Stanford University
Computer Science Department
CS 140 Midterm Exam
Dawson Engler
Winter 1999

Name: ________________________________

Please initial the bottom left corner of each page.

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, and about how many minutes you should spend on your answer. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

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

Signature: ____

Problem	**Points**	**Score**
1-7	14	
8	9	
9	9	
10	9	
11	9	
12	20	
Total	70	

True/False questions (14 points total)
(2 points for correct answer, 0 for blank, -1 for incorrect)
(Note: because of ambiguities, everyone gets full credit for questions 3, 6, 7)

1). True: Monitors can deadlock.

> *If monitor code has a circular dependency, it can deadlock. For example, for the circular buffer code we looked at in class, if a thread performing a "get" busy waited when it encountered an empty buffer (rather than blocking), no "put" thread could ever enter the monitor, causing deadlock.*

2). False: Increasing the page size will likely decrease working set size (in bytes).

> *Decreasing page size decreases internal fragmentation, which most likely will decrease the size (in bytes) of a program's working set.*
>
> *We also accepted "true" if in question 11b you defined working set size as the number of pages in the working set.*

3). There was no question 3. You get 2 free points.

4). False: Increasing the page size will likely decrease the chance that page revocation requires a disk write.

> *As the page size increases, the likelihood that a page is dirty increases (consider a system with a single page equal to the size of main memory). When a dirty page is evicted, it must first be written to disk.*

5). False: As the ratio of time slice to job length decreases, round-robin scheduling becomes equivalent to first-come-first-served.

> *As the time-slice length goes to infinity (the ratio increases), each job completes in a single time slice (the first time it runs), which is equivalent to first-come-first-served. As the time-slice length goes to zero (the ratio decreases) round-robin looks increasingly different than FCFS.*

6). And 7). We threw away these 2 questions since they were too ambiguous. You get 4 free points.

Short Answer Questions
(Each answer should at most be a few sentences long.)

8). You conceive of a get-rich scheme of retrofitting virtual memory onto old government computers. You decide to base your virtual memory system on the MIPS architecture, which when a TLB fault occurs, records both the faulting program counter and faulting virtual address and then traps to the OS. The OS uses this information to handle the fault and jump back to the indicated instruction or to signal an exception. Unfortunately, the most prevalent government computer in use, the Obsolete-2000, supports the following instruction:

```
inc reg, const
```

which adds 4 to "reg" and then adds "const" to the memory location pointed to by "reg." Assuming that main memory is represented as a vector "mem", the C code for these semantics would be:

```
reg = reg + 4;
mem[reg] = mem[reg] + const;
```

[Part A. (5 points)] Explain why a program that executes this instruction may not work after you add virtual memory.

The instruction does a memory access as well as a register increment. By adding virtual memory, there is now the possibility that the referenced memory has been swapped out to disk. This will trigger a page fault to bring in that page. However the register has already been incremented because instructions are atomic. When the page has been brought in and the instruction is re-executed, the register will be incremented again. So when the instruction causes a page fault, the register will be incremented twice.

5 points if mentioned that instruction could cause a page fault, and that could cause the register to be incremented twice.
2-3 points if realized that instruction could cause a page fault.
2-3 points if realized that instruction would be re-executed after a page fault.
0 points if talked about how incrementing reg by 4 would give a bad address.

[Part B. (4 points)] Fortunately, you notice the Obsolete also supports the instruction:

```
ref reg, const
```

which simply touches the memory at address "reg + const." Here the equivalent C code would be:

```
(void)mem[reg+const];
```

In a few sentences, sketch how a program loader could use this instruction along with techniques similar to those used to implement software fault isolation to fix programs containing "inc" so that they will work with virtual memory. Assume a non-preemptive scheduler. *Using your knowledge of linkers, be sure to state what values may need to be updated after your modifications.*

Insert a "ref reg" instruction in front of every "inc reg, const" instruction. This way, if the memory access were to cause a page fault, the page would be brought in before the "inc reg, const" instruction. Thus the "inc reg, const" instruction is guaranteed to complete without a page fault. The double increment problem mentioned above can not occur.

Since an instruction has been inserted after every "inc reg, const" instruction, all branch offsets and jump targets must be recalculated.

4 points if inserted ref instruction in front of every inc instruction. And said that this would prevent page faults from occurring in the inc instruction. And if said branch offsets needed to be recalculated.
2 points if inserted ref instruction, but for wrong reason.
1 point if adjusted branch offsets, jump addresses

9). [Part A. (5 points)] Ignoring the overhead of context switching, will I/O utilization on a round-robin system increase or decrease as the time-slice length is increased? Please give the intuition behind your answer, but not an epic essay.

I/O utilization will decrease. CPU-bound processes will dominate the processor, and I/O-bound processes (which require very short CPU bursts) will execute quickly and move back to the I/O queues. This means that total I/O utilization will decrease.

5 points for saying that I/O utilization will decrease and giving a convincing answer why it will do so. 1-3 points for saying it will decrease and giving an unconvincing answer. 1 point for saying it will increase and giving an (incorrect) explanation that still shows some knowledge of the subject matter.

[Part B. (4 points)] Will some programs take longer to finish as the time-slice length *decreases* on the above system? Why or why not? (Again assume a zero-overhead context switch.)

Some programs will take longer to finish. In particular, programs that used to finish within one quantum (time slice) but now need to be switched out will spend time waiting on the queue, causing them to take longer to finish. Not all programs will take longer to finish. Take for example 2 programs, one needing 2 time slices and the other needing 9 *time slices. Assume that the smaller program gets first dibs on the processor. If the quantum we're using is 2 time slices, the first program will finish in 2 time slices, and the second in 11. If we decrease the time-slice length to 1 time slice, the first program will finish in 3 time slices, but the second will still finish in 11.*

4 points for saying that some programs will take longer to finish and identifying what exactly causes them to take longer (the wait time in the queue) or giving a concrete example backing up your assertion. 2 points for saying that some programs will take longer to finish but not giving a complete answer as to what causes them to wait longer, or just mentioning context switch overhead. 1 point for saying that some programs take longer to finish but not giving a reasonable explanation, or for saying something else but then showing that you understand something about scheduling.

10). Your friend, Ben Bitdiddle, decides to emulate your success by selling the government the "No Race Condition" scheduler, as a way to eliminate their concurrency bugs. His idea is that since race conditions are caused by concurrent access to shared state that he will simply disable time-sharing and run programs, first come first served, until they terminate.

[Part A. (5 points)] Describe a program that works fine with round-robin scheduling, but no longer works under this "improved" scheduler (and why it does so).

A producer/consumer program or any program that depends on data from another program
Also accepted: Infinite loop
We took off for having the right answer but giving wrong reasons or giving wrong additional info

[Part B. (4 points)] Write a *short, self-contained* C program that is harmless under timesharing but will cause Ben's system to freeze.

```
main()
{
   while(1) ;
}
```

Full credit: waiting on a condition or a semaphore
Partial credit:
1. programs that were not harmless under timesharing: thrashing, infinite forking of children
2. using more than one process or multiple threads
3. writing pseudocode instead of a real C program

11). After switching from static linking to dynamic linking with static shared libraries, you notice that the working set size (in bytes) for most programs increases.
[Part A. (5 points)] What happened?

[Part B. (4 points)] Will making the page size smaller decrease or increase the number of bytes in the working set and why?

A less-short-answer Question

[12. (20 points)] The following C code implements a simple LIFO queue data structure.

```
struct elem {
        struct elem *next;
        T data;
};

struct queue {
        struct elem*head;
};

/* initialize queue: sequential access, does not need to be protected. */
void init(struct queue *q) {
        q->head = NULL;
}

/* insert e at the front of q */
void enqueue(struct queue *q, struct elem *e) {
        e->next = q->head;
        q->head = e;
}

/* return first element in queue */
struct elem *dequeue(struct queue *q) {
        struct elem *e;

        e = q->head;
        if(e)
                q->head = e->next;
        return e;
}
```

Your machine supports an atomic conditional assignment instruction, which the following procedure encapsulates:

```
        int cassign(T *addr,  T cur,  T val);
```

swap takes a pointer to any type T, and two values of type T and has the following semantics:

```
/* all code executed as an indivisible unit */
int cassign(T *addr, T cur, T val) {
        if(*addr != cur)
                return 0;
        else {
                *addr = val;
                return 1;
        }
}
```

Rewrite the dequeue and enqueue routines to use the conditional assignment procedure (rather than locks) to avoid race conditions in the presence of multiple threads. (You may rip out the previous page to make doing so easier.) Do not introduce any further variables or data structures (or alter the existing ones).

```
/* insert e at the front of q */
void enqueue(struct queue *q, struct elem *e) {
        do {
                e->next = q->head;
        } while(!swap(&q->head, e->next, e));
}

/* return first element in queue */
struct elem *dequeue(struct queue *q) {
        struct elem *e;

        do {
                e = q->head;
        } while(e && !swap(&q->head, e, e->next));

        return e;
}
```

The while loops are required because another thread could have altered q->head between the time it was read and swap tried to modify it. If neither routine used a loop, 7 points were taken off, if only one didn't use a loop, 4 points were taken off.

If the code no longer worked sequentially anywhere from 2 to 10 points were taken off, depending on how bad the situation was.

If the swap in enqueue was swap(&q->head, q->head, e), 4 points were taken off. This code has a race condition in that a thread could alter q->head between the assignment to e->next and the swap. If dequeue also had the equivalent error ("swap(&q->head, q->head, e->next)") a total of 7 points were taken off.

Sending the wrong types to swap lost a point.

Not handling an empty queue in dequeue lost 2 to 4 points, depending on how wrong the code was.