose registers. Yes. Part of the CPU context.

ii. Floating point registers. Yes. Part of the CPU context.

iii. Page table pointer. No. It is in PCB, not TCB.

iv. Stack pointer. Yes. Part of the CPU context.

v. Ready queue. No. Global data structure in the kernel.

vi. Program counter (instruction pointer). Yes. Part of the CPU context.

vii. Condition variables associated with this thread. No. They should be accessible by all threads in the same address space.

viii. Locks associated with this thread. No. They should be accessible by all threads in the same address space.

ix. File descriptors associated with this thread. No. Part of the PCB, not TCB.

b) [2 pts] What is the biggest disadvantage of implementing threads in user space? What is the biggest advantage?

The biggest advantage is that context switching between threads is faster because it does not require a trap into the kernel. The biggest disadvantage is that, if a user thread blocks in the kernel (e.g., waiting for I/O), no other threads in the process can run.

c) [2 pts] What advantages does a preemptive CPU scheduling algorithm have over a non-preemptive one?

- Better response time for interactive jobs
- Avoid starvation due to one job running forever
- More opportunities to achieve I/O parallelism by overlapping CPU and I/O

d) [2 pts] What is the difference between time-sharing and space-sharing a resource?

Time-sharing means assigning a process or user exclusive use of the resource for a short period of time; an example is CPU. Space-sharing means multiple processes or users can be using different parts of the resource simultaneously; examples are memory and disk.

2. Input and Output [10 pts]

Suppose a user process wishes to read some data from the disk. In this problem you will describe the steps that occur between when the process calls the read() library call, and when the data is returned to the process. The steps should cover the system call mechanism, the device driver interface, and the interaction between the driver and the hardware. Assume a monolithic operating system; synchronous, interrupt-driven I/O; and a DMA controller for the disk device. Also assume that the data is not available in the local disk cache. If you need to make additional assumptions, please state them.

Organize your description into three parts as follows. Describe each step in a few words, no more than one sentence per step.

a) [3 pts] Briefly list the steps between the process calling the read() library call and the execution of the system call handler.

- The application program calls the read() library
- The stub passes arguments according to system call convention, the system call number as an additional argument, and issues a trap instruction with the system call as the instruction operand.
- The system call code in the kernel saves the user context and switches to a kernel stack.
- The system call code in the kernel validates all arguments.
- The system call code looks up the system call number in the syscall vector, and dispatches the call to the corresponding system call handler in the kernel.

b) [3 pts] Briefly list the steps between executing the system call handler and calling the scheduler.

- The read() system call hander is invoked.
- The system call handler determines the “device address” of the data on the disk
- The system call handler calls read() on the disk device driver, passing the device address and user buffer address as arguments.
- The device driver allocates a kernel buffer for the read and initiates the DMA transfer from the disk to the kernel buffer.
- The process state is saved in the PCB and it is moved to “Blocked” state.
- The scheduler is called to run a new process.

c) [4 pts] Briefly list the steps between the DMA completion interrupt and the process computing on the newly read data.

- The DMA completion interrupt fires, invoking the disk interrupt handler.
- The interrupt handler notifies the device driver thread that the I/O has completed.
- The device driver thread copies the data from the kernel buffer to the user buffer and wakes up the blocked process (makes it ready).
- When the scheduler dispatches the process to run, it returns from the system call and can compute on the newly read data.

3. Mutual Exclusion and Deadlocks [10 pts]

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:

```
Status AtomicTransfer( Account a1, Account a2, float m ) {
        Acquire( a1->lock );
        if (( a1->balance – m ) < 0) {
                Release( a1->lock );
                Return NOT_ENOUGH_BALANCE;
        };
        Acquire( a2->lock );
        a1->balance -= m;
        a2->balance += m;
        Release( a2->lock );
        Release( a1->lock );
        return SUCCESS;
}
```

Note that AtomicTransfer must appear to occur atomically: there should be no interval of time during which an external thread (or person) can determine that money has been removed from an account but not transferred to another. In addition, the implementation should be highly concurrent—it must allow multiple transfers between unrelated accounts to happen in parallel. You may assume that a1 and a2 never refer to the same account.

a) [5 pts] Does this procedure transfer money atomically? Explain why or why not briefly. If it does not, show how to modify the code to fix it.

Yes, it does provide atomic transfer. The common mistake is to say that the original code does not provide atomic transfer. Since no other threads can get a lock on a1, no external thread can tell if any money has been deducted from a1 or not.

b) [5 pts] Does the procedure always work? Explain why or why not briefly. If it does not, show how to modify the code to fix it.

No, it does not always work. It can lead to deadlocks (one process transfers from a1 to a2 and another transfers from a2 to a1). One way is to use a global lock, but it is not highly concurrent. A better way is to enforce lock ordering:

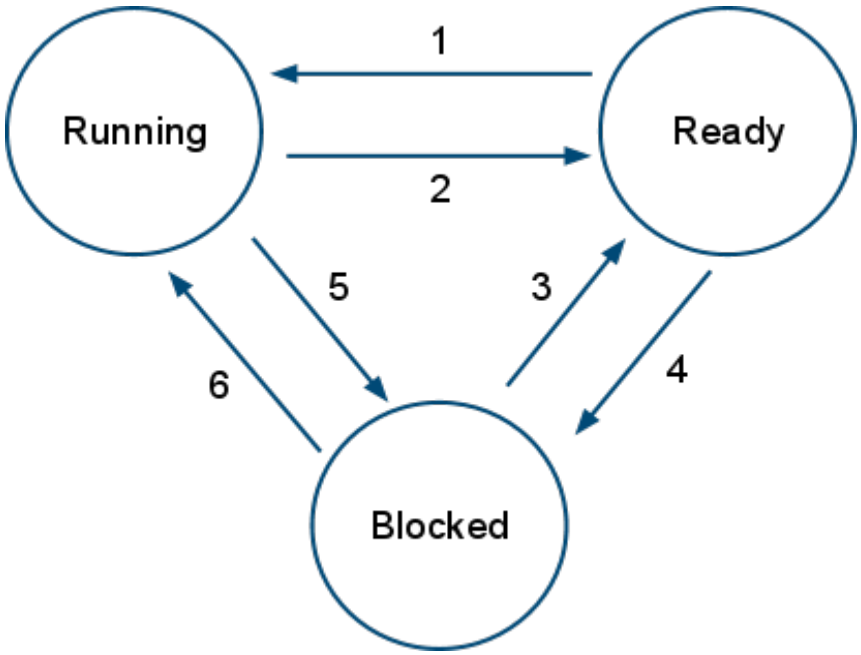
```
Status AtomicTransfer(Account a1, Account a2, float m) {
        if ( a1->id > a2->id ) {
                Acquire( a1->lock );
                Acquire( a2->lock );
        } else {
                Acquire( a2->lock );
                Acquire( a1->lock );
        };
        if (( a1->balance – m ) < 0) {
                Release( a2->lock );
```

```
            Release( a1->lock );
            Return NOT_ENOUGH_BALANCE;
        };
        a1->balance -= m;
        a2->balance += m;
        Release( a2->lock );
        Release( a1->lock );
        return SUCCESS;
}
```

Note that the lock releases can be in any order here.

4. Scheduling [10 pts]

a) [6 pts] Briefly explain what conditions cause a thread to move between each of the three states in a **preemptive** CPU scheduler, i.e., what causes each arrow. Label it N/A if it doesn't happen.



Arrow 1: CPU scheduler dispatches the thread

Arrow 2: Thread calls yield(), or a clock interrupt occurs and the thread is preempted

Arrow 3: Resource becomes available, or I/O completion interrupt fires

Arrow 4: N/A

Arrow 5: Thread blocks waiting for a resource to become free, or when performing I/O

Arrow 6: N/A

b) [2 pts] There are five jobs whose expected running times are 7, 3, 15, 8, and 4. In what order should they be run to minimize the average response time? Justify your answer in one or two sentences.

The order to achieve minimal average response time is: 3, 4, 7, 8, and 15. Shortest Time to Completion First (SCTF) is optimal for minimizing average response time.

c) [2 pts] Describe how a lottery scheduling algorithm could be made to approximate a CPU scheduling algorithm that always runs the job that hasn't run in the longest time. (No more than three sentences.)

Each time the scheduler dispatches a job, it takes away all the job's tickets and gives 1 ticket to all other jobs. The job that hasn't run in the longest time always has the most number of tickets.

5. Semaphores [10 pts]

You've just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure *hReady* when it's ready to react, and each O atom invokes a procedure *oReady* when it's ready. For this problem, you are to write the code for *hReady* and *oReady*. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the procedures must call the procedure *makeWater* (which just prints out a debug message that water was made). After the *makeWater* call, two instances of *hReady* and one instance of *oReady* should return.

Write the code for *hReady* and *oReady* using **semaphores** for synchronization. Your solution must avoid starvation and busy-waiting. *Hint: you may want to use three semaphores.*

Here is one solution:

For every triple of two hReady calls and an oReady call, exactly one call to makeWater should be made. In addition, a call to oReady or hReady should return only after successfully forming such a triple.

```
Semaphore hArrived(0);      // for oxygen threads to wait on
Semaphore hCanLeave(0);     // for hydrogen threads to wait on
Semaphore oMutex(1);        // to prevent multiple oxygen threads from
                            // fighting for hydrogen.

void hReady() {
        hArrived.V();       // announce arrival
        hCanLeave.P();      // wait until we're allowed to leave
}

void oReady() {
        oMutex.P();         // make sure only one oxygen claims arriving hydrogens
                            // to avoid deadlock.
        hArrived.P();       // wait for a hydrogen thread to arrive
```

```
        hArrived.P();           // wait for another hydrogen thread to arrive
        oMutex.V();             // allow another oxygen to start...

        makeWater();            // form water with those two atoms and ourself
                                // this assumes makeWater is thread-safe.
        hCanLeave.V();          // allow a hydrogen thread to leave
        hCanLeave.V();          // allow another hydrogen thread to leave

}
```

Because the oxygen thread is unique in any water-forming triple, it can most easily perform as the controlling thread for the interaction. Were we to have one of the hydrogen threads direct the interaction and call makeWater, we would have to implement a more elaborate *election* to assign the asymmetric roles. The mutex is required to make sure only one oxygen at a time controls the show; if more than one started counting hydrogens, we could have the situation where we have two hydrogens, multiple oxygens, and no water is made.