Stanford University
Computer Science Department
CS 140 Midterm
Dawson Engler
Winter 2001

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.

NOTE: We will take off points if a correct answer also includes incorrect or irrelevant information. (I.e., don't put in everything you know in hopes of saying the correct buzz word.)

Question	Points	Score
1	25	
2	10	
3	10	
4	15	
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 nor will I assist someone else cheating.

Name and Stanford ID:

Signature:

1. Short-attention-span questions (25 points)
Answer each of the following questions and, in a sentence or two, say *why* your answer holds. (5 points each).

1. When would you expect base and bounds to have better performance than page-based virtual memory?

   A large process that fits in main memory and randomly references its address space. Page based VM will suffer many TLB faults.

2. Why could two independent processes P1 and P2 that run correctly with paging possibly deadlock when paging is disabled?

   Typical hold and wait problem. Assume both P1 and P2 initially use a MB of memory, need 8MB in total to complete, but the machine only has 8MB in total. They will both be able to start, but won't be able to finish and will stay, tying up memory.

3. If we link libraries dynamically instead of statically, how can we still warn about unresolved symbols at static link time?

   Search the symbol table of all dynamic libraries at static link time.

4. In class, we discussed how to combine segmentation and page-based virtual memory. Why would you expect that reversing the way we implemented this (so that segments map virtual address ranges and page tables map segment tables) to not nearly be as useful?

   this loses use the advantage of page tables (large virtual address spaces patched together from non-contiguous memory), while keeping the overhead of the two level scheme. In addition, segment tables are small, so building them from pages doesn't buy you anything (and may lose because of internal frag).

2. Big time monitor fun (10 points)
Monitor M1 has called monitor M2, which has decided to wait on a condition.

1. Assume that M2 has no calls into M1 or any other monitor. What problem could happen if we release M2's lock, but not M1's? (Also, please give a quick example.)

   Deadlock. Example: T1 did the nested call from M1 into M2 and is now suspended waiting for a signal; T2 will issue the signal, but only after first calling into M1:

   ```
     T2
   M1.foo();
   M2.some_method_that_will_signal_T1();
   ```

   because we hold M1's lock, T2 won't be able to finish the M1 call and we will deadlock.

   Points:

   - *You get two points for mentioning deadlock.*
   - *You get another point if you mention that it's M1's lock that causing the deadlock.*
   - *If the solution is correct and an example is given you get full credit, however one point may be deducted if the explanation is too vague, or there is extra, incorrect information.*
   - *You get only one point if your explanation resembles what deadlock is, or you mention deadlock but has other incorrect information (such as mentioning deadlock and race-conditions)*

2. What problem could happen if we release both M1's lock and M2's lock? (Also, please give a quick example.)

   We've violated mutual exclusion, since we allow another thread to run. This could violate any conditions M1 was depending on.

   - *-1 if you mentioned that both M1 and M2's data can be corrupted.*
   - *-2 if there is no example, or you mentioned M2's data can be corrupted without mentioning M1's data may be corrupted.*
   - *Further points may be deducted if there is extra, incorrect information.*
   - *Also accepted one solution of livelock if you explicitly gave the scenario of how the locks may be required out-of-order. Any other answer with livelock without this scenario is rejected.*

3. We have a winner (10 points)
Assume we change lottery scheduling to be deterministic. As with normal lottery scheduling we want threads to receive the CPU in proportion to the number of tickets they have. For example, if a thread T1 has 10 tickets and T2 has 1, we would want T1 to get 10 times as much of the CPU as T2.

1. **(5 points)** We track two things for each thread: (1) the number of cycles it has consumed and (2) the number of tickets it has. How can we use these to calculate which thread to run? Please give both a formula and a description of what your formula is trying to accomplish.

   calculate cycles/tickets for all threads and take the minimum, (or invert and take max, though that will lead to more roundoff error.

   Points:

   - *5 points: correct ratio, correctly specificied to choose largest or smallest ratio for all threads in system.*
   - *4 points: mixed up choice of smallest/largest*
   - *3 points: used numCycles/numTickets in some sort of ratio, but didn't explain its use.*
   - *2 points: some sort of ratio, not correct numbers*
   - *1 points: some other incorrect algorithm (usually included*

2. **(5 points)** Assume T1 and T2 have been running on the system for a while. If a new thread T3 enters the system, why would it be better to set the cycles it has consumed to the minimum of those used by T1 and T2 rather than initializing it to 0?

   T3 will monopolize the CPU until it "catches up" with T1 and T2. Points:

   - *5 points: explicitly stated that T3 would completely monopolize the CPU until its numCycles/numTickets caught up with the other threads*
   - *4 points: said T3 would be favored to run, but didn't state it would starve threads T1 and T2.*
   - *3 points: T3 would somehow benefit, but vague.*
   - *2 points: Said T3 would starve (opposite is true)*
   - *1 points: Something related to thread choice fairness*

4. What happened? (15 points)
Assume you have implemented a simple fork/join interface:

```
int fork(void (*fp)(int *), int *data);
void join(void);
```

Where fork will create a thread that runs the procedure pointed to by fp with data as its sole argument and join will block until *all* threads forked by the current thread have exited. You plan use the world-famous fibonacci routine as your killer app. Taking the sequential version:

```
int fib(int n) {
        if(n <= 1)
                return 1;
        else
                return fib(n - 1) + fib(n - 2);
}
```

You change it to the (clumsier) fork/join based version:

```
int fib_fork(int *n) {
        if(*n <= 1)
                *n = 1;
        else
                int n1 = n - 1, n2 = n - 2;
                fork(fib_fork, &n1);
                fork(fib_fork, &n2);
                join(); /* join both threads */
                *n = n1 + n2;
        }
}
```

1. **(5 points)** Using your knowlege of the overheads of fork and join, when (if ever) would you expect the fork-based fib routine to be faster than the sequential version? (While you should roughly relate your answer to n and the number of processors available a formula is not necessary.)

   fork/join will have fairly high overhead compared to a fibonnacci invokation, so we'll need pretty high values of n, and a fair number of multiprocessors. -2 to -3 if you mention the right intuition, but do not relate to n and number of processors.

2. **(5 points)** Assume you run forked threads in seperate process address spaces. What problem will this cause for our use of n1 and n2?

They will be on the stack; so the stack addresses we pass in will mean different things in other address spaces. You'll either get a segfault or corrupt random parts of memory.

3. **(5 points)** Assume you are on a uniprocessor. Since there is no parallelism you decide to reduce fork/join overhead by changing fork to immediately call its given procedure and join to do nothing:

```
int fork(void (*fp)(), int *x) { fp(x); }
void join() { }
```

While this hack works great for fib_fork why might you expect it to cause other more complex programs to hang?

Because we run code sequentially, we won't be able to run threads out of order: if there are circular depenedencies, can now deadlock. -2 if you gave an example but did not explicitly talk about circularity.