Comprehensive Exam — Systems software

Fall 2006

November 15, 2006

Answer each of the following questions, and give a sentence or two justification for your answer **(5 points each)**.

1. Initial versions of BSD disabled and re-enabled interrupts as the sole means of providing mutual exclusion. What must BSD do (and why) if a kernel thread running with interrupts disabled sleeps (i.e., puts itself on a block queue and allows another thread to run)? Give an example of a problem this can cause. (Note: you only need to reason about interrupts to answer this question, the details of BSD are irrelevant.)

2. You have three processes P1, P2, P3, where P1 has the highest priority, P2 the middle, and P3 the lowest. P3 acquires a lock `l`: give an example where this can interact arbitrarily badly with priorities and suggest a realistic fix.

3. Why can we trivially implement perfect LRU for files accessed using `read` and `write` system calls, but not for memory mapped files?

Answer each of the following questions, and give a sentence or two justification for your answer **(10 points each)**.

1. Consider very-broken code that uses the "double-check" lock idiom:

```
0:   int *p = 0;
....
1:   if(!p)
2:      lock(l);
3:      if(!p) {
4:         int *t = malloc(sizeof *t);
5:         *t = 3;
6:         p = t;
7:      }
8:      unlock(l);
9:   }
10:  x = x / *p;
```

   How can an optimizing compiler interact badly with the use of the `t` temporary? Assume you don't use an optimizing compiler: if another thread has done a `free` previously, what can happen if the initializing thread gets context switched immediately after line 6 and another thread executes line 10 on another processor?

2. Assuming you have a program that never frees memory running on an OS that provides a routine "`void *getpage(void)`" that allocates a page of virtual memory and returns a pointer to it. Explain how to write a "`malloc`" implementation guaranteed to never fragment more than a page of memory. Can you generally make the same guarantee if the program uses `free`? Give the intuition why or why not.

3. On a system with a TLB, what does the OS have to do after revoking a page from a process? Assuming a fixed page size, what feature of the TLB will place a hard upper bound on the amount of physical memory your system can use? You have a 64-entry, direct-mapped TLB, and 4K pages. Describe two repeating memory accessses that will produce horrible performance as compared to a 64-entry, 2-way set associative TLB.

4. Conservative OSes such as various BSD operating systems require that whenever a physical page is allocated for virtual memory that they can also reserve a page size chunk of swap space. Other OSes (such as Linux) use a page allocation policy called "over commit" where they find space in swap on demand, as processes need it. What is the advantage of overcommit? The price Linux pays for these benefits is that under high memory load it kills processes. Why does it have to do this?

5. A "perfectly consistent" file system would synchronously write all persistent data modified by a system call to disk before returning to the user. Most file systems do no such thing but instead defer writes as much as possible. Assume: you can detect when the user can externally observe the result of a program. Explain how you can modify the synchronous file system to be much more efficient while still behaving "as if" it was synchrnous.

6. Assume the common Unix file system interface, in particular, that you have a way to *non-atomically* write file data, that you have `sync()`, and that rename is atomic. Explain how to overwrite a file `A` with new contents such that any crash will result in A having either the old or new contents. Give the sequence of calls you would do for this.