CS-537: Midterm Exam (Spring 2009)
I'm in Seattle; You're Taking an Exam

Please Read All Questions Carefully!

There are seven (7) total numbered pages.

Please put your NAME and student ID on THIS page, and JUST YOUR student ID (but NOT YOUR NAME) on every other page. Why are you doing this? So I can grade the exam anonymously. So, particularly important if you think I have something against you! But of course, I don't. Probably.

Name and Student ID: ____

Grading Page

	Points	Total Possible
Q1		20
Q2		20
Q3		20
Q4		20
Q5		20
Total		100

1. **Simple Scheduling, with Overheads.**

   Assume that jobs are submitted to a system in batches. Assume further that *sorting* on this machine is an expensive computation, and requires some amount of time per element sorted; you can estimate the time to sort $NumJobs$ elements with a function $Sort(NumJobs)$. Further assume that switching between jobs takes a fixed amount of time (say, on a context switch, or when one job ends and another begins); we will call this value $Switch$. In this question, you will answer some questions about scheduling algorithms given these overheads.

   First, assume three jobs arrive, of lengths 30, 20, and 10. Assume that $Sort(NumJobs) = NumJobs \times 10$, and that $Switch = 0$.

   (a) How long will the system take to compute the schedule for an SJF (Shortest Job First) Policy?

   (b) How long will it take to run the jobs once the schedule has been computed?

   (c) Including scheduling and switching overhead, what is the **average response time** for each job?

   (d) Including scheduling and switching overhead, what is the **average turnaround time** for each job?

   Now assume that switching is expensive, too; in fact, $Switch = 10$.

   (e) Including scheduling and switching overhead, what is the **average response time** for each job?

   (f) Including scheduling and switching overhead, what is the **average turnaround time** for each job?

   Now assume that these jobs are complete, and a different set of jobs arrive, of lengths 10, 10, 10, 10, and 10 (five jobs each of length 10). Again assume that $Sort(NumJobs) = NumJobs \times 10$, and that $Switch = 0$ again.

   (g) Assuming SJF and including all overheads, what is the **average response time** for each job?

   (h) Assuming SJF and including all overheads, what is the **average turnaround time** for each job?

   (i) What would a smarter policy be for this system? (How would it perform?)

2. **Code, Segments. Code Segments?**

   This simple question is about segmentation. Segmentation is an approach that uses some number of base and bounds registers to help virtualize memory.

   (a) What is a base register for?

   (b) What type of address is in the base register: physical, or virtual?

   (c) What is a bounds register for?

   (d) Why do we need more than one base/bounds register pair?

   (e) How many segments should the hardware support? (Why?)

   Now, assume we have a system with the following setup. There are two segments supported by the hardware. Address spaces are small (1KB), and the amount of physical memory on the system is 16KB. Assume that the segment-0 base register has the value 1KB, and its bounds (size) is set to 300 bytes; this segment grows upward. Assume the segment 1-base register has the value 5KB in it, and its bound is also 300; this segment grows downward (the negative direction).

   Assume we have the following program:

```
void *ptr = 20;
while (ptr <= 1024) {
    int x = (int *) *ptr; // LINE 1: read what is at address 'ptr'
    ptr = ptr + 20;       // LINE 2: increment 'ptr' to a new address
}
```

   (f) What virtual addresses are generated by this program at `LINE 1` by dereferencing `ptr` ? (assume the program runs to completion; please just list the addresses as generated by the loads from memory via the dereferencing of `ptr`; don't worry about instruction fetches or the store into `x`, for example)

   (g) How long will this program run before crashing (due to a segmentation violation)?

   (h) What physical addresses will be generated by dereferencing `ptr` before the program crashes?

   (i) What legal physical addresses could have been generated by dereferencing `ptr`, if the programmer had been more careful to avoid crashing?

3. **Paging and Page Tables.**

Assume the following: a 32-bit virtual address space, with a 1KB page size.

(a) How many bits are in the *offset* portion of the virtual address?

(b) How many bits are in the *VPN* portion of the virtual address?

Now, let's focus on the page table. Assume each *page table entry* is 4 bytes in size. Assuming a *linear* page table:

(c) How many entries are in the table?

(d) What is the total size of the table?

(e) In a live system, how much memory would be occupied by page tables? (what factors affect this?)

Linear page tables are too big. Hence, people came upon the idea of a *multi-level page table*, which uses a *page directory* to point to page-sized chunks of the page table. Assume we wish to build such a table, with two levels (as discussed in class). To access such a table, the VPN is split into two components: the VPN$_{PageDir}$ which indexes into the page directory, and the VPN$_{ChunkIndex}$ which indexes into the page of the page table where the PTEs are located.

(f) How many PTEs fit onto a single page in this system?

(g) How many bits are thus needed in the VPN$_{ChunkIndex}$?

(h) How many bits are needed in the VPN$_{PageDir}$?

(i) How much memory is needed for the page directory?

Finally, given the following memory allocations, write down both (a) how much memory our multi-level page table consumes and (b) how much memory a linear page table consumes:

(j) Code is located at address 0 and there are 100 4-byte instructions. The heap starts at page 1 and uses 3 total pages. The stack starts at the other end of the address space, grows backward, and uses 3 total pages.
- Multi-level page table size?
- Linear page table size?

(k) Code is located at address 0 and there are 100 4-byte instructions. The heap starts at page 1 and uses 1000 total pages. The stack starts at the other end, grows backwards, and uses 1000 total pages.
- Multi-level page table size?
- Linear page table size?

(l) The entire address space (every page) is used by the process.
- Multi-level page table size?
- Linear page table size?

4. **Tracing Virtual Memory.**

This question asks you to consider everything that happens in a system on a memory reference. Assume the following: 32-bit virtual addresses, 4KB page size, a 32-entry TLB, linear page tables (if it matters), and LRU replacement policies whenever such a policy might be needed by either hardware or software. Assume further that there are only 1024 pages of physical memory available. In this question, you will be running some *test code* and saying what happens when that code is run.

Before the test code is run, though, the following initialization code is run once (before testing begins). This code simply allocates (`NUM_PAGES*PAGE_SIZE`) number of bytes and then sets the first integer on each page to 0, where `PAGE_SIZE` is 4KB (as above) and `NUM_PAGES` is a constant (defined below). Assume `malloc()` returns page-aligned data in this example.

```
void *orig = malloc(NUM_PAGES * PAGE_SIZE); // allocate NUM_PAGES*PAGE_SIZE bytes
void *ptr = orig;
for (i = 0; i < NUM_PAGES; i++) {
    (int *) *ptr = 0; // init first value on each page
    ptr += PAGE_SIZE;
}
```

The code we are now interested in running, which we will call *the test code*, is the following:

```
ptr = orig;
for (i = 0; i < NUM_PAGES; i++) {
    int x = (int *) *ptr; // load value pointed to by ptr
    ptr += PAGE_SIZE;
}
```

For these questions, assume we are only interested in memory references to the malloc'd region through `ptr` (that is, ignore stores to `x` and instruction fetches). *How many TLB hits, TLB misses, and page faults occur during the test code when ...*

	TLB hits	TLB misses	Page Faults
(a) `NUM_PAGES` is 16?			
(b) `NUM_PAGES` is 32?			
(c) `NUM_PAGES` is 2048?			

Assume a memory reference takes roughly time M and that a disk access takes time D. *How long does the test code take to run (approximately), in terms of M and D, when...*

	TLB hits	TLB misses	Page Faults
(d) `NUM_PAGES` is 16?			
(e) `NUM_PAGES` is 32?			
(f) `NUM_PAGES` is 2048?			

Now assume we change the various replacement policies in the system to **MRU**. Given this change, *How long does the test code take to run (approximately), in terms of M and D, when...*

	TLB hits	TLB misses	Page Faults
(g) `NUM_PAGES` is 16?			
(h) `NUM_PAGES` is 32?			
(i) `NUM_PAGES` is 2048?			

Finally, assume you are to run this code on a new machine that you know very little about. In fact, you wish to use the test code to *learn* how big the TLB is and how much memory is on the given system.

(j) How could you use the test code above to learn these facts about the physical hardware?

5. **Virtual Machine Monitors.**

   This question is about virtual machine monitors.

   (a) Why do we need virtual machine monitors? (Doesn't the OS already virtualize the hardware?)

   (b) How do the abstractions presented by a virtual machine to the OS *differ* from the abstraction presented by the OS to an application?

   (c) How are the abstractions similar?

   (d) Why are TLB misses so slow when the OS is running on a VMM? (describe)

   (e) The OS has a *page table* to track virtual-to-physical translations. Does the VMM need a similar structure per operating system? (if so, why? if not, why not?)

   (f) The OS performs a *context switch* to switch between processes. Does the VMM perform a similar operation? (if so, why? if not, why not?)

   (g) VMMs sometimes have a hard time managing resources because they have so little information about what the OS is doing. Give an example where the decision making of the VMM is made more difficult because of its lack of insight into what the OS is doing.