University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2013 Anthony D. Joseph

Midterm Exam *Solutions*

March 13, 2013
CS162 Operating Systems

Your Name:	
SID AND 162 Login:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book and one 2-sided handwritten note** examination. You have 80 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. ***Make your answers as concise as possible.*** If there is something in a question that you believe is open to interpretation, then please ask us about it!

Good Luck!!

QUESTION	POINTS ASSIGNED	POINTS OBTAINED
1	28	
2	25	
3	17	
4	15	
5	15	
TOTAL	100	

1. (28 points total) True/False and short answer questions:
   a. (12 points) True/False and Why? **CIRCLE YOUR ANSWER.**
      i) The four conditions that must hold in order for deadlock to occur are: Hold-and-wait, circular waiting, starvation and mutual exclusion.

TRUE FALSE

Why? (One sentence)
FALSE. *Starvation is not a condition for deadlock. It should be no preemption. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

ii) The main advantage of multilevel page tables is that they use page table memory efficiently.

TRUE FALSE

Why? (One sentence)
TRUE. *Multilevel page tables use memory more efficiently for sparse address spaces than single level tables. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

iii) In the Nachos priority scheduler, if a **HIGH** priority thread is waiting for a **LOW** priority thread to release a lock, but there are *NO OTHER THREADS* in the system, the **LOW** priority thread's effective priority should be **LOW**.

TRUE FALSE

Why? (One sentence)
FALSE. *Priority locks always donate priority when appropriate, independent of the number of threads in the system. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

iv) In Nachos, a thread's effective priority can only change when it is waiting on a thread queue.

TRUE FALSE

Why? (One sentence)

FALSE*. A thread's effective priority can also change if it owns a resource. The correct answer was worth 1 points and the justification was worth an additional 2 points.*

b. (16 points) Short Answer Questions:

i) (4 points) Give a two to three sentence brief description of the difference between starvation and deadlock.

Starvation implies that a thread cannot make progress because other threads are using resources it needs. Starvation can be recovered from if, for example, the other processes finish. Deadlock is a circular wait without preemption that can never be recovered from.

ii) (4 points) When using the Banker's algorithm for resource allocation, if the system is in an unsafe state, will it always eventually deadlock? Briefly (1-2 sentences) state why or why not.

No, because processes may not request their total possible resources, and may release some resources before acquiring others. Full credit was given for saying the Banker's algorithm does not allow a system to enter an unsafe state.

iii) (4 points) In two to three sentences briefly discuss why caching is *increasingly* important in modern computer systems, and why it is of particular concern to the operating system.

Caching is important because the performance gaps in the memory hierarchy are increasing: CPU to memory, memory to disk, etc. Increasing capacity means we can cache more at higher levels of the hierarchy.

iv) (4 points) In two to three sentences briefly explain why the space shuttle failed to launch on April 10, 1981.
As described in Garman's "The Bug Heard 'round the World," paper, due to software changes, the PASS could with low probability (1 in 67) incorrectly initialize the system time. This resulted in the PASS being one cycle out of synchronization with the BFS. This caused the first shuttle launch to abort 20 minutes prior to the scheduled launch. The bug points out the challenges of building and maintaining real-time systems, even when hundreds of programmers are involved and hundreds of hours are spent on testing.

2. (25 points) Synchronization primitives: Consider a machine with hardware support for a single thread synchronization primitive, called Compare-And-Swap (CAS). Compare-and-swap is an atomic operation, provided by the hardware, with the following pseudocode:

```
int compare_and_swap(int *a, int old, int new) {
  if (*a == old) {
     *a = new;
     return 1;
  } else {
     return 0;
  }
}
```

Your first task is to implement the code for a simple spinlock using compare-and-swap. You are not allowed to assume any other hardware or kernel support exists (*e.g.,* disabling interrupts). You may assume your spinlock will be used correctly (*i.e.*, no double release or release by a thread not holding the lock)

a. (3 points) Fill in the code for the `spinlock` data structure.

```
struct spinlock { /* Fill in */

int value = 0;
```

We deducted one point for extraneous statements.

```
}
```

b. (4 points) Fill in the code for the `acquire` data function.

```
void acquire(struct spinlock *lock) { /* Fill in */

while (cas(&lock->value, 0, 1) == 0)
   ;    /* spin */
```

We deducted two points for extraneous statements, two points for a missing while and four points if your solution did not work.

```
}
```

c. (4 points) Fill in the code for the `release` data function.

```
void release(struct spinlock *lock) { /* Fill in */

lock->value = 0;
```

We deducted two points for extraneous statements, one point for an unnecessary Compare-and-Swap (stores of a word are atomic), and four points if your solution did not work.

```
}
```

After completing your implementation, you realize that using a spinlock is inefficient for applications that may hold the lock for a long time. You consider using the following two primitives to implement more efficient locks: `atomic_sleep` and `wake`.

`atomic_sleep` is an atomic operation, provided by the hardware, with the following pseudocode:

```
  void atomic_sleep(struct *lock, int *val1, int val2){
       *val1 = val2;  /* set val1 to val2 */
       enqueue(lock);     /* put current thread on a
                             lock's wait queue*/
       sleep();     /* put current thread to sleep */
  }
```

`wake` is non-atomic with the following pseudocode:

```
  void wake(struct lock *lock){
    dequeue(); /* remove a thread (if any) from lock's
                  wait queue and add it to the
                  scheduler's ready queue */
  }
```

Your second task is to reimplement your lock code more efficiently using `atomic_sleep` and `wake`. You may use Compare-And-Swap if you want. You are not allowed to assume any other hardware or kernel support exists (*e.g.,* disabling interrupts).

d. (4 points) Fill in the code for the **new** `lock` data structure.

```
struct lock { /* Fill in */

int guard = 0;
int value = 0;
Queue queue = NIL;
```

We deducted two points for a missing guard or value and one point for extraneous variables.

```
}
```

e. (5 points) Fill in the code for the **new** `acquire` data function.

```
void acquire(struct lock *lock) { /* Fill in */

  while (1) {
     while (cas(&lock->guard, 0, 1) == 0);
     if (value == 1) {
        atomic_sleep(&lock, &lock->guard, 0);
     } else {
        lock->value = 1;
        lock->guard = 0;
        return;
     }
  }
```

We expected solutions that used a spinlock on a guard for protecting the lock variable, and atomic_sleep for efficient waiting for the lock. We deducted two points for extraneous statements, two points for not setting the lock variable, one point for a missing outer while loop, 2 points for a missing guard, 2 points for not sleeping, one point for misusing the guard, and one point for a dangerous double release of the guard in acquire and release.

```
}
```

f. (5 points) Fill in the code for the **new** `release` data function.

```
void release(struct lock *lock) { /* Fill in */

   while (cas(&lock->guard, 0, 1) == 0);
   lock->value = 0;
   wake(&lock);
   lock->guard = 0;
```

We deducted two points for extraneous statements, two points for no guard, two points for failing to wake a waiting thread, and two points if your solution had other errors, such as not releasing the lock.

```
}
```

3. (17 points total) Memory management:

a. (7 points) Consider a memory system with a cache access time of 10ns and a memory access time of 200ns, *including the time to check the cache*. What hit rate H would we need in order to achieve an effective access time 10% greater than the cache access time? (Symbolic and/or fractional answers are OK)

Effective Access Time: $T_e = H * T_c + (1 - H) * T_m$,
where $T_c = 10ns$, $T_e = 1.1 * T_c$, *and* $T_m = 200ns$.

Thus, $(1.1)(10) = 10H + (1 - H)200$
$11 = 10H + 200 - 200H$
$-189 = -190H$
$H = 189/190$

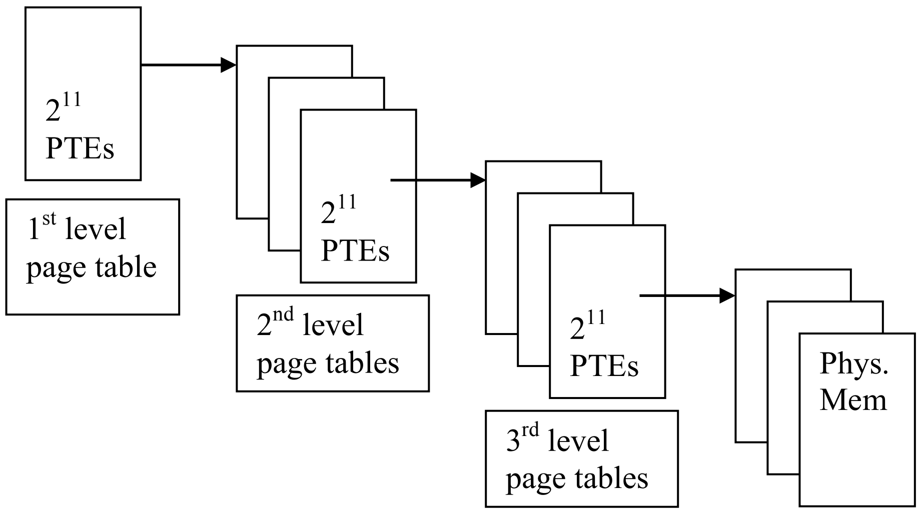
We awarded 4 pts for the correct formula and 3 pts for the correct answer. Many students missed the fact that the miss time includes ***both*** *the memory access time and the cache access time. If the formula was missing the cache access time, we deducted two points – if the answer based upon this incorrect formula was correct, we did not deduct any additional points.*

b. (10 points) Suppose you have a 47-bit virtual address space with a page size of 16 KB and that page table entry takes 8 bytes. How many levels of page tables would be required to map the virtual address space if every page table is required to fit into a single page? Be explicit in your explanation and show how a virtual address is structured.

A 1-page page table contains 2,048 or 2^{11} *PTEs* ($2^3 * 2^{11} = 2^{14}$ *bytes), pointing to* 2^{11} *pages (addressing a total of* $2^{11} * 2^{14} = 2^{25}$ *bytes). Adding a second level yields another* 2^{11} *pages of page tables, addressing* $2^{11} * 2^{11} * 2^{14} = 2^{36}$ *bytes. Adding a third level yields another* 2^{11} *pages of page tables, addressing* $2^{11} * 2^{11} * 2^{11} * 2^{14} = 2^{47}$ *bytes. So, we need* ***3 levels****.*

The correct answer is worth 5 pts. Correct reasoning is worth up to 5 pts (1 pt for identifying that there are 2^{11} *PTEs per page, 2 pts for describing how page tables are nested, and 2 pts based upon the quality of the argument).*

11 bit page	*11 bit page*	*11 bit page*	*14 bit offset*



4. (15 points total) Concurrency control: Building H_2O_2.

The goal of this exercise is for you to create a monitor with methods `Hydrogen()` and `Oxygen()`, which wait until a Hydrogen Peroxide molecule (H_2O_2) can be formed and then return. Don't worry about actually creating the Hydrogen Peroxide molecule; instead only need to wait until two hydrogen threads and two oxygen threads can be grouped together. For example, if two threads call Hydrogen, another thread calls Oxygen, and then a fourth thread calls Oxygen, the fourth thread should wake up the first three threads and they should then all return.

a. (3 points) Specify the correctness constraints. Be succinct and explicit in your answer.

1) Each hydrogen thread waits to be grouped with one other hydrogen and two oxygen threads before returning.
2) Each oxygen thread waits for another oxygen thread and two other hydrogens before returning.
3) Only one thread access shared state at a time
3 good answers received 1 point each. Full three points if the two correctness constraints for hydrogen and oxygen were CORRECTLY combined into one constraint. We deducted one point if there was no shared state constraint.

b. (12 points) Observe that there is only one condition any thread will wait for (i.e., a hydrogen peroxide molecule being formed). However, it will be necessary to signal hydrogen and oxygen threads independently, so we choose to use two condition variables, `waitingH` and `waitingO`.

State variable description	**Variable name**	**Initial value**
Number of waiting hydrogen threads	`wH`	`0`
Number of waiting oxygen threads	`wO`	`0`
Number of active hydrogen threads	`aH`	`0`
Number of active oxygen threads	`aO`	`0`

You start with the following code:

```
Hydrogen() {
  wH++;
  lock.acquire();
  while (aH == 0) {
    if (wH >= 2 && wO >= 2) {
      wH-=2; aH+=2;
      wO-=2; aO+=2;
      waitingH.broadcast();
      waitingO.broadcast();
    } else {
      waitingH.wait(&lock);
      lock.acquire();
    }
  }
  aH--;
  lock.release();
}

Oxygen() {
  wO++;
  lock.acquire();
  while (aO == 0) {
    if (wH >= 2 && wO >= 2) {
      wH-=2; aH+=2;
      wO-=2; aO+=2;
      waitingH.signal();
      waitingH.signal();
      waitingO.signal();
    } else {
      waitingO.wait(&lock);
    }
  }
  aO--;
  lock.release();
}
```

For each method, *say whether the implementation either (i) works, (ii) doesn't work, or (iii) is dangerous* – that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why (there maybe several errors) and briefly show how to fix it so it does work. Also, list and fix any inefficiencies. **You do not have to reimplement the methods.**

i. `Hydrogen()`

Nine points total. Correctness: 1 point for saying the routine is either (ii) ***doesn't work*** *or (iii)* ***dangerous*** *– depending on your answer for bug #2 below. Two points for each of three bugs and associated fixes (if the bugs mentioned implied a fix, then full credit was given):*

1. *Counter: The state variable wH is modified outside of a critical section*
2. *Potential deadlock: the thread attempts to reacquire the lock after waiting (when it already holds the lock). Depending on the implementation of locks, this could be a no-op or could deadlock.*
3. *Inefficient: The Hydrogen broadcast should be a signal.*
4. *Inefficient: The Oxygen broadcast should be two signals.*

We took off 1 point for any additional bugs, since there were no others, except for a complete rewrite of the implementation – which was not what we were looking for.

ii. `Oxygen()`

Three points total. Correctness: One point for (iii) ***dangerous****.*
Two points for each reason:

1. *Counter: State variable wO modified outside of a critical section.*

5. (15 points total) Scheduling. Consider the following processes, arrival times, and CPU processing requirements:

Process Name	Arrival Time	Processing Time
1	0	4
2	2	3
3	5	3
4	6	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). For RR and SRTF, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it. Also, assume that the currently running thread is not in the ready queue while it is running. The turnaround time is defined as the time a process takes to complete after it arrives.

Time	**FIFO**	**RR**	**SRTF**
0	**1**	**1**	**1**
1	*1*	*1*	*1*
2	*1*	*2*	*1*
3	*1*	*1*	*1*
4	*2*	*2*	*2*
5	*2*	*3*	*2*
6	*2*	*4*	*2*
7	*3*	*1*	*4*
8	*3*	*2*	*4*
9	*3*	*3*	*3*
10	*4*	*4*	*3*
11	*4*	*3*	*3*
Average Turnaround Time	*((4-0)+(7-2)+(10-5)+(12-6))/ 4 =* ***5***	*((8-0)+(9-2)+(12-5)+(11-6))/4 =* ***6.75***	*((4-0)+(7-2)+(12-5)+(9-6))/4=* ***4.75***

Each column is worth 5 points: 3 for correctness of the schedule (we deducted 1/2/3 points if you made minor/intermediate/major mistakes), and 2 for the average Turnaround time (1 point was deducted for minor errors).