CSE 451 Winter 2002 – Midterm 2

Name: _____

Each answer below is an actual answer submitted on this exam. I'm hoping that this will help you better gauge how much you need to write to receive full credit.

1. [10 points]
   Imagine you were writing the CPU scheduler for a system with a very odd characteristic: processes require either exactly 100msec. or 100 seconds of computing. All processes also do exactly two blocking IO events sometime during their first 100msec. of computing (no matter how much time they need total), but no IO or other blocking events after that. The IO events happen at unpredictable times during that first 100msec.

   Briefly discuss each of the following general approaches to scheduling on this system, assuming that the goal is to minimize process response times.

   (A) FCFS

   The first long process scheduled after its final I/O block will hold the system to the detriment of potentially many small I/O processes. This provides poor response time.

   (B) LCFS (Last-Come-First-Served)

   If 100 seconds come last, the response time is bad overall.

   (C) Round-Robin

   This would be a good choice, as the preemption would prevent 100 sec. blocks from clogging the system when 100 msec. blocks were trying to get through.

   (D) Multi-Level Feedback with no aging

   The best – since all IO happens during the first 100ms, it happens in the top queue, resulting in the best response time. Starvation is possible.

   (E) the Linux SCHED_OTHER policy

   Once the processes have performed IO they get a little bit of a boost but it goes away quickly so that this does not degenerate to part (D) – it ends up a lot like part (C) and should do fairly well.

2. [6 points]
   (A) What is rate-monotonic scheduling **and** when is it an appropriate choice?

Rate monotonic scheduling is when processes need the CPU periodically (such as a video player needs to put up frames 30x/sec). It is appropriate for real-time scheduling.
[RM is that plus the idea that the shorter the period the higher the priority.]

(B) In what important sense are Linux's "real time schedulers" not real-time schedulers at all?

They can't set a time deadline and be guaranteed to execute by that time.

(C) Give an example that illustrates the notion of "priority inversion"?

A high priority process is waiting on a low priority process; e.g., to give up a lock it holds.

3. [4 points]
Suppose you're using the Banker's Algorithm to avoid deadlock in a system that has multiple instances of at least some resources. Suppose at some point in time a new process, A, arrives, declares maximum resource requirements with the vector (m_0, m_1, …, m_R), and immediately requests exactly one unit of resource 0. ("Immediately" means that no other pertinent events have happened between the time A arrived and making its request.)

Assuming $m_0 > 0$, does the precise value of the maximum request vector (m_0, m_1, …, m_R) have any effect on whether or not the system will grant the request at the time it's made? If so, which values are important, and why. If not, why not?

Yes, the values of m_1,…, m_R may have an effect on whether the system grants the request. If the process could claim a resource held by another process, and that other process wants resource 0, we'll be in deadlock, so granting the request would lead to an unsafe state.

4. [4 points]
Suppose you have a multi-threaded application in which threads may execute the following procedure asynchronously. The procedure has a bug, in that it may lead to deadlock. **Fix it** so that it will never cause deadlock (under the assumption that the `mutex's` are used only by this one routine) **without affecting the locking granularity and without making unnecessary changes**.

```
mutex      locks[ASIZE];
int        A[ASIZE];

int  sub (int one, int two) {

     int temp;

     if (one > two) return sub(two,one);
```

```
        assert (one>=0 && one < ASIZE && two>=0 && two< ASIZE);

        mutex_lock(&locks[one]);

        mutex_lock(&locks[two]);

        temp = A[one];

        A[one] = A[one]+A[two];

        A[two] = A[two] + temp;

        mutex_unlock(&locks[one]);

        mutex_unlock(&locks[two]);

        return temp;

    }
```

[This is a truly beautiful answer – better than what I had thought of when I made up this question. I had in mind the more commonly given solution of comparing *one* and *two* and issuing the *mutex_lock*'s in sorted order. (Note: the order of the *mutex_unlock*'s is irrelevant.)]

5. [4 points]
Imagine that a contiguous memory allocator (e.g., `malloc()` or `new`) had an interface that allowed allocation only of blocks that were any multiple of 100 bytes in length.

(A) Would external fragmentation be an issue in this system? (**Briefly** explain.)

Yes, requests could be of varying sizes, so small spots could be left in between allocated blocks (after those small chunks were released).

(B) Would internal fragmentation be an issue in this system? (**Briefly** explain.)

Yes, because the allocator gives 100 bytes even if you only need 4 bytes of memory. Therefore, internal fragmentation exists.

6. [4 points]
Give **two** examples of useful functions that can be implemented in a system that provides address translation but not paging.

Two functions that are useful here are sharing segments between users and memory protection.

7. [4 points]
(A) What is the motivation for using page replacement policies that have the flavor of LRU replacement? (That is, what are they trying to achieve and under what

conditions do they achieve it?)

LRU tries to predict which page is less likely to be used in the future. It's based on the concept of locality, and is based on the assumption that the page which was used last will probably not be used again for the longest time.
[Taken literally, this is the opposite of LRU, but I assumed it was just sloppy wording. (Either that or I've transcribed it incorrectly…)]

(B) Why do systems implement approximations to LRU page replacement, rather than LRU itself?

It's expensive to keep track of the order of page accesses. It's easier to just mark a page as 'accessed.'
[The point here is that pure LRU requires overhead "per memory reference" whereas the approximations require overhead that is much less frequent.]

8. [4 points]
(A) How does virtual memory help protect one process from bugs in another process?

Each process has its own virtual address space, which maps to physical addresses. So it is not possible to accidentally read or write memory that another process is using.
[I assumed the idea of "distinct physical addresses."]

(B) How does virtual memory help protect a process from bugs in itself?

It provides protection bits so a process can't write over its own code, or other read-only structures.

9. [4 points]
(A) Why would a system that provides paging also implement swapping?

If the processes that are running start requesting lots of memory and the memory gets over allocated, the system will probably have to remove (or swap out) a process from the run queue in order to free memory and avoid thrashing.
[The one word "thrashing" would have received full credit.]

(B) Suppose you implemented a system with swapping but without paging. Would there be any advantage to then also implementing paging?

Yes, paging helps reduce external fragmentation.

10. [4 points]
(A) What is the point of "two-level paging"?

To decrease the total size of the page table because a lot of VM is never used between

the stack and the heap.

(B) Why is hierarchical paging not likely to be implemented in systems that have 64-bit virtual addresses?

There have to be many levels to achieve the same space effect than with 32 bits which in turn means many memory reads which cost too much time.

11. [2 points]
A "program profiler" is a tool that measures various performance characteristics about a single execution of a program. Profilers typically provide reports that indicate what fraction of CPU time was spent in each of the procedures of the program (during the particular execution that was measured). Obtaining the information needed to create such reports could be done by reading a high-resolution clock on each procedure entry (call to) and exit (return from), and accumulating the differences between those two times in per-procedure counters.
Mostly program profilers do **not** work this way, but use a simpler technique. What might they be doing? (Hint: the answer uses a program "map" that indicates the address range in which the code for each procedure is loaded, and is related to an aspect of the Linux SCHED_OTHER scheduler.)

At every time tick, the PC can be examined, compared to the program map and the indicated procedure scored as being executed at that point.

[For the purposes of grading, I interpreted "time tick" to mean a timer interrupt, rather than each clock cycle.]