Name: _____

UB ID Number: | | | | | | | | |

Question:	1	2	3	4	5	6	7	Total
Points:	5	5	5	5	5	5	20	40
Score:								

CSE421 Practice Midterm

This practice midterm is intended to familiarize you with the kind of problems that will appear on the actual midterm exam. The only significant difference is that this exam does *not* contain any multiple choice questions. The real midterm will contain 10 multiple choice questions worth 1 point each. These are drawn entirely from the lecture slides and intended to be very easy.

The point value assigned to each question is intended to suggest how to allocate your time. So you should work on a 5 point question for roughly 5 minutes. Each of the short answer questions are worth 5 points. The long answer question is worth 20 points.

We will work through the answers to this exam together in class. If you have time beforehand, you may want to sit down and devote 40 minutes to working through answers to the following problems.

Please fill out your name and UB ID number above. Also write your UB ID number at the bottom of each page of the exam in case the pages become separated.

I have neither given nor received help on this exam.

Sign and Date: _____

Short Answer

Choose **4 of the following 6** questions to answer. You may choose to answer additional questions, in which case you will receive credit for your best four answers.

1. (5 points) Explain the tradeoffs inherent to the scheduling quantum length. What happens when it gets very short? What happens when it gets very long?

Segment Base	Segment Bounds	Physical Offset	Permissions
1,000	450	500	Read
130,000	11,000	20,000	Read, Write
83,000	2,000	1,000	Read, Write

2. (5 points) Using the segmentation base and bounds table for the currently running process above, describe what would happen if each of the following pseudo-instructions were executed.

   Note: to make things easier on everyone the question uses base-10 arithmetic.

   1. load `1,200`
   2. store `140,000`
   3. load `36,788`
   4. load `84,100`
   5. store `1,000`

3. (5 points) One disadvantage of multi-level page tables is that the page table data structures themselves are fairly large. Assuming a two-level page table using 4K pages and 32-bit addresses, the top-level and second-level page tables are each themselves 4K: 4 bytes per entry * 2^{10} entries per table. If a process allocates many pages spread over its virtual address space, the page tables can become large.

   A potential solution to this problem is to *swap the page tables*, or move the top- and second-level page tables themselves to the swap disk. This approach can reduce the amount of memory in use. However, what difficulties can it cause?

4. (5 points) Explain the process of swapping out a page.

5. (5 points) 32-bit wide virtual addresses allow a process to address at most 4GB of memory. On certain architectures, such as the MIPS, a portion of that is reserved for use by the kernel, further reducing the potential size of the process virtual address space to 3GB or 2GB.

   Today, however, many machines ship with 8GB, 16GB or more memory. Explain how you might support larger amounts of memory and the implications of any ideas you propose. You may also argue why the existing address spaces might not be such a serious limitation.

6. (5 points) Explain how multi-level feedback queues (MLFQ) can starve processes. Propose one solution that addresses this problem.

Long Answer

Choose 1 of the following 2 questions to answer. **Please do not answer both questions.** If you do, we will only read one.

If you need additional space, continue and clearly label your answer on the back of this or other exam sheets.

7. (20 points) Choose one of the following two questions to answer:

   1. **Optimizing `fork()`.**

      `fork()` is an expensive system call. A significant component of that expense is copying the address space of the parent process, which is frequently rendered unnecessary by the child immediately calling `exec()` and destroying the copied address space.

      Propose an optimization to `fork()` motivated by this observation. Your solution should improve the performance of `fork()` followed by `exec()` but still provide private memory to the child in the case that `fork()` is *not* followed by `exec()`. You might want to use one of our system design principles as inspiration: if you wait to do something, you might never have to do it!

---

   2. **Priority inversion.**

      Priorities are intended to allow the user or system to express the importance of threads to the scheduler. Some schedulers use priorities as hints, but some use them strictly: a runnable high-priority thread will *always* preempt a runnable lower-priority thread.

      This can create problems when combined with synchronization primitives. Consider the following scenario. A system has three threads: one high-priority thread, a second medium-priority thread, and a third low-priority thread. In addition, the system has a single shared resource protected by a lock. Any thread can lock and unlock this shared resource, and the resource is non-preemptible, meaning a thread that tries to acquire the resource while it is in use will have to wait for the thread that is using the resource to finish and release the lock.

      First, identify one situation that would cause the high-priority thread to become blocked *indefinitely* waiting on the low-priority thread. We call this state *priority inversion* because it violates the goal of our strict priority scheduler. Second, propose a scheduling solution to this problem.

Scratch. **Please indicate what question you are answering.**

Scratch. **Please indicate what question you are answering.**