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

Spring 2013 Anthony D. Joseph

Midterm Exam

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)

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

TRUE FALSE

Why? (One sentence)

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)

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

TRUE FALSE

Why? (One sentence)

b. (16 points) Short Answer Questions:

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

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.

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.

iv) (4 points) In two to three sentences briefly explain why the space shuttle failed to launch on April 10, 1981.

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 */









}
```

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

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

















}
```

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

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









}
```

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 */








   }
```

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

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

   }





















}
```

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

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













}
```

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)

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.

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.

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()`

ii. `Oxygen()`

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			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
Average Turnaround Time			