Graduate Operating Systems
Spring 2016
Midterm Exam

Total time: 75 minutes; Total points: 100

Name: **Sol Ution**

In recognition of and in the spirit of University of South Florida Honor Code, I certify that I will not use unpermitted aid on this exam.

Signature: ______________________________

This examination is closed books and notes. No electronic and digital devices are allowed. You may not collaborate in any manner on this exam.

As with any exam, you should read through the questions first and start with those that you are most comfortable with. Answers will be graded for substance and clarity. Redundancy and vague generalities are considered bad form. Partial credit will be offered only for meaningful progress towards solving the problems.

Grading:

Problems 1, 2: Anda
Problems 5, 7: Hernan
Problems 3,4,6: Subu

Note: I found some mistakes in the solutions posted originally. They are highlighted in green.

1	/33
2	/8
3	/12
4	/10
5	/8
6	/12
7	/17
Total	/100

1. **Short attention span** (33 points): For each of the following statements indicate whether the statement is always true (T) or not (F) (1 point), and explain briefly why (2 points).
    a. Adding TLB to a system that doesn't have one never hurts performance.
    False. There is an inherent cost to accessing the TLB when page not in TLB, so it depends.

    b. It is possible in a single-processor system to not only interleave the execution of multiple processes but also to overlap their execution on the CPU.
    False. One CPU can only execute instructions from one process.

    c. Race condition is a situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
    False. This is typically called livelock. A race condition is the situation that leads to non-determinism due to the order in which processes are scheduled.

    d. Atomicity guarantees isolation from concurrent processes.
    True. The instructions part of an operation that is executed atomically cannot be interleaved with other instructions from a different process.

    e. A process that is waiting for access to a critical section protected by semaphores does not consume processor time.
    True, a semaphore is not implemented as busy-waiting. [Some students noted that depending on implementation, little busy-waiting might happen as a way to optimize execution – this explanation was given full credit.]

    f. In a multiprogramming system multiple processes exist concurrently in main memory.
    True, by the very definition of multiprogramming.

    g. Hardware support is needed to synchronize more than two processes.
    True. With only software support we cannot guarantee mutual exclusion. Semaphores are software tools but their implementation relies on hardware support (e.g., interrupts – by disabling it).

    h. Average throughput would be a good metric to use to compare the performance of two scheduling algorithms on an interactive system.
    False. Response time is what matters for interactive systems.

    i. If a multi-threaded program runs correctly in all cases on a single time-sliced processor, then it will run correctly if each thread is run on a separate processor in a shared-memory multiprocessor.
    True. Where the threads are running matters only in terms of their access to shared memory. If that works well in all cases on a shared processor, then nothing will change when threads are run on multiple processors.

j. Memory address translation is useful only if the total size of virtual memory (summed over all processes) needs to be larger than physical memory.
False, as discussed in detail in the Before VM paper.

k. If two executions of a race-free program on the same input perform the same sequence of acquire and release events in the same order, then the executions perform the same sequence of accesses to shared variables, and both executions produce the same final result.
False. You can have different results as per example worked in class (example worked in class should have been suggested). The original solution included the answer from question i).

2. **CPU Scheduling** (12 points):
   a. Describe how the lottery scheduling algorithm could be made to approximate a CPU scheduling algorithm that always runs the job that hasn't run in the longest time.

      Have the number of tickets accumulate over time. Thus, the longer a process waits to be scheduled, the more tickets it accumulates.

   b. In a modern computer system with a local page replacement algorithm and a multilevel feedback queue CPU scheduler there are two totally compute-bound jobs: JobA and JobB. JobA fits comfortably in memory while JobB suffers on average a page fault every 100 instructions.
      i. Which job (JobA or JobB) will be given the higher priority? (Explain why)
      Job B, because the page faults mean I/O operations, and the frequency of page faults (every 100 instructions) interrupts the process before it finishes its time quantum.

      ii. Which job (JobA or JobB) will get the largest share of the CPU? (Explain why)
      On one hand, Job A will have longer time quanta as being lower in the priority. On the other hand, Job B will have priority over Job A. So a correct answer should mention this situation – the answer depends on how long A's quantum is and how long B's I/O operation takes.

3. **Deadlock** (8 points): Which of the following will NOT guarantee that deadlock is avoided? Please circle all that apply and briefly explain your answer.
   a. Acquire all resources (locks) all at once, atomically.
   This eliminates the hold-and-wait condition.
   b. **Use locks sparingly**
   All 4 conditions for deadlock still possible.
   c. Acquire resources (locks) in a fixed order
   This makes circular wait impossible.
   d. Be willing to release a held lock if another lock you want is held, and then try

the whole thing over again
This eliminates both hold-and-wait and no-preemption conditions.

4. **RAID** (10 points): We have the following Broken RAID system (BRAID). BRAID sometimes doesn't fill in the redundant parts of the disk correctly. Circle which parity/redundancy bits BRAID got wrong below, or indicate that there is no problem:

   a. BRAID Level 4: **Whichever the parity disk is, bits 1, 2, and 4 are incorrect.**

```
Disk 0     Disk 1     Disk 2     Disk 3
0000       0111       0101       1111
```

   b. BRAID Level 5: Whichever the parity disk is, bits 3 and 4 are incorrect.

```
Disk 0     Disk 1     Disk 2     Disk 3
0000       0101       0001       0111
```

   c. BRAID Level 1 (Mirroring): Assuming Disks 0 and 1 are mirroring, and 2 and 3 are mirroring:

```
Disk 0     Disk 1     Disk 2     Disk 3
0000       0011       1101       1111
```

   d. BRAID Level 0: No parity/mirroring, thus if there is any mistake it is impossible to tell.

```
Disk 0     Disk 1     Disk 2     Disk 3
0000       0111       0101       1111
```

5. **Dining Philosophers** (8 points): Consider the procedure `put_forks` in the solution for dining philosophers discussed in class (and provided below). Suppose that the variable `state[i]` was set to `THINKING` after the two calls to test, rather than before. How would this change affect the solution for the case of 3 philosophers? For 100 philosophers?

```
#define N               5               /* number of philosophers */
#define LEFT            (i+N−1)%N       /* number of i's left neighbor */
#define RIGHT           (i+1)%N         /* number of i's right neighbor */
#define THINKING        0               /* philosopher is thinking */
#define HUNGRY          1               /* philosopher is trying to get forks */
#define EATING          2               /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N−1 */
{
     while (TRUE) {                     /* repeat forever */
          think( );                     /* philosopher is thinking */
          take_forks(i);                /* acquire two forks or block */
          eat( );                       /* yum-yum, spaghetti */
          put_forks(i);                 /* put both forks back on table */
     }
}

void take_forks(int i)                /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                    /* enter critical region */
     state[i] = HUNGRY;               /* record fact that philosopher i is hungry */
     test(i);                         /* try to acquire 2 forks */
     up(&mutex);                      /* exit critical region */
     down(&s[i]);                     /* block if forks were not acquired */
}

void put_forks(i)                     /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                    /* enter critical region */
     state[i] = THINKING;             /* philosopher has finished eating */
     test(LEFT);                      /* see if left neighbor can now eat */
     test(RIGHT);                     /* see if right neighbor can now eat */
     up(&mutex);                      /* exit critical region */
}

void test(i)                          /* i: philosopher number, from 0 to N−1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

`put_fork()` never wakes up the neighbors. Thus, if a philosopher gets blocked, it will never wake up. For 3 philosophers, only the first who gets to eat has a chance to ever eat, the others are blocked forever. For 100 philosophers, at most 50 (every other philosopher) have a chance to ever eat.

Incorrect answer: to call this situation deadlock.

6. **Memory** (12 points): Assume that you have a machine with a fixed amount of physical memory and demand-paged virtual memory system.
   a. Is it possible that doubling the page size can reduce the number of page faults? If so, describe in what condition. If not, describe why.

Yes. For example, if every address within a page is reference, by doubling the page size the page fault rate is halved, as for every page loaded in memory (thus, every page fault) we will get twice as many hits as before.

   b. Is it possible that halving the page size can reduce the number of page faults? If so, describe in what condition this is possible. If not, describe why.

Yes. For example, if memory references are made only to the first half of each (initial) page, then by halving the page size we fit double the number of initial pages in memory and only the relevant memory.

7. **Synchronization** (17 points)**:** Old Unix-like systems did not provide any primitives intended for synchronization of user processes, but they did provide pipes. A pipe is a channel for transmitting a stream of data, based on a fixed-sized buffer (often 4 KB). Reading an empty pipe blocks until data is written. Writing to a pipe whose buffer is full blocks until data is read.
   a. Explain how to implement a semaphore using a pipe. (Hint: take advantage of the pipe's blocking semantics.)

```
typedef struct {pipe p} sem;

void signal (sem s){
        write(s.p, 1); //write one by in the pipe
}

void wait(sem s){
        read(s.p, 1 byte); //read 1 byte from the pipe
        }
```

   b. A semaphore implemented as a pipe may deadlock in a situation where a conventionally implemented semaphore would not. Describe the situation.

   The deadlock can happen on signal(), when more than 4096 consecutive calls are made before a wait(). In that case, signal blocks, possibly leading to deadlock.

   Another solution possible: if the write is on larger than 1 byte in a), then signal blocks after fewer calls.