CS-537: Midterm Exam (Fall 2011)

You Are The Grader

Please Read All Questions Carefully!

There are thirteen (13) total numbered pages (1, 2, 3, and then 1, ..., 10).

Please put your NAME and student ID on THIS page.

Name and Student ID: SOLUTIONS

2

You have found the lost exam of a famous student, Ime A. Stoodant. Ime took this exam years ago, but never had it graded or returned. It's kind of a sad story, really: once the exam was lost, Ime dropped out of school and never completed 537. Because Ime never finished 537, Ime never really learned what a semaphore was, how a context switch worked, or how to compute an address translation with a multi-level page table. As you might have guessed, Ime's life was ruined due to this lack of OS knowledge.

But now the lost exam has been found! Good news for Ime, and for you too, because **you are going to grade Ime's exam**. Specifically, **FOR EACH QUESTION on Ime's exam:**

- If Ime's answer is correct, just mark it correct (with a check mark).
- If Ime's answer is wrong, mark it wrong (with an X) AND **PROVIDE THE CORRECT ANSWER.**

Here are two examples. The first question is answered correctly, and thus all you would need to do is mark it correct with a check mark. The second question is answered incorrectly, and thus you should both mark it incorrect as well as provide the correct answer

1. What does the acronym TLB stand for?

TLB stands for Translation Lookaside Buffer. ✓ good job!

2. What are TLBs used for?

TLBs are used to make page tables smaller ✗

No! TLBs are used to make paging <u>faster</u>!
(by keeping a small cache of popular address translations, a TLB helps avoid costly page table lookups)

HOW YOU WILL BE GRADED:

You have to be graded too! Sorry about that. What I'll be grading is how well you did in grading Ime's exam. Thus, if Ime got it right and you marked it with a check, you will get full credit for that question. If Ime got it wrong and you mark it wrong **AND** provide the right answer, you also get full credit. It's as simple as that!

Good luck!

AND PLEASE READ (and GRADE) EACH QUESTION CAREFULLY!

Grading Page

	Points	Total Possible
Q1	5	5
Q2	5	5
Q3	5	5
Q4	5	5
Q5	5	5
Q6	5	5
Q7	5	5
Q8	5	5
Q9	5	5
Q10	5	5
Q11	5	5
Q12	5	5
Q13	5	5
Q14	5	5
Q15	5	5
Q16	5	5
Q17	5	5
Q18	5	5
Q19	5	5
Q20	5	5
Q21	5	5
Q22	5	5
Total	110	110

but -10 because
so late! => 100 still
good!

CS-537: Midterm Exam (Spring 1978)
MIDTERM WITHOUT CLEVER THEME OR TITLE

Please Read All Questions Carefully!

There are ten (10) total numbered pages.

Please put your NAME and student ID on THIS page.

Name and Student ID: Ime A. Stoodant
(student ID: 42)

INSTRUCTIONS: These questions are all short answers. Good luck! And remember, no using your slide rule.

1. Most hardware provides user mode and kernel mode; applications are run in user mode, while the OS runs in kernel mode. The tricky part is transitioning from one mode to the other, which is done via a system call. What takes place during a system call?

all ok but

1) Jump into kernel (trap)
2) Run kernel code as designated by trap table
3) return from trap

missing:
=> raise/lower privilege
=> save/restore registers

2. To support trap handling, the OS has to set up a trap table of some kind; the trap table is used by the hardware to know which OS code to run when a particular trap occurs. How does the OS inform the hardware of the location of the trap table?

✓ When booting, the OS uses a special hardware instruction to tell the h/w the location of the trap table.
Important: this instruction is privileged
(otherwise any process could install a trap table!)

3. Timer interrupts are a useful mechanism for the OS. Why?

✗ To keep track of time, duh!

Most important:
allows OS to keep control of machine

Also: allows implementation of certain scheduling policies

4. A typical OS provides some APIs to create processes. In Unix-based systems, fork(), exec(), and wait() are used. Write some code that uses these system calls to launch a new child process, have the child execute a program named "hello", (with no arguments), and have the parent wait for the child to complete.

X

```
int done=0;
int rc = fork();
if (rc==0) { //child
   char *argv [2];
   argv [0] = strdup("hello");
   argv [1] = NULL; //important!
   execvp (argv[0], argv);
   done=1;
} else if (rc>0) { //parent
   while (done==0)
      ; //spin
}
```

-> doesn't work at all (done is not shared across parent/child)

-> just use wait(); in parent instead

5. To compare scheduling policies, we often use certain metrics. Here we use a new metric: total completion time. The total completion time tells us when all of the jobs in a workload are done. What do you think about this metric?

X I think it's great professor! Very useful.

OK answer: not a useful metric as all policies would seem more or less the same.

Nuance: the metric would measure something about context-switch overhead (thus could have some use)

6. One of the best scheduling policies out there is shortest-time-to-completion first, or STCF. What does STCF do? If it's so great, why do we rarely see it implemented?

STCF: picks the job with the least time left to run and runs it. It's the preemptive generalization of SJF.

X why it's rarely used: the OS is implemented in C => always hard to write correct C code.

Impossible to implement in general because we don't know job run time (in most cases)

7. In the MLFQ scheduler, there is a rule that states the following:

> Rule 5: After some time period S, move all the jobs in the system to the top-most queue.

What is the purpose of this rule?

X To confuse me?

Primarily to avoid starvation; also to re-learn something about job, which is useful if job changes from batch ↔ interactive

8. Lottery scheduling uses tickets to represent a share of the CPU. Thus, if process A has 100 tickets, and another process B has 200, process A should run roughly 1/3 of the time. Does lottery scheduling prevent starvation?

No! Lottery scheduling has been around for a long time, and there are still people starving all around the world. ☹

Probabilistically yes (though things can get pretty bad if ticket allocation is wildly imbalanced)

9. A user can allocate memory by calling malloc() and free it by calling free(). When a user doesn't call free(), it is called a ''memory leak''. When is it OK to have a memory leak in your program?

NEVER! (how dare you even suggest this!)

if the program is short-lived (just about to exit), there is little point in freeing memory; the OS cleans up the entire address space anyhow.

10. In a system using a base-and-bounds pair of registers to virtualize a tiny 1KB address space, we see the following address reference trace:

```
Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  2: 0x000003a1 (decimal:  929) --> SEGMENTATION VIOLATION
```

What can we say about the value of the base register? What abount the bounds register?

Compute base by subtracting: $14611 - 776 = \boxed{13835}$ ✓

Compute base again : $14432 - 579 = 13853$ ✗ ~~(base changed?)~~ ✓

597 $= \boxed{13835}$

Bounds: ~~can't~~ say anything about it (not enough info)

recall: bounds is size of valid region

we know 776 is valid, 929 isn't; thus

$776 < \text{bounds} \leq 929$

11. Base and bounds (or dynamic relocation) has some strengths and weaknesses as a mechanism for implementing virtual memory. What are they? (list)

Strengths

→ simple ✓

→ Fast ✓

→ provides protection

→ little space overhead

Weaknesses

→ Not complex ✗

→ Not slow ✗

→ not flexible

→ doesn't support sparse address spaces

→ depending on layout of address space, can have either internal or external fragmentation

12. Segmentation is a generalization of base and bounds. What are its strengths and weaknesses? ✗ umm...., something about fragmentation?

Strengths

→ simple

→ fast

→ protection

→ little space overhead

→ somewhat supports sparse AS

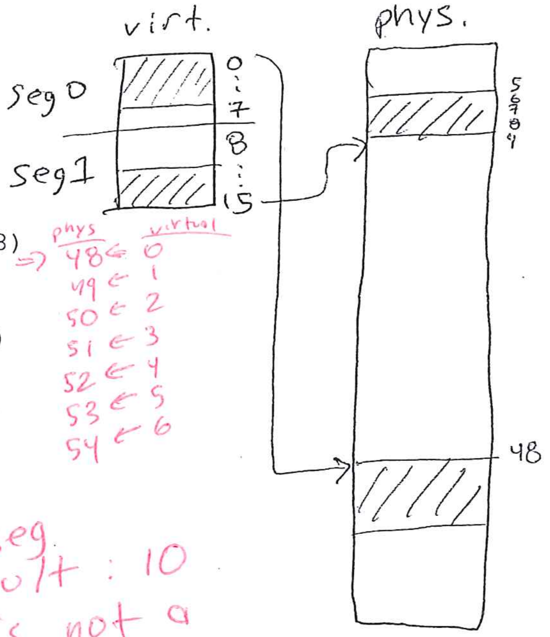
→ less internal fragmentation

Weaknesses

→ still not fully flexible (must match segments to how AS is used)

→ external fragmentation a big problem

13. Here we have a trace of virtual address references in a segmented system. The system has two segments in each address space, and uses the first bit of the virtual address to differentiate which segment a reference is in. Segment 0 holds code and a heap (and therefore grows in the positive direction); Segment 1 holds a stack (and therefore grows in the negative direction). Please translate the following references, or mark a segmentation violation if the address reference is out of bounds.



The address space size is 16 bytes (tiny!).

The physical memory is only 64 bytes (also tiny!).

Here is some segment register information:

```
  Segment 0 base  (grows positive) : 0x30 (decimal 48)
  Segment 0 limit                  : 7

  Segment 1 base  (grows negative) : 0x09 (decimal 9)
  Segment 1 limit                  : 4
```

phys	virtual
⇒ 48 ←	0
49 ←	1
50 ←	2
51 ←	3
52 ←	4
53 ←	5
54 ←	6

VA	PA
15 →	8
14 →	7
13 →	6
12 →	5

And here is the Virtual Address Trace:

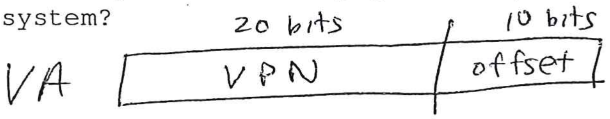
VA 0: 0x0000000e (decimal: 14) --> 7 ✓

VA 2: 0x00000006 (decimal: 6) --> 54 ✓

VA 4: 0x0000000a (decimal: 10) --> 3 X seg. fault : 10 is not a valid virtual address

14. A system uses paging to implement virtual memory. Specifically, it uses a simple linear page table. The virtual address space is of size 1 GB (30 bits); the page size is 1 KB; each page table entry holds only a valid bit and the resulting page-frame number (PFN); the system has a maximum of 2^{15} physical pages (32 MB of physical memory can be addressed at most).

How much memory is used for page tables, when there are 100 processes running in the system?

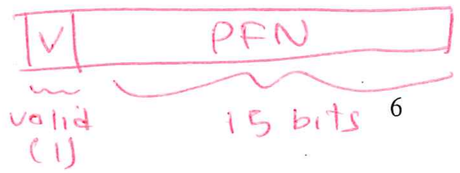


⇒ size of page table is thus 2^{20} or $\boxed{1 \text{ MB}}$ X

⇒ if 100 processes, X $\boxed{100 \text{ MB}}$

to calculate properly, must know size of page table entry (PTE)

PTE :



⇒ 2 bytes

⇒ 2^{20} x 2 bytes / page table ⇒ 2 MB

x 100 processes ⇒ 200 MB

15. TLBs are critical in the implementation of a virtual memory. Assume the following code snippet:

```
int i;
int p[1024];
for (i = 0; i < 1024; i++)
    p[i] = 0;
```

Describe the TLB behavior during this sequence. How many hits and how many misses take place when this code is first run? Assume a 1-KB page size.

X Too hard! You are a mean professor.

Each int is 4 bytes => 1024 ints => 4 KB

If they line up exactly on a page boundary,
=> 4 pages, otherwise 5

Also, code pages for instruction fetch:
=> either 1 (or two if code spans 2 pages)

Thus, 4 -> 6 TLB misses

16. A system uses multi-level page tables to implement virtual memory. It's a simple two-level table, with a page directory that points to pieces of the page table. Each page directory is only a single page, and consists of 32 page-directory entries (PDEs). A PDE has a simple format: a valid bit followed by the PFN of the page of the page table (if valid). Here is one entire page directory (wrapped across two lines):

```
7f fe 7f 7f 7f d4 7f 7f 7f 9e 7f ad 7f 7f 7f d6
7f 7f 7f 7f 7f 7f 7f e9 a1 e8 7f 7f 7f 7f 7f 7f
```

How much space savings (in bytes) does this multi-level page table provide, as compared to a typical linear page table?

x7f => 0 1111111
valid bit = 0

PDE: | V | PFW |

above: I circled valid entries => 8 total
(24 not valid)

X => 24 pages savings (x32 bytes for byte savings)

Almost! You forgot about the page directory itself! Thus, only 23 pages saved

17. LRU is considered a good page-replacement policy. Describe what a good page-replacement policy should do, what LRU (least-recently-used) replacement is.

A good policy minimizes the miss rate by keeping popular pages in memory, LRU does this by ~~keeping~~ the least-recently-used pages in memory. evict LRU, don't keep it!

OK

18. LRU also has its downsides. Describe a case where LRU behaves really poorly.

Trick question! LRU never does poorly! X

Easy: Loop over M pages repeatedly; have N pages of memory, and ensure M > N. In this case, LRU is pessimal (worst possible), missing on each access

19. LRU is hard to implement perfectly. Instead, most systems use reference bits to approximate LRU. How are reference bits used?

Reference bit gets set (in PTE) when a page is referenced (accessed), Thus, to evict a page, all OS has to do is scan and find a page w/ ref bit = 0.

X you forgot: how do the ref bits get cleared? (your approach might end up w/ all 1's) Clock scans and clears the ref bit when the page is not evicted

20. The atomic exchange (or test-and-set) instruction returns the old value of a memory location while setting it to a new value, atomically. It can be used to implement a spin lock. Please do so!

```
struct lock_t {
  ✓ int flag;
}

⇒ mutex_lock (struct lock_t *m) {
      while (xchg (&m->flag, 2) == 2)
            ; // spin
  }
```

Unusual, but OK!

```
⇒ mutex_init (struct lock_t *m) {
  ✓ m->flag = 0;
  }

⇒ mutex_unlock (struct lock_t *m) {
  ✓ m->flag = 0;
  }
```

21. Condition variables are useful in certain types of multi-threaded programs. Describe what a condition variable is, and show an example of how you would use one in a little code snippet.

forgot to study this → be kind ☺

CV is a queue with two operations;
wait () puts calling thread to sleep on that queue (also releases lock)
signal () wakes one sleeping thread off of queue

```
child () {
   lock (m);
   done = 1;
   signal (c);
   unlock (m);
}

parent () {
   done = 0;
   create (child);
   lock (m);
   while (done == 0)
       wait (c, m);
   unlock (m);
}
```

example: implement a parent thread waiting for a child to complete;
note: wait () releases lock

22. In class we discussed the producer/consumer problem, and provided this (broken) solution:

```
void *producer(void *arg) {
  int i;
  while (1) {
    mutex_lock(&mutex);              // line p1
    if (count == MAX)                // line p2
      cond_wait(&empty, &mutex);     // line p3
    put(i);                          // line p4
    cond_signal(&full);              // line p5
    mutex_unlock(&mutex);            // line p6
  }
}

void *consumer(void *arg) {
  int i;
  while (1) {
    mutex_lock(&mutex);              // line c1
    if (count == 0)                  // line c2
      cond_wait(&full, &mutex);      // line c3
    int tmp = get();                 // line c4
    cond_signal(&empty);             // line c5
    mutex_unlock(&mutex);            // line c6
    printf("%d\n", tmp);
  }
}
```

Describe why this solution is broken, and demonstrate it with a specific example of thread interleavings (assume two consumers and one producer):

Something to do with the if? almost!

if => while solves problem

problem occurs when:

T1 c1 c2 c3 | sleep (awake)

T2 P1 P2 P4 P5 | P6

woke T1

T3 c1 c2 c4 c5 c6

C4

tries to get() from empty queue!

time