

**CS 574**  
**Midterm Solutions**  
**October 10, 2005**

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write your social security number, but not your name, on each sheet of paper you turn in
- show your work whenever appropriate. There can be no partial credit unless I see how answers were arrived
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 10:20 to finish the exam. The questions are equally weighted.

1. Process memory regions are managed using two completely separate data structures (linked lists and red-black trees). What is the purpose of each of them?

*The linked list is useful for finding adjacent memory regions; this is especially helpful in the various region merging operations.*

*The red-black tree is useful for performing a search of the regions looking for one including a particular address.*

Points	Remarks
10	per data structure description
2	linked list description that doesn't mention scanning or finding adjacent regions
5	linked list description that only mentions scanning, not adjacency

2. When an interrupt or exception occurs, the hardware puts just enough of the CPU state on the system stack to be able to do a return. The Linux assembly-code interrupt handler puts all the rest of the CPU state on the stack before doing any other work. Why?

*It's entirely possible (even likely!) that all the registers are being used for things expression evaluation. Once the kernel goes back to C code, all these registers are going to get corrupted. So... we need to save them first.*

3. Suppose two kernel threads are obtaining input and placing it in a buffer. They each use the same code:

```
while (1) {
    awry[i++] = newin();
    if (i > 100) i = 0;
}
```

The `newin()` function correctly obtains one unit of input. `awry` and `i` are both shared between the threads. In this machine, the `i++` operation is not atomic.

- (a) Show that it is possible for inputs to be lost using this code (assume some other code someplace is correctly removing the inputs from the array – the inputs aren't lost due to `i` wrapping around to 0).

*Note that there are many different traces that will show the same behavior; this is just one of them.*

- i.  $P_0$  reads  $i$ .
- ii.  $P_1$  reads  $i$  (and obtains the same value  $P_0$  did).
- iii.  $P_0$  increments its copy of  $i$  (and since  $i++$  is a post-increment operator, it will “remember”  $i$ 's old value for use in writing to `awry`).
- iv.  $P_1$  increments its copy of  $i$  (likewise,  $P_1$  will “remember”  $i$ 's old value. This is the same value  $P_0$  is remembering).
- v.  $P_0$  writes its copy of  $i$  back to memory.
- vi.  $P_1$  writes its copy of  $i$  back to memory.

vii.  $P_0$  writes its unit of input to *awry*, at the location specified by *i*'s old value.

viii.  $P_1$  does the same thing, and writes to the same location.

At this point, the unit of input read by  $P_0$  has been lost.

- (b) Modify the code so that it will correctly put each new input in a new location in the array. You may add new local or global variables, you may restructure the code, and you may add any of the synchronization primitives from Chapter 5. A correct solution will only lock the operations that actually need it (so don't just insert a spin-lock or something at the beginning and the end of the loop).

The main thing here is that the increment (and remember the old value) has to be made atomic. Here's code that will do it (assuming *tmp* is local to this thread, while *sem* is a global semaphore).

```
while (1) {
    int tmp;
    down(sem);
    tmp = i++;
    if (i > 100) i = 0;
    up(sem);
    awry[n] = newin();
}
```

Points	Notes
5	vague description that doesn't really talk about interleaved memory accesses
3	<code>newin()</code> in mutex; <code>if()</code> not in mutex
3	assuming broken behavior in the consumer process

4. When the kernel decides to switch the current process, the first step is to change the Page Global Directory, which installs a new address space. How does the kernel keep executing when its address space has just been changed out from under it?

*When a new process is created, the kernel page tables are copied into its page tables. So the memory space has moved out from under it, but the part that matters to it (the kernel itself) doesn't get changed when that happens.*

5. In writing an ordinary program, what ideas from the kernel's slab allocator could be useful in doing memory management? This is assuming you're doing your own memory management, and not just calling `malloc()` whenever you want a new object.

*Pretty much the whole thing can be taken intact, except that you'd use `malloc()` to get slabs of objects. Allocating and deallocating objects would be much faster than using `malloc()` (since `malloc()` is general purpose); you could use slab coloring to help your cache hit rates.*