Operating Systems
CSCI-UA.0202 Spring 2013

Midterm Exam
ANSWERS

1. **True/False**. Circle the appropriate choice (there are no trick questions).

   (a) **T** The two primary purposes of an operating system are to manage the resources of the computer and to provide a convenient interface to the hardware for programmers.

   (b) **F** When the CPU is in kernel mode, the CPU is prevented from accessing areas of memory that are not owned by the current process.

   (c) **T** An interrupt table contains the addresses of the handlers for the various interrupts.

   (d) **F** If N batch jobs are scheduled using Shortest Job First scheduling, the total time it takes to execute the N processes is less than for any other batch schedule (assuming no two jobs take the same time).

   (e) **F** Each thread of a process has its own virtual address space.

   (f) **F** Every time a clock interrupt occurs, a context switch from one process to another is performed.

   (g) **T** In a multiprogrammed system using partitioning (i.e. each process occupies a contiguous block of memory), addresses can be relocated at run time using base registers.

   (h) **F** In a multiprogrammed system using partitioning, the Best Fit strategy (where a process is placed in the smallest hole in memory large enough for the process) provides the most effective use of memory.

   (i) **T** In a virtual memory system, a virtual page and a physical page frame must be the same size.

   (j) **F** In a virtual memory system, a virtual address and a physical address must be the same size.

2. (a) What is the difference between a trap and an interrupt?

   A trap is generated by the running program, e.g. by a system call, a memory violation, or a divide by zero. An interrupt may be caused by an external event – such as the disk controller signalling that a disk read request has completed.

   (b) What does the CPU hardware do when a trap or interrupt occurs? Describe the specific steps that the hardware performs (but just the hardware - not the OS or any process).

   When a trap or interrupt occurs, the CPU pushes the program counter and other registers (e.g. the stack pointer, status register, etc.) onto the stack, switches to kernel mode, and jumps to the address contained in the interrupt table entry corresponding to the interrupt that occurred.

   (c) Why is a special instruction, e.g. IRET, for returning from an interrupt handler required in order to resume executing a user-level process. That is, why can't a normal return instruction, e.g. RET, be used?

   When returning from an interrupt handler (to resume executing a process), the CPU must switch from kernel mode to user mode. A normal return instruction doesn't accomplish this.

(d) When a process executes a fork() system call, a duplicate process (i.e. the child process) is created. How does the code in the processes – since it is identical in both the parent and child processes – know which process is the parent and which is the child?

In the child process the fork() call returns 0, whereas in the parent process the fork() call returns the PID of the child process.

(e) Give a simple example of code that would operate differently in the parent and in the child.

```
if (fork() == 0)
    printf("I'm the child.\n");
else
    printf("I'm the parent.\n");
```

3. (a) Give a simple of example of a race condition by writing a code segment (in C) which, if it were executed by two processes executed at roughly the same time, could cause a race condition. Explain what bug could arise from the race condition.

```
int queue[SIZE], i = 0;  //assume these are shared between processes

void insert(int x)
{
  queue[i] = x;
  i++;
}
```

Since a clock interrupt (and a context switch) could occur between the assignment to queue[i] and the incrementing of i, after one process has inserted a value into the i'th element of the queue, another process could take over (before i is incremented) and insert a different value into the same element of the queue, overwriting the first value.

(b) Add some additional code, in conventional C (i.e. no semaphores or TSL), so that the race condition bug no longer exists.

Strict alternation:

```
int turn = 0;  //assume this is shared

//Process 0
void insert(int x)
{
  while (turn != 0) ;   //just loop
  queue[i] = x;
  i++;
  turn = 1;
}

//Process 1
void insert(int x)
{
  while (turn != 1 ;   //just loop
  queue[i] = x;
  i++;
  turn = 0;
}
```

(c) What is the disadvantage of the above solution compared to using TSL (test-and-set-lock)?

Because it requires the two processes to take turns, if one process is substantially slower than the other (e.g does much more work in between insert operations), the faster process will waste substantial time waiting for its turn.

(d) Suppose C provided a procedure `tsl(x)` that atomically returns the value of `x` and sets `x` to 1. Modify your original code (from part (a)) to use `tsl` to fix the race condition.

```
int lock = 0;  //assume this is shared. zero means unlocked.

void insert(int x)
{
  while (tsl(lock) != 0) ;  //just loop
  queue[i] = x;
  i++;
  lock = 0;
}
```

(e) What is the disadvantage of your solution using `tsl` compared to using semaphores?

The above use of tsl is an example of busy waiting, where the process that is locked out of the critical section wastes CPU time looping until another process releases the lock (i.e. sets the lock to zero). Using semaphores, there is no busy waiting - a process that performs a down operation on a semaphore whose value is zero is blocked. That blocked process will not resume executing until after another process performs an up operation on the semaphore, avoid the wasteful looping seen above.

4. (a) Describe the lottery scheduling algorithm.

In lottery scheduling, higher priority processes are assigned a greater number of "lottery tickets" than lower priority processes. When the scheduler needs to pick a process to run, it draws a lottery ticket and the process with that ticket gets to run. Thus, a higher priority process will have a statistically greater chance of being chosen to run than a lower priorty process.

(b) What is the advantage of using lottery scheduling over strict priority scheduling (where the highest priority ready process is the next process chosen to run)?

In strict priority scheduling, if there is a constant stream of ready high priority processes, low priority processes may be "starved" – i.e never get to run because a high-priority is always available to be chosen. This is extremely unlikely to happen in lottery scheduling because even a process with few lottery tickets will, with statistical certainty, eventually be chosen to run.

(c) Given two processes in the READY state, one that is CPU-bound and one that is I/O-bound, which process should be given a higher priorty for running next (all other things being equal)? Justify your answer.

The I/O-bound process should be given a higher priority. Since it will run for a short time and then issue an I/O request, the CPU-bound process will be able to then run at the same time that the I/O device is performing the I/O – maximizing the use of system resources. If the CPU-bound process were given a higher priorty, the I/O process would simply wait, without being able to start the I/O operation. When it ultimately is given the CPU and makes its I/O request, there is a greater likelihood of the CPU sitting idle while the I/O-bound process waits (because the CPU-bound process has already run).

(d) In the first programming assignment, you may have noticed that some processes were able to run for more than the quantum (which was 40ms) without being put back on the ready queue. Under what circumstances would this occur?

Suppose Process A blocks in between clock interrupts, e.g. due to an I/O request, and that Process B takes over the CPU at that point. Thus, Process B will not have started at a clock interrupt. The OS only checks if a process has used up its quantum when a clock interrupt occurs, so by the time the OS sees that Process B has used up its quantum, Process B will already have used up more than the fixed quantum time. For example, suppose that a clock interrupt happens every 10ms and that the quantum is 40ms (as in the assignment). Suppose that Process B starts running at time 5. The OS will check at time 10, 20, 30, 40, and 50 to see if Process B has used up its quantum. It is only at time 50, after Process B has run for 45ms, that the OS will see that Process B has used up its quantum.

5. Suppose you have a virtual memory system where addresses are 22 bits and the page size is 4096 (i.e. 2^{12}) bytes.

   (a) How many bits of a virtual address are used to determine the virtual page number and how many bits are used to determine the offset?

   Since the page size is 2^{12} bytes, the rightmost 12 bits of the virtual address will be used for the offset within a page. The remaining 10 leftmost bits are used to determine the virtual page number.

   (b) How many elements would a page table need to have?

   Since 10 bits are used to specify the virtual page number, there are $2^{10} = 1024$ virtual pages, which is how many elements the page table would need to have. Another way to arrive at this answer is: since the virtual address space is 2^{22} bytes (because addresses are 22 bits) and there are 2^{12} bytes per page, there are $2^{22}/2^{12} = 2^{10} = 1024$ pages in the virtual address space.

   (c) Suppose, during the execution of a process, the MMU performs the following virtual address to physical address translations:

```
0000000101101011010111 → 0000010110101011010111
0000011110000010110100 → 0010000001000010110100
0001010001010100010010 → 0000000101010100010010
0000000101000000010010 → 0000010110000000010010
```

How many different elements of the page table are accessed by the MMU to produce the above translations? Which elements of the page table are they and what page frame numbers do these elements contain?

If we separate the the leftmost 10 bits of each address that identify the page number from the 12 rightmost bits identifying the offset in the above translation, as follows,

```
0000000101 101011010111 → 0000010110 101011010111
0000011110 000010110100 → 0010000001 000010110100
0001010001 010100010010 → 0000000101 010100010010
0000000101 000000010010 → 0000010110 000000010010
```

we see that there were only three distinct pages accessed among the four virtual addresses above (the virtual page numbers in the first and fourth addresses, above, are the same). Thus, the page table must have elements as follows:

PageTable[0000000101] contains 0000010110
PageTable[0000011110] contains 0010000001
PageTable[0001010001] contains 0000000101

(d) Briefly describe the sequence of events that occur when a process tries to access a location in its virtual address space that is not currently in RAM. Start with the initial attempt to access the location and end with the location being accessed successfully.

When a process attempts to issue the virtual address for a location that is not currently in RAM, the following events occur (note that this ignores the TLB, which you were not expected to know about):

- **The MMU will access the page table entry for the specified virtual page and see that the present/absent bit indicates "absent", and thus generate a page fault trap.**
- **The handler in the OS for page faults will block the current process, issue a disk read request for the virtual page, and invoke the scheduler to choose another process to run.**
- **When the disk read request has completed and the requested page is in RAM, the disk controller will issue a disk interrupt.**
- **The OS (via the disk interrupt handler) will update the page table entry for the virtual page that was just loaded into RAM, and put the blocked process on the ready queue.**
- **When the schedule chooses the process (which originally caused the page fault) to run, the process re-attempts to issue the virtual address, which will now succeed because the corresponding virtual page is in RAM.**