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		25
Q2		25
Q3		25
Q4		25
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?

   Three jobs means 3×10 **or** 30

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

   Total run time: $10 + 20 + 30$ **or** 60

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

   First job starts at 30 (end at 40); second starts at 40 (ends at 60); third starts at 60 (ends at 90). Thus, $\frac{30+40+60}{3}$ **or** $43.33...$**.**

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

   From above, jobs end at 40, 60, 90; hence $\frac{190}{3}$ **or** $63.33...$**.**

   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?

   Some people assumed the first switch to the job (from the OS) cost you, others didn't; both were OK by me. This answer assumes it does cost you. Job 1 starts at 40 (end at 50); Job 2 starts at 60 (ends at 80); Job 3 starts at 90 (ends at 120). Hence $\frac{40+60+90}{3}$ **or** $63.33...$**.**

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

   As above: $\frac{50+80+120}{3}$ **or** $83.33...$**.**

   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?

   Sorting time: 50**. Thus, response times are** $50, 60, 70, 80, 90$**. Average is thus** 70**.**

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

   Completion times: $60, 70, 80, 90, 100$**. Average is thus:** 80**.**

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

   Depends on the metric (for sure), but the key realization here is any policy that avoids the (useless) sort is going to make things much faster. So, I was happy with FIFO, RR, etc.

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?

   It personally hurt my feelings when people would write "the base register is for the base address." You can't define a term with the exact same term! The base register holds the physical adddress of the location where a segment was placed in memory and thus allow for relocation of the segment in physical memory.

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

   As above: physical.

   (c) What is a bounds register for?

   The bounds tells the system the size of the segment (or ending address). It is used for protection, to prevent accesses from escaping the segment and going into another's address space.

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

   One pair only allows you to relocate the entire address space, which uses a great deal of physical memory (in particular for large and sparse address spaces). More than one pair allows for much better support of sparse address spaces.

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

   At least two (say one for code+heap, the other for the stack) gives the system the ability to support sparse address spaces. Three could also be useful, as adding a separate one for code allows code to be shared. More than that could be useful but perhaps beyond the scope of the exam.

   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)

   Starts at virtual address 20; increments by 20 until 1020 (assuming no crash, etc.)

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

   Access to virtual addresses 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280 all are fine (14 iterations). Access to 300 exceeds the bounds of segment 0 and thus will cause an exception (we know that the virtual addresses are in the 0th segment because the top bit of these addresses are 0; virtual addresses between 512-1023 have the top bit set to 1 and thus are in the 1st segment).

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

   The above virtual addresses must be added to the seg 0 base register (value 1024) to get physical addresses. Thus, 1044, ..., 1324 will be generated before crashing.

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

   If the program skipped the loop iterations that would cause a crash, it would skip the middle part of the address space which is not mapped and has no legal addresses. The last 300 bytes of the AS

are also legal (below 1024 starting at 724), and hence those could have been accessed legally. For example, if 1020 was accessed, it would be legal and translate to 5KB minus 4 or 5116 (remember the stack grows the other way). The first legal address in this range would be 740 which translates to 4836. Thus, every virtual address between 4836 and 5116 (inclusive) could have been legally accessed, if the loop skipped over 300 to 720 (which would all fault).

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?
10 bits (for a 1KB page, you need 10 bits to address each byte).

(b) How many bits are in the *VPN* portion of the virtual address?
That leaves 22 bits for the VPN (32 bits total - 10 bits of offset).

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?
2^{22} **entries (one for each virtual page).**

(d) What is the total size of the table?
$2^{22} \times 4$ **because each entry is 4 bytes in size (as stated in the question).** 2^{20} **is 1MB; hence** 2^{22} **is 4MB; hence the total answer is 16MB for the entire page table.**

(e) In a live system, how much memory would be occupied by page tables? (what factors affect this?)
16MB times the number of processes. If there are 100 processes, for example, this implies 1600MB of page table space is needed!

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?
We know a PTE is 4 bytes in size. Hence, 256 entries fit into one 1KB page.

(g) How many bits are thus needed in the VPN$_{ChunkIndex}$?
We need 8 bits to tell us which entry we are referring to ($2^8 = 256$**).**

(h) How many bits are needed in the VPN$_{PageDir}$?
Given that the VPN is 22 bits, and subtracting 8 for the ChunkIndex, that leaves us 14 bits for the PageDir.

(i) How much memory is needed for the page directory?
Depends on how big each page directory entry (PDE) is. Assuming 4 bytes, we get 2^{14} **entries times 4 bytes/entry or 64KB.**

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?
Each chunk of the page table covers 256 consecutive pages; hence the code pages and heap in this example fit onto the same chunk of the page table. The stack, at the other end of the address space, needs one too. Hence 2 pages (1KB each) plus the page directory (64KB), or 66KB.
- Linear page table size?
The linear page table is always the same regardless of the usage of pages in the address space. Hence, 16 MB.

(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?
  Here we need four pages of the page table to translate the first 1001 pages of the address space (four pages of the page table covers 1024 pages) and another four pages to translate the last 1000 pages of the stack. Hence, 8 pages (1KB each) plus the page directory (64KB) gets us to 72KB.
- Linear page table size?
  16 MB.

(l) The entire address space (every page) is used by the process.

- Multi-level page table size?
  If all pages are used, you get the full linear page table (16 MB) plus the page directory (64KB).
- Linear page table size?
  16 MB.

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?	16	0	0
(b) `NUM_PAGES` is 32?	32	0	0
(c) `NUM_PAGES` is 2048?	0	2048	2048

The workload just loops over some pages accessing each one once. Thus, the second time through the loop with a small number of pages (less than the TLB size, for example) just yields TLB hits. With a large number of pages, and LRU replacement, nothing we want is ever in the cache or TLB, hence all misses/page faults.

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?	16M		
(e) `NUM_PAGES` is 32?	32M		
(f) `NUM_PAGES` is 2048?		$2048 \times 2M$	2048D

A hit means a load to memory which we assume costs M and hence the TLB hits just cost M. On TLB misses we need to consult memory twice, once for the page table (minimally) and once for the actual memory load, hence 2M per TLB miss. Finally, each page fault costs us a disk access.

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?	16M		
(h) `NUM_PAGES` is 32?	32M		
(i) `NUM_PAGES` is 2048?	32M	$2016 \times 2M$	1024D

With MRU, the first 31 accesses get lodged in the TLB, and the first 1023 pages get lodged in memory; subsequent accesses simply push the most-recently used TLB entry/page out. The next time through the loop, the first 31 accesses are thus hits, and the first 1023 page accesses are in memory; everything else are misses until the last page, which also is a hit (do a small MRU example if this doesn't make sense). Thus, 32 TLB hits (the rest misses); 1024 page faults (the rest in memory).

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?

Just looking for something simple here. You could basically time how long an iteration of the loop lasts. If it lasts something like NUM_PAGES times memory access time, those are TLB hits; if it lasts twice that, TLB misses, and thus you can know how big the TLB is. One more big jump occurs when NUM_PAGES is finally bigger than memory and causes lots of (very slow) disk accesses.

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?)

   Lots of reasons here. Consolidation, need to run different OSes, security, etc.

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

   The VMM must present a hardware interface to the OS (to trick it into thinking it actually is running on the hardware. The OS, on the other hand, presents nicer abstractions of the machine (like files and directories instead of a raw disk, for example).

   (c) How are the abstractions similar?

   They are both virtualizations of the hardware; they both allow sharing while providing some isolation. Other answers are possible.

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

   On a software-managed TLB, the TLB miss traps to the VMM which must then jump to the OS code to handle the fault; when that code runs and tries to install a new entry into the TLB, another trap to the VMM; finally, when the OS tries to return to the user, another trap to the VMM, which finally returns to user mode and continues the process. A lot of extra work!

   (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?)

   Sure. It needs to track, per OS, where its "physical" page map to (to which machine pages they map).

   (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?)

   Sure, it does. It has more work to do, though, as it has to save/restore any privileged registers too.

   (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.

   Lots of possible answers here. One good example: OS in the idle loop looks no different than an OS running some useful code, and thus might be fairly scheduled by the VMM, wasting precious CPU cycles.