This midterm consists of 4 questions on 7 pages (including this one). When you receive the signal to start, please make sure that your copy is complete.

Pseudo-code is acceptable where code is required. Answer the questions concisely and legibly. Answers that include both correct and incorrect or irrelevant statements will not receive full marks.

If you use any space for rough work, indicate clearly what you want marked.

Q1: _____/6  
Q2: _____/8  
Q3: _____/8  
Q4: _____/10  
Total: _____/32
Q1. (1 mark each) Indicate below, for each statement, whether it is (T)rue or (F)alse. Circle the correct answer.

T / F: User-level applications should always be able to access data on disk without the overhead of going through the OS.

T / F: Threads share the same address space, including global variables, stack and heap.

T / F: During a context switch, the OS has to save the registers for the currently executing process, and restore those for the process that will take over the CPU next.

T / F: Interrupts should be disabled whenever the processor is in kernel mode.

T / F: Priority inversion happens only in a non-preemptive system.

T / F: A performance problem with the Round Robin scheduler is that when a process makes an I/O request, the CPU sits idle until the quantum expires.

Q2. (2 marks each)

a) For a system call, if more than 6 parameters are needed, we can package the rest in a struct and pass a pointer to it as the 6th parameter of the system call. Explain briefly why do we have to validate such pointers and use safe functions like copy_from_user() and copy_to_user()?

Ans: Discussed in lecture slides. Could pass wrong pointers, e.g., a) possibly a NULL pointer that will be dereferenced and crash the kernel, or b) a process could provide a pointer that points to kernel memory, to trick the kernel into overwriting its own data, and possibly gaining access to restricted memory.

b) Explain when can a process go into the running state and when does it get placed into the blocked queue.

Ans: See process state transition diagram discussed in lecture slides.
- When the scheduler dispatches the process into execution on the CPU.
- When the process has to wait for I/O (or when it gets a SIGSTOP).

c) With Hoare monitor semantics, a broadcast operation on a condition variable is impossible. Argue whether this statement is valid and briefly explain why.

Ans: Discussed in lectures. With broadcast operations, we are not guaranteed that the condition holds once a process gets unblocked from the condition it waits on. If we signal one process, that process is guaranteed that its condition holds. If we broadcast, the lucky one that gets to run to completion through the monitor, might change data such that the condition no longer holds for the next process. So the next process would have to recheck the condition, like in Mesa semantics (while loop).

d) In assignment A1, why do we need to intercept exit_group? Explain briefly what type of situation can arise that would require us to be able to know when a process exits and act accordingly.

Ans: If a process exits without being de-monitored, then there's no way for us to know to remove it from any pid list where it might be monitored in. As a result, we have to intercept exit_group, in order to be able to properly remove the pid from the corresponding lists.
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 condition variables, 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

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;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER, not_full = PTHREAD_COND_INITIALIZER;

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

void * populate_list() {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while(CalculateListLength()==capacity) {
            pthread_cond_wait(&not_full,&mutex);
        }
        InsertValue();
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
    }
}
Q4.

a) (8 marks) Consider that:
4 processes (P0-P3) are being run
- Each process Pi starts at time i
- Each process does a 3-unit CPU burst, a 1-unit I/O burst, and then a 6-unit CPU burst
The scheduler is a 3-queue (Q0-Q2) priority scheduler (Q0 is the highest priority)
- Q0 uses a round-robin scheduler with a quantum of 2
- Q1 uses a round-robin scheduler with a quantum of 1
- Q2 uses a FCFS scheduler
- 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 IO
- 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>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>IO</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
</tr>
<tr>
<td>P1</td>
<td>Q0</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>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>
</tr>
<tr>
<td>P2</td>
<td>Q0</td>
<td>Q0</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>IO</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
</tr>
<tr>
<td>P3</td>
<td>Q0</td>
<td>Q0</td>
<td>CPU</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>IO</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>P1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>P2</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>Q2</td>
<td>Q2</td>
<td>Q2</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>Q1</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>P3</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>Q1</td>
<td>Q1</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>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
</tr>
</tbody>
</table>

b) (2 marks) Explain briefly how does the context switch time relate to the implementation of the round-robin scheduling heuristic in general.

*Ans: Again, discussed in lectures. It affects our decision on what a good time quantum is. Although we want the time quantum to be small, we want it to be large with respect to context switch time. Otherwise, we could end up spending more time in context switching than in processes doing actual useful work.*