Operating Systems, PhD Qualifying Exam, Spring 2014

- This is 2-hour closed-book test. Calculator is allowed.

Name: ______________________________

1. [20 points] Multiprocessor programming in shared memory systems needs synchronization support for mutual exclusion to shared variables. Instead of a binary state for lock (i.e., available or held), *ticket lock* allows a lock to have multiple states. The basic idea is that when a thread wishes to acquire a lock, a globally shared variable used for lock state is first used to determine which thread's turn it is. Then, when the lock state represents a requesting thread's turn, the thread safely enters a critical section.

   Write `acquire()` and `release()` functions for ticket lock by using atomic Fetch-And-Increment instruction. The following code shows the data structure for lock, its initialization, and the C pseudocode for Fetch-And-Increment instruction.

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

int FetchAndIncrement(int *addr) {
    int old = *addr;
    *addr = old + 1;
    return old;
}
```

2. [30 points] Two important metrics for the CPU scheduler design are turnaround time and response time. The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system. The response time is defined as the time from when the job arrives in the system to the first time it is scheduled.

   (a) [10 points] When three jobs of length 5 arrive in the system at the same time, calculate the average turnaround time and average response time in Shortest Job First (SJF) scheduler and Round Robin (RR) scheduler with a time slice of 1.

   (b) [20 points] SJF reduces turnaround time by running shorter jobs first. Unfortunately, the OS does not generally know how long a job will run for, exactly the knowledge that SJF requires. RR reduces response time by running a job for a time slice. Unfortunately, RR increases turnaround time terribly. Design a scheduler that reduces both turnaround time and response time without knowing the job running time.

3. [30 points] Virtual memory system has a page size of 1024 bytes and physical memory size of 1 Mbytes. The following C code runs on the virtual memory system.

```
int x[1024*1024];
for (int i = 0; i < 1024*1024; i++) {
    x[i] = 0;
}
```

Assume the following:

- The program code is mapped to the first frame (i.e., frame number 0) in the memory.
- The size of `int` type element is 4 bytes. Array `x` begins at virtual address `0` and the virtual address of `x[i]` is `i` $\times$ 4.
- The only memory accesses are to the elements of the array. The loop index variable is stored in register.
- The pages are replaced in LRU order.

(a) [6 points] How many different pages will the program access?

(b) [6 points] How many frames does the 1 Mbytes memory have?

(c) [6 points] The access of the first element (`x[0]`) causes a page fault. After that, which array element will cause a page fault for its access?

(d) [6 points] How many page faults will occur during the entire code execution?

(e) [6 points] If the system has a four-entry fully-associative TLB, show the TLB content at the end of program execution.

4. [20 points] Dining philosopher's problem assumes that there are five philosophers spending their lives thinking and eating. The philosophers sit around a table. Between each pair of philosophers is a single chopstick (and thus, five total). The philosophers each have times when they think, and don't need any chopsticks, and times when they eat. In order to eat, a philosopher needs two chopsticks, both the one on her left and the one on her right. This problem can be implemented with five semaphores for five concurrent philosopher processes. A semaphore has `wait()` and `signal()` functions for synchronization. The following code for philosopher `p` (`p = 0..4`) has a serious problem.

Explain the problem and change the code to solve the problem.

```
semaphore chopsticks[5];

// philosopher p
while (1) {
    think();

    wait(chopsticks[p]);              // get left chopstick
    wait(chopsticks[(p + 1) % 5]);    // get right chopstick

    eat();

    signal(chopsticks[p]);            // put left chopstick
    signal(chopsticks[(p + 1) % 5]);  // put right chopstick
}
```