Stanford University
Computer Science Department
CS 140 Midterm
Dawson Engler
Winter 2000

This is an open-book exam. You have 50 minutes to answer as many questions as possible. The number in parenthesis at the beginning of each question indicates the number of points given to the question. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

Question	Points	Score
1	24	
2-3	12	
4	8	
5	16	
total	60	

Stanford University Honor Code

In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam.

Name:

Signature:

1. Short-attention-span questions (24 points)
Answer each of the following questions and, in a sentence or two, say *why* your answer holds. (4 points each).

1. (4 points) Why does an atomic swap instruction (atomically) exchange data between a register and a memory address, rather than swapping between two registers? (Especially since on modern machines the latter would be faster.)

   Each thread has it's own private register set, so swapping between registers is invisible to other threads and, thus, fairly pointless.

2. (4 points) On a system with a TLB, what does the OS have to do after revoking a page from a process?

   It has to delete any TLB entry that translates to that page. This can be done by flushing the entire TLB or, on architectures that support more precise modifications, nuking exactly those entries.

3. (4 points) Assume we modify a Mesa-style monitor system to, when a condition is signalled, putting the woken thread on the front of the run queue so that the scheduler will pick it to run when the signalling thread either exits or blocks. Do we still need to use "while" loops instead of "if" statements to check conditions?

   Yes. The producer could have signalled a condition as true, and then violated it. Or the producer could signal multiple conditions, waking up multiple threads that can then interfer with each other.

4. (4 points) Assume you have a system with several heavily-paging processes. Would you expect the system to be more efficient with a Unix Multi-level scheduler or strict round robin?

   A heavily-paging process is equivalent to an I/O intensive process. So, multi-level should be more efficient, since it gives better I/O utilization.

5. (4 points) What scheduling algorithm can give non-linear improvements in completion time when you run the same job mix on an identical machine that has a faster CPU?

   Round robin. There is a big performance difference between completing in your first quantum and completing later. If you keep the same time slice quantum then, as the CPU gets faster, more work gets done, and it will be more frequently the case that an application completes in its first quantum.

6. (4 points) In the presence of I/O, can multilevel give a faster completion time than a shortest-job-first scheduler that preempts blocks processes?

   Yes. ML is explicitly aimed at keeping I/O busy. SJF will just do what it says, ignoring I/O other than switching. Example: If there is a long job than does frequent I/O and shorter jobs that do none. short jobs will always run.

2. Link later, pay now. (6 points)
When using dynamic shared libraries we do not extract just the functions we need from the library, but (conceptually) map the entire library into the process's address space. In the context of how we constructed our libraries, give two reasons why we do this, even though the alternative would result in a more compact virtual address space.

1. *Internal calls (calls between functions within the library) will no longer work.*
2. *Since we compact the library, we will not be able to share code pages with other processes.*

3. Bad programs get shot (6 points)
Assume we have a system with multiple separate monitors that can call each other. Explain what general problem this can cause and give a simple, clear(!) example of it. *Deadlock can occur. For example: we have two threads, T1 and T2. Suppose we have two monitors, M1 and M2. In procedure P1 (part of M1), there is a call to procedure P2 in M2. Likewise, in procedure P3 in M2, there's a call to procedure P4 in M1. If T1 is in procedure P1, and T2 is in procedure P3, then T1 owns M1, T2 owns M2. T1 will block when the call to P2 is made, and T2 will block when the call to P4 is made.*

4. Stupid data structure tricks (8 points)
For what type of address space layouts would the following page table structures have much better space and/or time performance than a linear page table (i.e., a single-level array)?

1. (2 points) A dense vector where lookup happens by binary search.
   Sparse address space.

2. (2 points) A hash table.
   (Large) sparse address space.

3. (2 points) Using a page table + segmentation.
   Sparse address space with small segments.

4. (2 points) Give one case where the linear page table will perform better than all of the previous structures. *A completely (or close) full address space.*

5. Race condition fun (16 points)

For each of the following situations, indicate whether they are a possible race or not and give a sentence or two intuition for your answer. Additionally, for races, please give a short, reasonable example of code that illustrates it. To simplify the problem, define a race condition as threads giving results different then if the memory they read and wrote was contained in a single critical section.
This question was exceptional in its poor wording. The spirit of the question is: will the final result of the given sequence of reads and writes be different than any possible execution sequence in the presence of locks.

1. (2 points) Thread T_1 writes memory location m; thread T_2 writes location m.
   This is fine, since it gives results obtainable if m was wrapped in a critical section. E.g.,:

```
  T1              T2
lock(l);        lock(l);
m = 5;           m = 6;
unlock(l);      unlock(l);
```

2. (3 points) T_1 reads m; T_2 writes m; T_1 writes m.
   We lose T_2's update; which cannot happen if accesses to m are wrapped in a critical section.

3. (6 points) T_1 reads and writes m, T_2 reads m. T_1 reads and writes m', T_2 reads and writes m'.

   We threw this question out, since it was much harder than anticpated.

4. (5 points) T_1 reads m, T_2 reads and writes m. T_1 reads and writes m', T_2 reads and writes m'.
   We threw this question out, since it was much harder than anticpated.