1. (4 points) System calls:
   (a) What is a system call?
   (b) What is the relevance of system calls to the modes of operation of the CPU?
   (c) Your first program had calls to `lstat()` and to `readdir()`. What are their purposes?
   (d) I/O operations are restricted to the O.S. Both calls above result in disk I/O. But `readdir()` is not a system call. How do you explain this?

2. You are given the following current state of a system:
   
<table>
<thead>
<tr>
<th>Process</th>
<th>Allocated</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&lt;0,1,0,0&gt;</td>
<td>&lt;0,1,0,2&gt;</td>
</tr>
<tr>
<td>B</td>
<td>&lt;0,0,0,1&gt;</td>
<td>&lt;1,0,1,0&gt;</td>
</tr>
<tr>
<td>C</td>
<td>&lt;0,0,0,1&gt;</td>
<td>&lt;0,2,0,1&gt;</td>
</tr>
<tr>
<td>D</td>
<td>&lt;1,0,0,0&gt;</td>
<td>&lt;1,0,0,1&gt;</td>
</tr>
<tr>
<td>E</td>
<td>&lt;1,0,1,0&gt;</td>
<td>&lt;0,0,1,0&gt;</td>
</tr>
</tbody>
</table>

   Is the system in a deadlock state? If so, indicate which of the processes are involved in the deadlock. If not, give a process completion sequence.

3. You compile and run the following program on a Linux machine. List one possible output from the program (assuming that the output from a `printf` is never interrupted by the output from another `printf`). How many possible output orderings are there?

   ```c
   #include <stdio.h>
   int main()
   {
     int X = 100;
     int pid1, pid2;

     printf("Original X = %d\n", X);

     pid1 = fork();
     if ( pid1 == 0 )
     {
       X = 200;
     }
     else
     {
       X = 400;
     }
     printf("Now X is %d\n", X);

     pid2 = fork();
     printf("Finally %d has a child\n", X);
   }
   ```

4. You have four tasks T1, T2, T3 and T4 to be completed by processes P1, P2, P3 and P4 respectively. All processes begin execution concurrently. However, P4 should execute T4 only after at least two of the other processes complete their tasks. How can this synchronization be achieved using semaphores? List all four processes using the following
format:
/* Declare and initialize semaphores */
Process P1:
    /* Initializations as needed*/
    /* Semaphore actions as appropriate */
    T1();
    /* Semaphore actions as appropriate */

4. Assume that you have the following code:

```c
double rainfall[10000];
int main()
    int i;
    for(i = 0; i < 10000; i++)
        rainfall[i] = 0;
}
```

Your operating system uses pure demand paging, with a page size of 500 bytes. The size of a double is 8 bytes. All of the code fits on one page (page 0). The rainfall array is allocated on a heap, which begins on page 1. The process is allocated 2 frames. The page replacement algorithm is LRU. How many page faults does the process incur?

5. Translate the virtual addresses given below based on the following page table. If any of the virtual addresses is “invalid” because the page that contains the address is not loaded into a frame, indicate so. The size of each page is 1000 bytes.

<table>
<thead>
<tr>
<th>Page #</th>
<th>Reference bit</th>
<th>Valid bit</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

   a) 1051
   b) 3333
   c) 2882
   d) 4215

6. If the page replacement algorithm is LRU, how many page faults would the following reference string incur: 1, 2, 1, 1, 3, 4, 1, 3, 1, 1, 5, 1, 3, 6, 5, 2, 1, 5, 7, 6, 1? How many page faults would the optimal algorithm encounter? Assume that in both cases you have 3 frames available, none of which contains any pages to begin with.

7. We know that the main advantage of symlinks is that a symlink on one disk could “target” a file on a different disk. There are, however, some situations where symlinks are advantageous even if the link and its target are on the same disk. Name one such situation. (Try to think of a situation where a hardlink would fail to work properly after some operations.)

8. Assume that a two processor system has an operating system with a scheduler that ensures that both processes execute the same number of processes. There is a list of $n$ processes to be scheduled on these processors. How many schedules are possible? You may assume that $n$ is even. (Hint: First, the scheduler needs to select half the processes for execution on processor #1. In how many ways can this be done? You need to recall a little bit about combinations
here. Second, in how many ways can the \( n/2 \) processes be ordered for each processor? Now put everything together.)

9. Consider the Linux operating system with the old \( O(n) \) scheduling algorithm. The scheduling algorithm gives the CPU to the process in the run queue with the most number of credits. When a process forks to create a new process, the child process should be given some number of initial credits. Given the following alternatives, rate each as “reasonable” or “unreasonable”. In each case explain your rating:
   a) The child process is given as many credits as owned by the parent.
      Answer: Unreasonable, because this is unfair to other processes, which now have to wait longer to get the CPU.
   b) The child process is given some fixed (non-zero) number of credits.
      Answer: Unreasonable, pretty much for the same reason as above, but a small fixed number of credits is not too unreasonable.
   c) Reduce the parent's credits by half, and give that many credits to the child.
      Answer: Reasonable—other processes are not unduly affected by the new process. Also, the child process will have a chance to go soon enough.
   d) Give the child process zero credits.
      Answer: Although fair to other processes, it greatly delays the time it takes for the new process to get to the CPU, and is hence unreasonable.

10. At some point in time three multiprogramming systems were observed with the following performances:
   a) CPU utilization: 3%         Disk utilization: 98%
   b) CPU utilization: 98%       Disk utilization: 3%
   c) CPU utilization: 3%         Disk utilization: 3%
In which (one or more) of the three systems would you add more processes (increase the degree of multiprogramming) to the system? Why wouldn't you add more processes to the rest?