Midterm Exam

CSC 256/456

March 7, 2007

1. **Privileged instructions:** (10 points) Which of the following instructions should be privileged (and only allowed to execute in kernel mode)?

   - Load a value from a memory address to a general-purpose register.
   - Load a value from a memory address to an entry in the translation look-aside buffer (TLB).
   - Flush/clear the content in the translation look-aside buffer (TLB).
   - Read the current value in the program counter (PC) register.
   - Set a new value in the program counter (PC) register.
   - Turn off interrupts.
   - Call a subroutine — pushes the return address onto the stack and jumps to the subroutine.
   - Return from a subroutine — pops the return address off the stack and jumps to it.
   - x86 *MMX* (Matrix Math eXtensions) instructions — access and perform arithmetic operations on a special set of 64-bit registers to speed up multi-media data processing.
   - Halt the CPU.

   Answer: Load into TLB; flush the TLB; turn off the interrupts; halt the CPU.

2. **Processes & threads:**

   (a) (2 points) Describe the difference in context switch costs for processes, kernel threads, and user-level threads. Explain your answer.

   Answer: A process context switch is more costly than a kernel thread context switch due to the additional address space switching cost (*e.g.*, TLB flushing, updating page table registers). A kernel thread context switch is more costly than a user-level thread context switch due to the additional kernel-user crossings.

   (b) (2 points) In order to achieve high performance, a newly developed Web server (called *Worthlix*) employs user-level threads to support concurrent requests. In other words, Worthlix spawns a new user-level thread to handle each incoming Web request. All user-level threads run on top of a single kernel thread. What is the problem for this method?

   Answer: There are at least two potential problems. First, a group of user threads can only utilize a single CPU. Second, when a single user-level thread blocks on I/O, the whole Web server will be blocked since the OS kernel does not know the existence of other ready-to-run user-level threads.

   (c) (2 points) Without preemption, a process (or thread) can run forever and cause other processes (or threads) to starve. Describe the preemption mechanism for processes and kernel threads.

Answer: The preemption for processes and kernel threads is based on clock interrupts. At every clock interrupt, the kernel gets the opportunity to preempt the current running process (or kernel thread) and switch to someone else.

(d) (4 points) Please also suggest a possible way to support preemption for user-level threads. You do not have to work out all the details.

Answer: To support user-level thread preemption, the kernel can make an upcall to the user-level thread manager at each clock interrupt such that the user-level thread manager can have an opportunity to switch out the current user-level thread. Another possible way to support user-level thread preemption is to customize the user-level thread code such that it calls the thread scheduler every once a while. This may be achieved through some compiler support. There is a non-trivial amount of detail omitted here.

3. (10 points) The following algorithm, developed by Dekker, is the first known correct software solution to the critical section problem for two processes. The two processes, P_0 and P_1, share the following variables:

```
bool flag[2];  /* initially both flags are false */
int turn = 0 or 1;
```

The following program is for process P_i (i=0 or 1), with P_j (j=1 or 0) being the other process:

```
for (;;) {
   flag[i] = true;
   while (flag[j])
      if (turn==j) {
         flag[i] = false;
         while (turn==j);  /* spin wait */
         flag[i] = true;
      }
   ...
   critical section
   ...
   turn = j;
   flag[i] = false;
   ...
   remainder section
   ...
}
```

Explain that the algorithm satisfies all three requirements (mutual exclusion, progress, bounded waiting) for the critical section problem.

Answer:

- **Mutual exclusion:** We prove it by contradiction. Assume both P_0 and P_1 are in the critical section simultaneously. Without loss of generality, we assume P_0 enters first and P_1 enters when P_0 is already inside. At the time P_1 enters, condition `flag[1] && !flag[0]` holds. This is impossible since `flag[0]` is always true when P_0 is in its critical section. *Contradiction!*

- **Progress:** When one of them (assume P_j) is not trying to enter the critical section, then `flag[j]` is false. Therefore P_i can always enter whenever it wants to. When both processes are trying to get into the critical section, P_{turn} can always enter. Thus progress is guaranteed.
- **Bounded waiting:** When both processes continuously try to get into the critical section, the variable `turn` alternates between 0 and 1. This forces P_0 and P_1 to alternate their entrance into the critical section.

4. **Block and deadlock**.

   (a) (5 points) Programs may request some free memory during the course of execution. Without special care, a request for free memory may block for a potentially long period of time (several milliseconds). Explain how this can happen.

   Answer: When there is no free physical memory, some existing pages have to be evicted. If such pages are dirty, they have to be written to disk before free memory becomes available.

   (b) (5 points) Blocking inside an interrupt handler is a recipe for deadlock. Explain how deadlock may occur when an interrupt handler makes a potentially blocking request for free memory.

   Answer: When the request for free memory blocks on I/O, the I/O may never complete since the I/O completion interrupt is disabled within the interrupt handler.

5. **Reliability**. Device drivers are probably the buggiest part of an OS kernel since their developers (hired by device manufacturers) may not be as *nerdy* as the core kernel developers.

   (a) (4 points) In order to prevent buggy drivers from overwriting other parts of the kernel, we let each device driver to have its own kernel-level page table. A driver's page table makes clear that it can only write to a designated set of pages. When the CPU control transfers in and out of a driver, the kernel switches page tables appropriately. Explain that this scheme can only protect against buggy drivers, not malicious drivers.

   Answer: Since device drivers run in kernel mode, they can almost do anything they want. In particular, a malicious driver can change its page table (*e.g.*, making all pages writable) or it can even directly manipulate the TLB (for software-filled TLBs) such that it can write to anywhere it wants to.

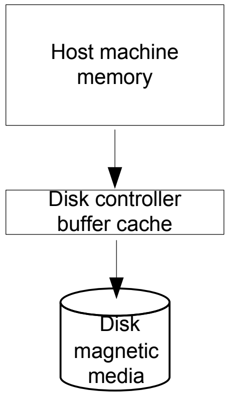
   (b) (3 points) Except overwriting other parts of the kernel, describe another potential harm of buggy (not malicious) drivers on the rest of the operating system.

   Answer: They may go into an infinite loop. They may also go into deadlocks.

   (c) (3 points) The part of device drivers that uses privileged I/O instructions or accesses privileged device registers must run in the kernel mode, but other parts may run in the user mode. Running parts of device drivers in user mode may improve the OS reliability. Describe any performance problem with this scheme.

   Answer: This would result in more kernel/user crossings, which can be costly.

6. **Second-level buffer cache**. The disk or storage controller contains a fast-access buffer that is independent of the host memory. The buffer can be accessed in a much faster speed than the disk drive magnetic media. The size of the controller buffer is typically between 2MB and 64MB, which is much smaller than the host memory size. Since the disk controller buffer



cache will only be contacted when a data access misses the host machine memory, we call it a *second-level* buffer cache.

(a) (5 points) Assume the host operating system employs the LRU page replacement in its memory management. Would it be effective if the disk controller also employs the LRU page replacement in the controller buffer cache management? Explain your answer.

Answer: LRU caching is not very useful here since the data that is just sent to host will almost certainly be cached in the host memory. Remember that the host memory is much larger than the controller buffer.

(b) (5 points) Explain how you would manage a second-level buffer cache efficiently. It is fine if your management scheme needs certain information from the host machine. Of course, to ease the implementation, we would like as little assistance from the host machine as possible. Be as open-minded as you can in answering this question. I am not looking for a particular *correct* answer.

Answer: The key is to cache data that is not currently in memory. If the host can provide information on evicted data, then the disk controller can cache those evicted data. Every time some new data is loaded into the host memory and the disk controller buffer has a cached copy of the data, the data should be removed from the disk controller buffer cache.