CS 2200 Midterm Solutions, Fall 1999
Part A (Closed Book and Notes)
(30 Points, 20 Minutes)

1. I was expecting all of you to answer (f) of course! ☺

2. Both of them result in memory space that is available but not utilizable to satisfy a memory allocation request.
   Internal: A situation that occurs when memory management uses partitions (or pages) of predetermined sizes resulting in over-allocation than what is needed by the memory allocation request.
   External: A situation that occurs wherein none of the available free memory partitions can satisfy the memory allocation request, despite the fact that the cumulative free memory available is greater than the size of the memory allocation request.

3. One possible set of actions:
   - Stop issuing new instructions into the pipeline
   - Allow instructions in the pipeline to complete
   - Record PC value to return to for program resumption
   - Send interrupt acknowledge
   - Receive interrupt vector
   - Use vector to locate and load new PC corresponding to the interrupt handler
   - Resume pipeline processing by starting to fetch from this new stream of instructions

4. A possible scheme
   - Keep a "low water mark" (L) and a "high water mark" (H) for page fault rates that are deemed "acceptable"
   - If (pf-rate < L) swap-in new jobs (i.e. increase degree of multiprogramming)
   - If (pf-rate >H) swap-out some jobs (decrease degree of multiprogramming)

5. **False**. Not *always* for a variety of reasons:
   - The page table may be in special processor registers (although highly unlikely)
   - Such architectures invariably use a TLB to cache recent translations, therefore, the need to access the page-table (assuming it to be in memory) is *only* upon misses in TLB

6. **False.** Not *always.*
   - Reducing the CPI may increase the clock cycle time (C)
   - The expression for Execution time is N * CPI * C; therefore, if the impact of C going up is more dominant than CPI going down then it will degrade performance.

7. **False**. Not all the registers in the processor need be saved and restored during a call/return sequence. By observing a sensible procedure calling convention in the compiler, the overhead for register save/restore can be made largely independent of the number of registers in the processor architecture.

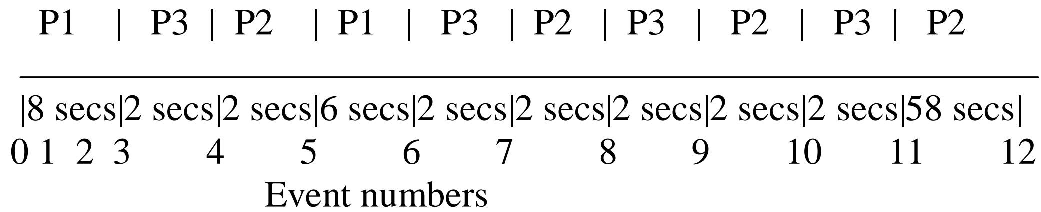
CS 2200 Midterm Solution, Fall 1999
Part B (Open Book and Notes)
(70 Points, 60 Minutes)

1. Execution time = N * CPI * C
   Execution time old (E-old) = (0.3 * 2 + 0.1 * 5 + 0.6 *1) * C = 1.7 C
   Execution time new (E-new) = (0.3 * 2 + 0.1 * 3 + 0.6 * 1) * (1+x) C,
   where x is the increase in clock cycle time.
   For the architectural change to be acceptable,
   E-new <= E-old
   1.5 (1+x) C <= 1.7 C
   Solving, increase in clock cycle time should be no more than 13%.

2. Effective memory access time
   = (hit-time for L1) +
   (miss-rate for L1 * hit time for L2) +
   (miss-rate for L1 * miss-rate for L2 * memory access time)
   = 2 ns + 0.02 * 10 ns + 0.02 * x * 60 ns,
   where x is the miss- rate for L2 cache
   The effective memory access time should be less than 3 ns.
   Solving, x is less than or equal to 66%.

   So the hit rate for L2 should be at least 34%.

3. Here is a GANTT chart



Events:

0. P1 arrives and is scheduled
1. P2 arrives at time = 2 and waits
2. P3 arrives at time = 4 and waits
3. P1 done with CPU burst now on to I/0; P2 and P3 ready to be scheduled, but P3 is higher priority and gets the CPU
4. P3 moves to do I/O; P2 gets to run on the CPU
5. P1 and P3 done with I/O; P1 is higher priority than P3 and P2; P1 pre-empts P2 and is scheduled on the CPU
6. P1 completes and exits the system; P3 is scheduled on the CPU
7. P3 goes to do I/O; P2 scheduled on the CPU
8. P3 is done with I/O pre-empts P2
9. P3 goes to do I/O; P2 is scheduled on the CPU
10. P3 is done with I/O pre-empts P2
11. P3 completes and exits the system; P2 gets the CPU
12. P2 completes and exits the system

Turnaround time:

P1: 18 secs
P2: 86-2 = 84 secs
P3: 28-4 = 24 secs

Average waiting time = (0 for P1 + 20 for P2 + 10 for P3)/3 = 10 secs

4. Basic scheme is to use an approximate LRU using the ref and dirty bits in the TLB. Upon a TLB miss software does the translation. It evicts a TLB entry, and stores the new translation in the TLB entry. Upon a page fault, a victim physical frame is selected.

```
/* Keep a counter per  physical page frame; Initialize the counters to zero;
 */
typedef struct page_frame_t {
     int pf-counter;  /* initialized to zero */
     int pf-dirty;
} page_frame;

page_frame PM[SIZEOF_PHYSICAL_MEM];
Approximate LRU maintenance algorithm:
     { /* sample and save ref bits from TLB */
             for j = 0 to size-of-TLB {
                     read TLB(j);
                     if (TLB[j].valid) {
                             shift TLB[j].ref-bit
                             into MSB of PM[TLB[j].PPN].pf-counter;
                     }
                     clear_refbit_TLB[j];
             }
             for all PPN not currently in TLB {
                     shift 0 into MSB of PM[PPN].pf-counter;
             }
     }
```

software TLB miss algorithm:

```
{
        /* upon miss select a victim from TLB; update the PM
         * structure;
         */
        do the address translation;
        select an entry to evict from TLB;  /* this will depend on the
                                             * organization
                                             */
        let j be the index;

        if TLB[j].valid {
                /* update the PM structure corresponding to this entry */
                PM[TLB[j].PPN].pf-dirty = TLB[j].dirty;
                shift TLB[j].ref-bit
                into MSB of PM[TLB[j].PPN].pf-counter;
        }
        update TLB[j] with new address translation;
}
```

Page replacement algorithm:

```
{ /*select a victim page */
        for j = 0 to SIZEOF-PHYSICAL-MEM {
                victim-PPN = PPN s.t.  PM[PPN].pf-counter is smallest;
        }
        /* victim page needs to written out if either its entry is currently
         * marked dirty in TLB or the PM structure indicates it is dirty
         */
        j = search-TLB(victim-PPN);
        if (j >= 0) {
                /* j  the TLB index of victim-PPN */
                if TLB[j].dirty {
                        write victim-PPN to disk;
                }
        }
        else {
                if (PM[victim-PPN].pf-dirty) {
                        write victim-PPN to disk;
                }
        }
        PM[PPN].pf-dirty = not-dirty;
        return(victim-PPN);
}
```

Search TLB for victim-PPN:

```
{
        for j = 0 to size-of-TLB {
                if (TLB[j].PPN == victim-PPN)
                        return(j);
        }
        if j == size-of-TLB
                return(-1);
}
```