# Part I: Short Questions

1. Which of the following are more like policies, and which are more like mechanisms? **For each answer, circle policy or mechanism.**

   (a) The timer interrupt — Policy / Mechanism

   (b) How long a time quantum should be — Policy / Mechanism

   (c) Saving the register state of a process — Policy / Mechanism

   (d) Continuing to run the current process when a disk I/O interrupt occurs — Policy / Mechanism

2. For a workload consisting of ten CPU-bound jobs of all the same length (each would run for 10 seconds in a dedicated environment), which policy would result in the **lowest average response time**? **Please circle ONE answer.**

   (a) Round-robin with a 100 millisecond quantum

   (b) Shortest Job First

   (c) Shortest Time to Completion First

   (d) Round-robin with a 10 nanosecond quantum

3. Processes (or threads) can be in one of three states: **Running**, **Ready**, or **Blocked**. For each of the following four examples, write down which state the process (or thread) is in:

   (a) Waiting in `Domain_Read()` for a message from some other process to arrive.

   (b) Spin-waiting for a variable `x` to become non-zero.

   (c) Having just completed an I/O, waiting to get scheduled again on the CPU.

   (d) Waiting inside of `pthread_cond_wait()` for some other thread to signal it.

4. What is a **cooperative** approach to scheduling processes, and why is it potentially a bad idea?

5. Assume we run the following code snippet. After waiting for a "long" time, how many processes will be running on the machine, ignoring all other processes except those involved with this code snippet? You can assume that fork() never fails. Feel free to add a short explanation to your answer.

```
void
runMe()
{
    for (int i = 0; i < 100; i++) {
        int rc = fork();
        if (rc == 0) {
            while (1)
                ; // spin forever
        } else {
            while (1)
                ; // spin forever
        }
    }
}
```

   **Number of processes running:**

6. In class, we gave the following code as an implementation of mutual exclusion:

```
boolean lock[0] = lock[1] = false;
int turn = 0;
void deposit (int amount) {
    lock[pid] = true;
    turn = 1 - pid;
    while (lock[1-pid] && (turn == (1 - pid)))
        ; // spin
    balance = balance + amount;
    lock[pid] = false;
}
```

Let's say we replace the statement `turn = 1 - pid` with the statement `turn = BinaryRandom()`, where the function `BinaryRandom()` returns a 1 or 0 at random to whomever calls it. **Will the code still function properly? If so, why, and if not, what problem could occur?**

7. Assume the following code snippet, where we have two semaphores, 'mutex' and 'signal':

```
Thread 1                                    Thread 2

sem_wait(mutex);                            sem_wait(mutex);
if (x > 0)                                  x++;
    sem_post(signal);                       sem_post(signal);
sem_post(mutex);                            sem_post(mutex);
sem_wait(signal);
```

We want 'mutex' to provide mutual exclusion among the two threads, and for 'signal' to provide a way for thread 2 to activate thread 1 when 'x' is greater than 0. What should the initial values of each of the two semaphores be? (Assume that 'x' is always positive or zero, and that there are only these two threads in the system).

**Value of mutex:**

**Value of signal:**

8. Which of the following will **NOT** guarantee that deadlock is avoided? **Please circle all that apply.**

   (a) Acquire all resources (locks) all at once, atomically

   (b) Use locks sparingly

   (c) Acquire resources (locks) in a fixed order

   (d) Be willing to release a held lock if another lock you want is held, and then try the whole thing over again

9. A number of threads periodically call into the following routine, to make sure that a file that is shared between them has already been opened (after calling this routine, a thread might go ahead and call write() on that file, for example). Assume there is a global integer `fd`, which is set to -1 when the fd is closed, and a global lock `lock`, which is used for synchronization. Here is the code:

```
void MakeSureFileIsOpen() {
    mutex_lock(&lock);
    if (fd == -1)
        fd = open(''/tmp/file'', O_WRONLY);
    mutex_unlock(&lock);
}
```

   However, you get clever, and decide to re-write the code as follows:

```
void MakeSureFileIsOpen() {
    if (fd == -1) {
        mutex_lock(&lock);
        if (fd == -1)
            fd = open(''/tmp/file'', O_WRONLY);
        mutex_unlock(&lock);
    }
}
```

   **Does this code still work correctly? Why? If so, what advantage do we gain by using this implementation, and why is the condition** `(fd == -1)` **rechecked inside the mutex? If not, why doesn't it work?**

10. For a workload consisting of ten CPU-bound jobs of varying lengths (half run for 1 second, and the other half for ten seconds), which policy would result in the lowest total run time **for the entire workload?** Assume that context switch time is zero for this problem, and **please circle all that apply.**

    (a) Shortest Job First

    (b) Shortest-Time to Completion First

    (c) Round-robin with a 100 millisecond quantum

    (d) Multi-level Feedback Queue

11. A mechanism that can be used for synchronization is the ability to turn on and off interrupts.

    **a)** How can you use this to implement a critical section?

    **b)** Why does does it work on a single processor system?

    **b)** Why does't does it work on a multi-processor system?

    **c)** Why is this generally a bad idea, whether on a single or multi-processor system?

# Part II: Longer Questions

1. **Race to the Finish**

   Assume we are in an environment with many threads running. Take the following C code snippet:

```
int z = 0; // global variable, shared among threads
void update (int x, int y) {
    z = z + x + y;
}
```

   Assume that threads may all be calling update with different values for x and y.

   **a):** Write assembly code that implements the function update(). Assume you have three instructions at your disposal: (1) **load [address], Rdest**, (2) **add Rdest, Rsrc1, Rsrc2**, and (3) **store Rsrc, [address]**. Also, feel free to assume that when update() is called, the value of 'x' is already in R1, and the value of 'y' is in R2.

   **b):** Because this code is not guarded with a lock or other synchronization primitive, a "race condition" could occur. Describe what this means.

   **c):** Now, label places in the **assembly** code where a timer interrupt and switch to another thread could result in such a race condition occuring.

   **d):** Now, assume we change the C code as follows:

```
void update (int x, int y) {
    z = x + y; // note we just set z equal to x+y (not additive)
}
```

   If two threads call update() at "nearly" the same time, the first like this: 'update(3,4)', and second like this: 'update(10,20)', what are the possible outcomes? If we place a lock around the routine (e.g., before setting z = x + y, we acquire a lock, and after, we release it), does this change the behavior of this snippet?