



# COS 318 : Midterm Exam

---

## 1. Basic Concept (10 points)

Answer the following questions (True or False). Use exactly one sentence to describe why you choose your answer. Without the reasoning, you will not get any points.

**1.1 (3 points) Of the following terms, circle the ones stored in the thread control block (TCB) and give a few words (no more than one sentence) why it should or should not be in the TCB.**

- a) General purpose registers. Yes. Part of the CPU context.
- b) Floating point registers. Yes. Part of the CPU context.
- c) Page table pointer. No. It is in PCB, but not TCB.
- d) Stack pointer. Yes. Part of the CPU context.
- e) Ready queue. No. Global data in the kernel.
- f) Program counter (instruction pointer). Yes. Part of the CPU context.
- g) Condition variables associated with this thread. No. They should be accessible by all threads in the same address space.
- h) Locks associated with this thread. No. They should be accessible by all threads in the same address space.
- i) File descriptors associated with this thread. No. Part of the PCB, not TCB.

**1.2 (2 points) Explain why an OS designer needs to worry about deadlocks involving physical memory but not CPU (in no more than two sentences).**

Physical memory is not a preempt-able resource, whereas CPU is a preempt-able resource. By preemptive scheduling, the OS can take CPU away from a process or thread.

**1.3 (2 points) Use one sentence to describe each of the steps of the system call mechanism for an operating system with a monolithic kernel.**

Step 1: A user application program calls the system call stub library call and using procedure call convention to pass arguments.

Step 2: 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.

Step 3: The system call mechanism in the kernel validates all arguments and dispatch the call to the corresponding system call routine in the kernel.

Step 4: The system call routine in the kernel executes the system call.

Step 5: The kernel now issues a special return instruction to return to the system call stub.

Step 6: The stub returns to the application caller.

**1.4 (3 points) Use one sentence to describe each of the steps of the system call mechanism for an operating system with a micro-kernel.**

Step 1: A user application program calls the system call stub library call and using procedure call convention to pass arguments.

Step 2: 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.

Step 3: The system call mechanism in the kernel checks to see if the system call is implemented in the micro-kernel or in an OS service process. If it is implemented in an OS service process, go to step 5. Otherwise, the system call mechanism dispatch to the internal system call routine.

Step 4: The system call routine in the micro-kernel validates all argument and executes the system call. Go to step 8.

Step 5. The micro-kernel sends a system call request message with arguments to the OS service process.

Step 6: The OS service process receives the system call message, validates all arguments, executes the system call, and returns the result back to the micro-kernel by sending a result message.

Step 7: The micro-kernel receives the result message call.

Step 8: The micro-kernel issues a special return instruction to return to the system call stub.

Step 9: The stub returns to the application caller.

## 2 CPU Scheduling (10 points)

**2.1 (3 points) 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? Can you explain why?**

The order to achieve minimal average response time is: 3, 4, 7, 8, and 15.

**2.2 (3 points) 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.**

To approximate a scheduling algorithm that always runs the job that hasn't run in the longest time, we can make a lottery scheduling algorithm do the following. At each time slice, the scheduler takes away all tickets from the job it runs next, and gives 1 ticket to all other jobs. In this case, the job that hasn't run in the longest time always has the most number of tickets.

**2.3 (4 points) Suppose there program A is a CPU job that runs forever and program B is a mixed CPU and I/O job that also runs forever (program B consumes 99 msec CPU and performs 10msec I/O, looping forever). Assume that the overhead to issue an I/O request is 0 and that the context switch overhead is 0. Answer the following questions:**

**(a) If the operating system does preemptive round-robin scheduling with an 1-second time slice, what are the CPU utilization and I/O utilization?**

The two processes run in the following way:

- A (1 sec CPU)
- B (99 msec CPU and issues an I/O request),
- Overlapping B (10 msec I/O) and A (1 sec CPU)
- B (99 msec CPU and issues an I/O request)
- Overlapping B (10 msec I/O) and A (1 sec CPU)
- ...

The CPU utilization is close to 100%, because I/O completely overlaps with CPU jobs.

The I/O utilization is close to 0.9%. At the steady state, there is 10 msec I/O every 1.099 sec.

**(b) If the operating system does preemptive round-robin scheduling with an 100msec time slice, what are the CPU utilization and I/O utilization?**

The two processes run in the following way:

- A (100 msec CPU)
- B (99 msec CPU and issues an I/O request),
- Overlapping B (10 msec I/O) and A (100 msec CPU)
- B (99 msec CPU and issues an I/O request)
- Overlapping B (10 msec I/O) and A (100 msec CPU)
- ...

The CPU utilization is close to 100%, because I/O completely overlaps with CPU jobs.

The I/O utilization is close to 5%. At the steady state, there is 10 msec I/O every 199 msec.

### 3 Mutual Exclusion and Deadlocks (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:

```

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;
}

```

Please answer the following questions:

- 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.
- Does this procedure always work? Explain why or why not briefly. If it does not, show how to modify the code to fix it.

**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): 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): 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. Another 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. Monitors (10 points)

You are asked to show how to implement an eventcount package with Mesa-style monitor at a job interview at a challenging startup company. An eventcount is an object that keeps a count of the number of events that have occurred in a system. There are four primitives of the eventcount abstraction:

- `Init( EC ec )`: Initialize the eventcount object.
- `Advance( EC ec )`: Increment the counter in the eventcount object by 1.
- `Read( EC ec )`: Return the value of the counter in the eventcount object.
- `Await( EC ec, int v )`: Block until the value of the counter in the eventcount object

reaches a specific value  $v$ .

**Note that `Advance`, `Read` and `Await` primitives should be atomic. User programs can use this set of primitives to implement synchronizations based on event occurrences conveniently without using other synchronization primitives.**

**Your job is to first define the data structure of the eventcount and then show how to use Mesa-style monitor to implement the primitives.**

The data structure definition for EC is:

```
struct EC {
    int counter;
    Mutex lock;    /* mutex */
    Cond c;        /* condition variable */
};
```

The new code is:

```
Init( EC ec ) {
    Mutex_Init( ec.lock );
    Acquire( ec.lock );
    ec.counter = 0;
    Cond_Init( ec.c );
    Release( ec.lock );
};

Read( EC ec ) {
    return ec.counter;    /* assume atomic memory load */
};

Advance( EC ec ) {
    Acquire( ec.lock );
    ec.counter++;
    Release( ec.lock );
    Broadcast( ec.c );
};

Await( EC ec, int v ) {
    Acquire( ec.lock );
    while ( ec.counter < v )
        Wait( ec.lock, ec.c );
    Release( ec.lock );
};
```