Do NOT turn this page until you have received the signal to start.

(Please fill out the identification section above, write your name on the back of the test, and read the instructions below.)

Good Luck!
Q1. (1 mark each) Indicate below, for each statement, whether it is (T)rue or (F)alse. Circle the correct answer.

T / F: System calls provide an interface for user-level processes to request services from the operating system

T / F: Since threads are more lightweight, they could cooperate better if they had their own address space, just like processes.

T / F: During a context switch, when the return-from-trap instruction is executed, instead of returning to the process that was running before the context switch, we resume the execution of another process.

T / F: Concurrent operations on a shared resource where the outcome depends on the order in which accesses take place, is called a race condition.

T / F: Under Mesa semantics, when a process blocked on a condition is awoken, the condition is guaranteed to hold and does not need to be checked again.

Q2. (2 marks each)

a) In the top part of each process’s address space, there is a mapping of kernel addresses, containing the exception vector, some other OS code, etc. Explain briefly what are the advantages to having the kernel code mapped in each process’s address space, and some possible problems that OS designers need to account for.

Ans: Discussed in the lecture slides.

Advantages: Efficiency/performance reasons. Basically, we want to try to minimize the overhead of utilizing kernel services, since system calls and such are quite frequent. Mapping the kernel code and (some of the) data into each process addr space does just that.

Problem: The process could try to read or modify OS code. A designer has to ensure that the process cannot peek and poke into the kernel code and data, and make sure that those memory regions have different access rights/permissions (using page tables).

b) Name two cases when a process can be placed in the ready state.

Ans: See the process state transition diagram discussed in the lecture slides.
- When the process begins.
- When the timer interrupt comes in and the scheduler dispatches a different process into execution.
- When the process is in the blocked state waiting for I/O and an signal comes in indicating that the I/O burst is done.

c) Semaphores have certain limitations compared to condition variables. Argue the validity of this statement, and give a brief example.

Ans: Discussed in class – see lecture slides on Synchronization.

Can be hard to reason about synchronization -- The reason for waiting is embedded in the semaphore’s P() operation
E.g. “if count == 0, then sleep”

Sometimes you want a more complex wait condition
E.g. “if x == 0 and (y > 0 or z > 0), then sleep”

If it’s such a condition, then it must be checked outside the P()

But checking requires mutual exclusion => Easy to get stuck this way

So semaphores are too simple to handle more complex conditions than just “hitting negatives”.

d) In assignment A1, once we intercept the system call ‘SYS_write’, any process invoking this system call will go through the interceptor. Explain in what situations were you supposed to log a message, and when should you call the original (saved) system call?

Ans: Log a message if monitored = 2, or if monitored = 1, and the pid is in the list of monitored processes for SYS_write. The original system call should always be called, regardless if we log a message or not.
Q3. (8 marks) Consider the following problem: we have two functions that operate on a list called `listhead`, defined below. The function `populate_list` keeps adding nodes to the list, as long as the length of the list does not exceed a given capacity stored in the variable `capacity`. When that happens, it has to wait for `clear_list` to remove some elements from the list. The function `clear_list` keeps removing nodes from the front of the list, as long as the length of the list does not drop to 0. When that happens, it has to wait until more elements get inserted in the list.

Using mutexes and semaphores, make sure that these functions are correctly synchronized, to exhibit the behaviour requirements described above.

Consider that you have the following functions, with the following meaning:

- `CalculateListLength()` = calculates and returns the length of the list `listhead`
- `DeleteFirst()` = removes the first element from the list `listhead` and updates the `listhead` global variable
- `InsertValue()` = inserts a random value into the list `listhead` somewhere in the list

```c
typedef struct _node {
    int value;
    struct _node * next;
} node;

node *listhead;
int number = 42;
int capacity = 10;
int loops = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t not_empty, not_full;

void * populate_list() {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (CalculateListLength() == capacity) {
            pthread_mutex_unlock(&mutex);
            sem_wait(&not_full);
            pthread_mutex_lock(&mutex);
        }
        InsertValue();
        sem_post(&not_empty);
        pthread_mutex_unlock(&mutex);
    }
}

void * clear_list() {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (CalculateListLength() == 0) {
            pthread_mutex_unlock(&mutex);
            sem_wait(&not_empty);
            pthread_mutex_lock(&mutex);
        }
        DeleteFirst();
        sem_post(&not_full);
        pthread_mutex_unlock(&mutex);
    }
}
```
Q4.

a) (8 marks) Consider that:
4 processes (P0-P3) are being run
• Each process Pi starts at time 2 * i
• Each process does a 3-unit CPU burst, a 2-unit I/O burst, and then a 5-unit CPU burst
The scheduler is a 3-queue (Q0-Q2) priority scheduler (Q0 is the highest priority)
• Each queue uses Round-Robin with a quantum of 2
• New processes and processes returning from I/O start in Q0
• If a process is preempted, it moves from Qi to Qi+1

Indicate below, in each cell, what each process does from the point when it starts until it finishes. From the moment when a process Pi starts, each of the cells on Pi’s row should be filled with only one of the following labels:
- CPU: if the process is in the running state (if it has control of the CPU during that timeslot)
- IO: if the process is waiting for I/O
- Qi: if the process is in a ready state, waiting in Qi (where i between 0 and 2).

Be very careful when you fill this table, any mistake could cause your entire schedule to be off by one. Go over it carefully and analyze at each step in time, what each process should be doing.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>IO</td>
<td>IO</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>IO</td>
<td>IO</td>
<td>Q0</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>IO</td>
<td>IO</td>
<td>Q0</td>
<td>Q0</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
</tr>
<tr>
<td>P0</td>
<td>CPU</td>
<td>CPU</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>CPU</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>CPU</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>IO</td>
<td>IO</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>CPU</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) (2 marks) How do we choose the length of the time quantum for a round-robin scheduling policy in general? What can happen if we choose either a too large, or a too short time quantum? Explain briefly.

Ans: Again, straight from lecture slides:
Too large ($q \rightarrow \infty$) => process runs to completion first, this turns into First Come First Served basically.
Too small ($q \rightarrow 0$) => turns into CPU sharing.