CS-537: Midterm Exam (Fall 2009)

The "Lost" OS: Building on the Popularity of Television

Please Read All Questions Carefully!

There are eleven (11) 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... (insert sinister music here)

Name and Student ID: ____

Grading Page

	Points	Total Possible
Q1		20
Q2		20
Q3		20
Q4		20
Q5		20
Q6		20
Total		120

News flash: An amazing operating system's source code has been lost! Let's call this operating system "LostOS". Your job, alas, is to help recover some aspects of what LostOS was doing.

Fortunately, you still have the LostOS binary, and can boot the kernel and even runs some jobs on it. Thus, you can sometimes empirically see how LostOS behaves. Further, when it is running, you can examine the state of memory to help figure out what LostOS is doing. You even have access to some old versions of LostOS, so you can learn how it has changed over time. You will even (occasionally) get some tips from the Old Gray Person (OGP), an old-time hacker who remembers bits and pieces about what LostOS does. Overall, you have a lot of information at your disposal in the relentless pursuit of recovering the hidden secrets of LostOS. Lucky you!

So, are you up for the challenge?[1]

- Yes: ____
- No: ____
- Unsure: ____

  Answer: Any of the above were accepted for full credit (roughly 0 points).

[1] Note that whatever your answer is, it still leads to the same result: you taking this exam.

1. **Lost Scheduling.**

According to the Old Gray Person (OGP), an old version of LostOS used a **Round Robin** scheduling policy in order to decide which jobs to run, with a 100-millisecond timeslice.

(a) What are the major strengths of the Round Robin approach?

Most important: improves response time of jobs, as they get scheduled immediately. Also, very simple, and fair across jobs (no starvation for example).

(b) What are its major weaknesses?

Most important: really bad for turnaround time, as it strings each job out as long as possible. Possibly context-switch overhead too.

The creators of LostOS eventually decided to modify this initial Round Robin, and in later versions changed the timeslice to 1 millisecond (instead of 100).

(c) In what way did this change make the LostOS scheduler better? (if any)

Improves response time even further!

(d) In what way did this change make the LostOS scheduler worse? (if any)

But, context-switch cost could be a problem. In general, the smaller the time slice, the more this overhead will matter.

Finally, according to OGP, the original creators got sick of Round Robin and decided to replace the decision of which process runs next with a Random approach (i.e., one that picks the next process to run at random, from the set of ready processes).

(e) How does the Random approach compare to traditional Round Robin? Is it better, worse, or quite similar?

I meant that at each quantum, a random job is picked (but some people took it to mean that the jobs were run in their entirety, which I accepted). In the way I meant it, RR(deterministic) and RR(random) actually behave pretty similarly. You can figure out the rest.

2. **Lost Base Values.**

An old version of LostOS, so says Old Gray Person (OGP), used segmentation (not paging) to provide some form of primitive virtual memory. In this problem, we'll use that knowledge to discover some lost base and bounds values.

OGP remember a few things. First, the system only had 2 segments (segment 0 for code and a growing heap, segment 1 for a negatively-growing stack). OGP also recalls on this old version of LostOS, the virtual address space size was only 128 bytes, and there was only 1K of physical memory.

OGP has one set of traces from one old program. In particular, the traces tell you which virtual address (VA) was accessed (in both hex and decimal forms), and then whether or not the access was valid or not (and hence a segmentation violation). If valid, the physical address (in both hex and decimal) are reported. Oddly enough, programs in LostOS were allowed to keep running after memory-access violations, and thus we have a long trace that continues even after such a violation occurred.

Here is the trace:

```
Virtual Address Trace
  VA 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000003ec (decimal: 1004)
  VA 0x0000001d (decimal:   29) --> VALID in SEG0: 0x0000021d (decimal:  541)
  VA 0x00000050 (decimal:   80) --> SEGMENTATION VIOLATION (SEG1)
  VA 0x0000001e (decimal:   30) --> SEGMENTATION VIOLATION (SEG0)
  VA 0x00000058 (decimal:   88) --> VALID in SEG1: 0x000003d8 (decimal:  984)
  VA 0x00000061 (decimal:   97) --> VALID in SEG1: 0x000003e1 (decimal:  993)
  VA 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA 0x00000064 (decimal:  100) --> VALID in SEG1: 0x000003e4 (decimal:  996)
  VA 0x0000003d (decimal:   61) --> SEGMENTATION VIOLATION (SEG0)
  VA 0x0000000c (decimal:   12) --> VALID in SEG0: 0x0000020c (decimal:  524)
  VA 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000205 (decimal:  517)
  VA 0x0000002f (decimal:   47) --> SEGMENTATION VIOLATION (SEG0)
```

Now, let's try to recover some values. **Note: you may not be able to perfectly recover all values; if so, provide the best answer you can given the information above.**

(a) From this trace, what was the **base register of segment 0** set to?
To calculate this, find a valid reference to segment 0. For example:

```
VA 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000205 (decimal:  517)
```

Here you can see that virtual address 5 translates to physical address 517. Subtracting the offset into segment 0 (5) from 517 gets us 512, which must be the base address (0x400 for those who love hex).

(b) From this trace, what was the **bounds register of segment 0** set to? **For this, you need to find (in the best case) two references, one that is at virtual address N and is valid, and one that is at address N+1 but is not valid; that will tell us the bound precisely. And hence:**

```
VA 0x0000001d (decimal:   29) --> VALID in SEG0: 0x0000021d (decimal:  541)
VA 0x0000001e (decimal:   30) --> SEGMENTATION VIOLATION (SEG0)
```

This tells us that any virtual address from 0 through 29 is valid, whereas 30 is not. Hence, the size of segment zero is 30, which we thus conclude is the bound. If you want the physical address instead, it is base plus size, or 542.

(c) From this trace, what was the **base register of segment 1** set to? **Similar to the question above, but remember that segment 1 goes backwards:**

```
VA 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000003ec (decimal: 1004)
```

Thus, if 108 maps to 1004, we know that 127 (the last valid byte of the virtual address space) would map to 1023 (add 19 to 108 gets us 127; thus add 19 to 1004 to get 1023). Hence, the base (as described in the notes) would be 1024, just beyond the end of where the backward-growing segment lies.

(d) From this trace, what was the **bounds register of segment 1** set to?

The bound here is harder, because it is imprecise.

```
VA 0x00000050 (decimal:   80) --> SEGMENTATION VIOLATION (SEG1)
VA 0x00000058 (decimal:   88) --> VALID in SEG1: 0x000003d8 (decimal:  984)
```

As you can see, 80 goes too far backwards, but 88 is fine. With 88, the size must at least allow 88 through 127 to be valid; thus 40 bytes is the minimum segment size. It clearly could also be that 81 is valid, which adds 7 more bytes into the segment size, or 47. Thus, the size of this segment is between 40 and 47, inclusive.

Bonus: figure out exactly the parameters I ran the homework segmentation simulator with to get this problem.

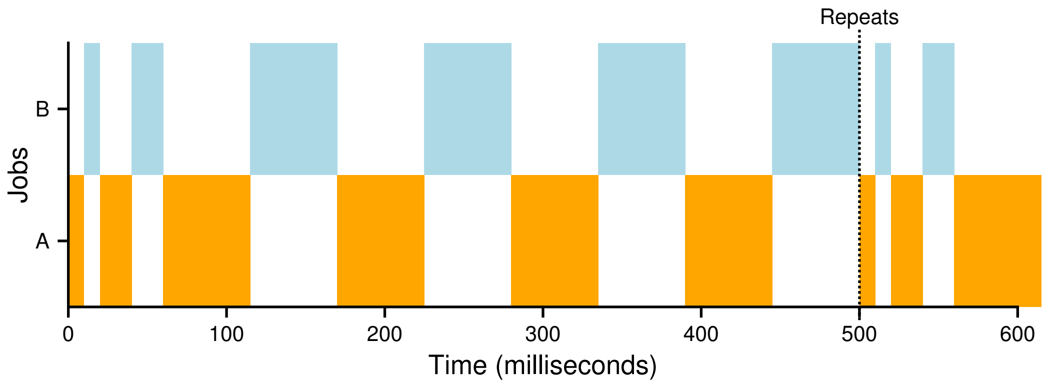
(e) In later versions of LostOS, OGP tells us that segmentation was dropped entirely and replaced with paging. What are some positives of the switch? What are some potential negatives? (i.e., why might you prefer segmentation over paging?)

Paging: No external fragmentation and thus much easier to manage from OS perspective; just hand out pages! But, paging has potentially high space costs (think of the size of a page table versus a couple of segment registers) and time costs (think of the TLB translation that has to occur).

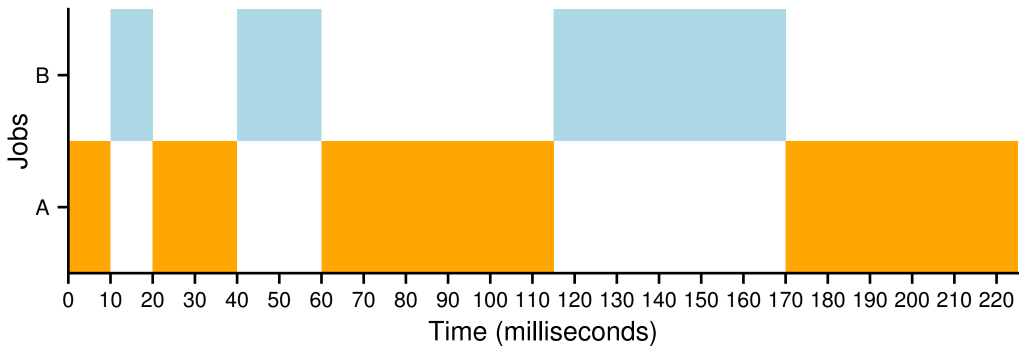
3. **Lost MLFQ Parameters.**

In the latest LostOS, there is an MLFQ (multi-level feedback queue) scheduler. Unfortunately, all of its parameters have been lost! Your job here is to examine a scheduling trace and determine what those critical parameters are.

Here is a timeline of what happens when two **CPU-bound** (no I/O) jobs, A and B, run:



This figure shows when A and B run over time; note that after 500 milliseconds, the behavior repeats, indefinitely (until the jobs are done). To help you further, a closeup of the first part of the graph is shown here:



Now, for some questions:

(a) How many queues do you think there are in this MLFQ scheduler? **Three queues. Check the closeup, and you will see three distinct time slices being used (10 ms, 20 ms, and 55ms).**

(b) How long is the time slice at the top-most (high priority) queue? **10 milliseconds (the first time slice you get).**

(c) How long is the time slice at the bottom-most (low priority) queue? **55 milliseconds (it's the longest time slice shown).**

(d) How often do processes get moved back to the topmost queue? **Seemingly every 500 milliseconds (though perhaps you'd need to see more to confirm that).**

(e) Why does the scheduling policy MLFQ move processes to higher priority levels (i.e., the topmost queue) sometimes? **Most important reason: to avoid starvation. Also, to reassess a job and see if it is behaving "differently" at different points in time.**

4. **Lost Translations**

An early version of the LostOS page replacement policy implemented a "perfect LRU" policy, with a little hardware support. The hardware support, according to the Old Gray Person (OGP), was as follows: on every memory reference to a particular VPN, the hardware would update a timestamp in the page-table entry (PTE) of that process's page table. In this version of the system, LostOS used a simple **linear page table**.

OGP tells you some more information to help you out with this: the original LostOS PTE contains only a **valid bit**, a **present bit**, the **timestamp** (32 bits, with the time in seconds since 1970), and the **page-frame number (PFN)** (30 bits). Thus, the entire PTE is 64 bits (or 8 bytes). Finally, OGP remembers that the timestamp field is used as a disk address when the page has been swapped to disk. Unfortunately, OGP can't remember what order these were stored in memory!

Fortunately, OGP did save some memory dumps of page tables, and is willing to share a few entries from them here. These are all in hex values, of course (OGP only speaks in hex).

The First Entries from the Page Table for Process 1:

```
C0 00 01 00 4A DC 70 04
80 00 00 00 00 00 10 00
7F FF FF FF FF FF FF FF
7F FF FF FF FF FF FF FF
...
```

The First Entries from the Page Table for Process 2:

```
C0 00 00 10 4A DC 90 CB
C0 00 00 11 4A DC 90 C0
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
...
```

Recall that hex F = binary 1111, E = 1110, D = 1101, C = 1100, B = 1011, A = 1010, 9 = 1001, 8 = 1000, and so forth.

OGP also remembers that when the valid bit was set to 0, the rest of the fields were filled with all 0's or all 1's (but not a mix of 0's and 1's).

Finally, OGP gives you can example of what the current time looks like, in seconds, in hex (isn't OGP nice?):

```
0x4ADCCD3F
(which is decimal 1255984438 seconds since January 1, 1970)
```

And now, time for some questions (next page):

(a) LostOS has a valid bit; in general, what is the valid bit used for?

Valid bit: Please don't just say it is to check whether something is valid, that is redundant! Better answer: to check whether a particular virtual page has been allocated and is thus now a part of the address space of a process.

(b) LostOS has a present bit; in general, what is the present bit used for?

The present bit determines whether a VALID page is in memory or on disk. If on disk, access to said page will cause a page fault to occur.

(c) Given the information above, what do you think the likely structure of the page-table entry (PTE) is? (draw a picture, and show clearly which bits you think are used for which purposes) **Top bit: Valid; Next bit: present; then 30 bits of PFN, then 32 bits of timestamp.**

(d) Assuming your PTE structure, what would happen when **process 1** accesses virtual page 0? **Must consult the page table of process 1 to see. First entry is C0 00 01 00 4A DC 70 04. Thus, valid and present with a PFN of 0x0100.**

(e) Assuming your PTE structure, what would happen when **process 1** accesses virtual page 1? **Looks to be valid but NOT present; disk address 0x1000.**

(f) Assuming your PTE structure, what would happen when **process 1** accesses virtual page 2? **Not valid; segmentation violation.**

(g) Assuming your PTE structure, what would happen when **process 2** accesses virtual page 0? **Valid and present; PFN is 0x10.**

(h) Assuming your PTE structure, what would happen when **process 2** accesses virtual page 1? **Valid and present; PFN is 0x11.**

(i) Assuming your PTE structure, what would happen when **process 2** accesses virtual page 2? **Not valid; segmentation violation.**

5. **Lost Replacement Policies (Part II).**

   Note: You should read Question 4 before reading/doing this question.

   In this question, we try to uncover the some of the policies LostOS used to replace pages. As was stated above, early versions of LostOS implements "perfect" LRU replacement policy by putting timestamps in the page table of each process, and having the hardware update said timestamps when a page is referenced (i.e., read from or written to).

   (a) Given this hardware support, what work does the OS have to do to implement perfect LRU? What is bad about this approach?

   If the hardware sets a timestamp in the page table on each access, the OS would have to scan every valid entry of all page tables to find the LRU page, and thus choose to replace it. This is an expensive action, taking a lot of CPU time!

   At some point, OGP decided to change the system to use a simpler policy that only approximates LRU. Thus, the first change is to the hardware, to replace the timestamp on each access with a single bit, called the **reference bit**. Upon each access to a page, the hardware sets a reference bit for that page to 1.

   The second change is to the OS to use this bit to make replacement decisions. Unfortunately, this code has been lost as well. Assume the following data structures (note: the PTE structure uses integers for simplicity of coding, but in actuality would pack the data much more carefully using bitwise operators):

```
// a page table entry (PTE)
// NOTE: integers used for simplicity
typedef struct __pte_t {
  int      valid;
  int      present;
  int      referenced;
  unsigned pfn;        // the translation
} pte_t;

// a linear page table
// NOTE: if a process exists, the entire table is allocated
typedef struct __page_table_t {
  pte_t    pagetable[PAGE_TABLE_ENTRIES];
} page_table_t;

// the array of all page tables in system
// NOTE: if process X does not exist, p[X] will be NULL;
page_table_t *p[MAX_PROCESSES];
```

Continued on next page...

(b) Write the code that picks a page to replace by finding one that has not recently been referenced. Things to consider: Where to start the search for unused pages? Should the reference bit ever be cleared? For simplicity, the code should simply set two integers (pointers to which are passed into the `replace()` routine): `process`, to the process whose page should be replaced, and `VPN`, to the virtual page number of the page to be replaced.

```
// any global variables?


void replace(int *process, int *VPN) {
    // these are the variables you update with the answer
    *process =
    *VPN =
}
```

The basic idea here is to scan through all the processes in the system, checking page tables for a valid and present page that has not been referenced. A good answer made sure that page table entries were indeed valid and present! A good answer also cleared the reference bits as it went, so that the system had a way to actively differentiate which pages are really being accessed. Finally, a good answer didn't always start at the same place (say with process 0 and vpn 0), because that would be unfair to said process; better to start at a random location or keep track of where you looked last in a global variable and start your search there.

```
for (int P = rand(MAX_PROCESSES); P < MAX_PROCESSES; P = (P + 1) % MAX_PROCESSES) {
    if (p[P] != NULL) {
        for (page = 0; page < PAGE_TABLE_ENTRIES; page++) {
            if (p[P]->pagetable[page].valid &&
                p[P]->pagetable[page].present) {
                if (p[P]->pagetable[page].referenced == 0) {
                    *process = P;
                    *VPN = page;
                    return;
                } else {
                    p[P]->pagetable[page].referenced = 0;
                }
            }
        }
    }
}
```

6. **Lost Material.**

   In this final question, this exam asks you questions about other parts of the material we covered. This "lost" material could have been developed into fuller, harder questions, but you only have 2 hours, after all.

   Please circle "True" if the statement is true, and "False" if it is, well, false.

   (a) The main reason to have a multi-level page table is to speed up address translation. True False
   False. Main reason is to save space in memory.

   (b) The main reason to have a hardware TLB is to speed up address translation. True False
   True. That is the point of the TLB, to cache address translations and hence speed up the entire process.

   (c) Using a multi-level page table increases TLB hit time. True False
   False. TLB hit time is not affected by page-table structure.

   (d) As the amount of addressable physical memory grows, the size of each page-table entry (PTE) must grow as well. True False
   Generally true, as the PFN must get bigger, and thus the PTE. Could argue that the PTE might be big enough already to accommodate a slightly bigger PFN...

   (e) As the size of the virtual address space grows, the amount of space occupied by a linear page table also grows. True False
   True.

   (f) In a 2-level multi-level page table, each page directory entry should have a valid bit. True False
   True. This allows us to save space in memory by having large regions of page table left unallocated.

   (g) In a 2-level multi-level page table, the size of the page directory increases as the virtual address space gets larger. True False
   True. You need more page-directory entries to point to all those parts of the page table.

   (h) When running on a virtual machine monitor, the OS still thinks it is in charge of the physical placement of pages. True False
   True. It is an illusion after all.

   (i) When running on a virtual machine monitor, TLB hits take longer. True False
   False. TLB hits, all done in hardware, should take the same time.

   (j) When running on a virtual machine monitor, TLB misses take longer. True False
   True. Have to bounce into the VMM, which may bounce into the OS, and then back to the VMM. However, a software cache of translations in the VMM could help with this...