CSE421 Midterm Solutions

—SOLUTION SET—

09 Mar 2012

This midterm exam consists of three types of questions:

1. **10 multiple choice** questions worth 1 point each. These are drawn directly from lecture slides and intended to be easy.
2. **6 short answer** questions worth 5 points each. You can answer as many as you want, but we will give you credit for your best four answers for a total of up to 20 points. You should be able to answer the short answer questions in four or five sentences.
3. **2 long answer** questions worth 20 points each. **Please answer only one long answer question.** If you answer both, we will only grade one. Your answer to the long answer should span a page or two.

Please answer each question as **clearly** and **succinctly** as possible. Feel free to draw pictures or diagrams if they help you to do so. **No aids of any kind are permitted.**

The point value assigned to each question is intended to suggest how to allocate your time. So you should work on a 5 point question for roughly 5 minutes.

Multiple Choice

1. (10 points) Answer all **ten** of the following questions. Each is worth **one** point.

   (a) In the story that GWA (Geoff) began class with on Monday, March 4th, why was the Harvard student concerned about his grade?
   ◯ He never attended class. ◯ He never arrived at class on time. [✓ **He usually fell asleep in class.**] ◯ He was using drugs.

   (b) All of the following are inter-process (IPC) communication mechanisms *except*
   ◯ shared files. ◯ exit codes. ◯ pipes. [✓ **non-uniform memory.**]

   (c) New processes are created by calling
   [✓ `fork()`.] ◯ `exec()`. ◯ `create()`. ◯ `new()`.

   (d) What does `exec()` do to the process file table?
   ◯ Copies it. ◯ Opens `STDIN`, `STDOUT` and `STDERR`. [✓ **Nothing.**] ◯ Resets it.

   (e) What MIPS instruction is used by the kernel to return to userspace after handling an exception?
   [✓ `rfe`] ◯ `lw` ◯ `syscall` ◯ `addiu`

   (f) Which of the following is *not* a requirement for deadlock?
   ◯ Multiple independent resource requests. [✓ **A linear dependency graph.**] ◯ Protected access to shared resources. ◯ No resource preemption.

   (g) Which of the following is *not* an example of an operating system policy?
   ◯ Deciding which thread to run. ◯ Giving preference to interactive tasks. [✓ **Using timer interrupts to stop a running thread.**] ◯ Choosing a thread to run at random.

   (h) When using our computers, normal users are generally *not* actively aware of
   ◯ responsiveness. ◯ continuity. [✓ **resource allocation.**] ◯ interactivity.

   (i) Con Kolivas is
   ◯ the maintainer of the Linux scheduling subsystem. ◯ a Turing award winner. ◯ opposed to the use of profanity. [✓ **an Australian anaesthetist and Linux hacker.**]

   (j) Address translation allows the kernel to implement what abstraction?
   [✓ **Address spaces.**] ◯ Files. ◯ Threads. ◯ Processes.

Short Answer

Choose **4 of the following 6** questions to answer. You may choose to answer additional questions, in which case you will receive credit for your best four answers.

Virtual Page Number	Physical Page Number	Permissions
98	119	Read
2	120	Read

Table 1: **TLB.**

Virtual Page Number	Location	Address	Permissions
98	Memory	119	Read, Write
0	Memory	12	Read, Write
2	Memory	120	Read
3	Disk	2	Read, Write

Table 2: **Process Page Table.** Each entry corresponds to a PTE.

2. (5 points) Using both the TLB (Table 1) and page table (Table 2) for the running process above, describe what would happen if each of the following pseudo-instructions were executed. In particular, make sure to identify any TLB or page faults. Assume this (fairly weird) machine uses 1000 byte pages.

   Note: to make things easier on everyone the question uses base-10 arithmetic.

Solution:

1. load `2378` $\rightarrow$ load `120378`
2. store `98733` $\rightarrow$ TLB fault (entry marked read-only), store `119733` (entry marked read-write in PTE).
3. load `36788` $\rightarrow$ TLB fault (entry missing), page fault (no PTE). Kernel must either kill this process or create a new PTE and load the TLB appropriately before the instruction can continue.
4. load `143` $\rightarrow$ TLB fault (missing entry), load `12143` (PTE indicates that the page is in memory).
5. store `3700` $\rightarrow$ TLB fault (entry missing), page fault (page is on disk). The operating system will load the page into a physical page and restart the instruction. Note that the address in the PTE is a disk address, *not a memory address.* There is no way to know the final translation given the information you were provided.

3. (5 points) Identify and describe three serious problems with the code snippet below. (You may want to use the line numbers to help identify the problems.) Assume `sharedStateLock` and `sharedStateCV` have been properly initialized.

```
 1
 2 struct lock * sharedStateLock;
 3 struct cv * sharedStateCV;
 4 bool getGoing = false;
 5 int sharedState = 0;
 6
 7 void
 8 fubar(int doubleRainbow)
 9 {
10   // Might already have the lock!
11   if (!lock_do_i_hold(sharedStateLock)) {
12     lock_acquire(sharedStateLock);
13   }
14
15   // Wait to get going! Grab the lock to protect the shared state.
16   while (!getGoing) {
17     ;
18   }
19   lock_release(sharedStateLock);
20
21   // Reset shared state before we make our changes.
22   sharedState = 0;
23
24   lock_acquire(sharedStateLock);
25   sharedState = doubleRainbow;
26   lock_release(sharedStateLock);
27
28   // Let everyone know about the double rainbow!
29   cv_signal(sharedStateCV, sharedStateLock);
30   return;
31 }
```

Solution to Problem #3

Solution:

1. **Line 11:** incorrect use of the `lock` interface. Method `lock_do_i_hold()` is intended to be private.
2. **Line 15:** busy waiting.
3. **Line 15:** if `sharedStateLock` really protects `getGoing` then `getGoing` can never change while we are looping holding the lock! Otherwise, `sharedStateLock` does not really protect `getGoing`. Either way, it's a problem.
4. **Line 22:** resets `sharedState` after releasing the lock in Line 19.
5. **Line 29:** calls `cv_signal()` without holding the lock.
6. **Line 29:** should call `cv_broadcast()` to let *everyone* know about the double rainbow.

4. (5 points) Give an example of (1) a hardware interrupt, (2) a software interrupt, and (3) an exception. (Three examples total.) Briefly describe what happens when an interrupt is triggered.

Solution:

- **Hardware interrupt:** disk read completes, network packet arrives, timer fires, etc.
- **Software interrupt:** system call such as `read()`, `write()`, `fork()`, etc.
- **Exception:** divide by zero (our favorite!), invalid instruction, TLB miss.

When an interrupt is triggered the processor:

- enters privileged mode,
- records state necessary to process the interrupt,
- jumps to a pre-determined memory location and begins executing instructions.

5. (5 points) First, from the perspective of the operating system, what is the difference between interactive and non-interactive threads? Or, put another way, describe how the operating system might try to guess whether a thread is interactive or not. Second, describe how either multi-level feedback queues (MLFQ) or the Rotating Staircase Deadline (RSDL) scheduler prioritize interactive threads.

Solution:

Interactive threads interact with the user and tend to spend a lot of time sleeping waiting for user input. So the operating system might guess that a thread that spends most of its time on the waiting queue (waiting for I/O to complete) is an interactive thread. Conversely, a thread that spends most of its time using the CPU and not using devices can, by definition, not be interacting with the user.

MLFQ preferences interactive threads by penalizing threads that use their entire CPU quantum by lowering their priority. Threads that sleep before their quantum is complete, as interactive threads frequently do, receive a priority boost.

RSDL preferences interactive threads in a similar way. Threads that consume their quantum fall down the staircase, where they must compete for CPU time with lower-priority threads. Threads that sleep frequently will usually not use their quantum before a “major rotation” takes place, at which point they will return to their starting priority during the next scheduling round. RSDL also produces a bounded waiting time for threads at any level of the scheduler, which is great for interactivity.

6. (5 points) Operating systems require special privileges to multiplex memory. Below, describe:

   - **what** special privileges are required,
   - **how** they are used,
   - and **why** they are needed.

Solution:

- **What special privileges are required:** the operating system controls the virtual to physical address translations implemented by the MMU, either by loading the TLB itself (software-managed TLB) or by controlling the page table entries (hardware-managed TLB).
- **How they are used:** by controlling the indirection from virtual to physical addresses, the operating system can control what memory a process has access to and prevent it from accessing memory it should not have access to.
- **Why they are needed:** the address space abstraction is private to each process and IPC using memory sharing should require explicit cooperation by both processes. Adding the virtual-to-physical level of indirection allows the kernel to preserve the private nature of process memory.

7. (5 points) Describe the tradeoffs surrounding memory page size. What happens when pages become very small? What happens when pages become very large?

Solution:

When pages become *very small*, internal fragmentation decreases (good), the size of kernel memory management data structures increases (bad), and the amount of memory that can be translated by a fixed-size TLB decreases (bad).

When pages become *very big*, internal fragmentation increases (bad), the size of kernel memory management data structures shrinks (good), and the amount of memory that can be translated by a fixed-size TLB increases (good).

Long Answer

Choose 1 of the following 2 questions to answer. **Please do not answer both questions.** If you do, we will only read one.

If you need additional space, continue and clearly label your answer on the back of this or other exam sheets.

8. (20 points) Choose one of the following two questions to answer:

    1. **User- v. kernel-level multithreading.** In class we focused our discussion of the thread abstraction on kernel-level threads. However, a popular alternative is to implement threads in userspace libraries. We refer to these threads as user-level threads.

       First, explain what a userspace library would need to do to implement the thread abstraction. How are threads created? Where is thread state stored? How do you perform a context switch? Can you implement preemption and, if so, how? What support from the kernel, if any, is needed to accomplish these things?

       Second, discuss the tradeoffs between implementing threads in userspace and in the kernel. What's potentially better about user-level threads? What's potentially better about kernel-level threads? Give one example of an application that you argue would perform better using user threads and one application that you argue would perform better with kernel threads.

---

    2. **System design principles.** We have discussed a number of general systems design principles throughout the semester. As an example, when motivating on-demand paging we introduced the idea that procrastination might be effective if it allows the kernel to avoid doing things that it will never have had to do, such as loading an unused code page into a process address space.

       List three *other* design principles that we have discussed this semester. Explain each design principle clearly and illustrate each principle with an operating systems example drawn from class. In addition, for each principle construct a new example of its applicability not drawn from class. Your examples do not necessarily have to be drawn from the world of operating systems or even the world of computers, but those are good places to start.

Solution: User-level threads.

Solution:

- **Grading rubric.**
  This question was broken into 9 *mini questions*. Answering all 9 mini questions received a full mark of 20 points. Answering 7–8 mini questions received 15 points. Answering 4–6 mini questions received 10 points. Answering 2–3 questions received 5 points. Demonstrating some understanding of what a thread was received 2 points.
- **How are threads created?**
  Very similarly to the way that they are created in the kernel. Imagine a library routine `userthreadfork()` which is similar to `threadfork()` on OS/161 which a user process would reach through a `vfork()` or `clone()` type system call. `userthreadfork()` could either copy the state of the caller, like `fork()`, or take a function pointer and arguments to begin execution, like `threadfork()`.
- **Where is thread state stored?**
  Instead of storing its state in kernel memory, as `threadfork()` does, `userthreadfork()` stores thread state (registers and the stack) in user memory.
- **How do you perform a context switch?**
  Again, this is very similar to the way it is done in the kernel. The only meaningful difference is that the state is loaded and stored from or into user memory. And in fact, the standard C library provides some support for saving and manipulating thread context via the `setcontext` family of functions.
- **Can you implement preemption?**
  To some degree, yes. One example of how is for the thread scheduling library to use signals generated by a periodic timer to stop currently-running threads and perform user thread scheduling. Unlike kernel preemption, however, a lack of privilege means that user threads must agree to not overwrite the process signal handlers with their own code which could prevent the userspace thread library scheduler from executing.
- **What kernel support is needed?**
  Not much except for possibly timer-driven signals.
- **Tradeoffs between user and kernel threads.**
  The tradeoffs primarily concern two properties of user-level threads: (1) they do not require a kernel boundary crossing to create, but (2) they are also not visible to the kernel. Because the process does not have to enter the kernel thread creation, destruction, and scheduling may be much faster and more lightweight. However, the process will be scheduled by the kernel as if it only had one thread, and in addition a single thread in a multi-userthreaded application that performs a blocking system call will block *all* threads until the call completes.

For these reasons, multi-threaded applications that frequently spawn new threads to perform short-lived tasks, particularly computational tasks, may do better with a userspace threading library. An example might be a GUI application that forks threads in response to user actions. These threads might not do much except update some in-memory state and exit, and provide more of a conceptual framework for programming the GUI. In this case, thread creation and context switching are common threads will not be around long enough to need kernel scheduling visibility.

On the other hand, multi-threaded applications that establish so-called "thread pools" consisting of long-lived threads that do many more complex tasks involving I/O may benefit from the kernel's knowledge of these threads existence and the ability to do multiple blocking system calls concurrently. An example of this kind of application might be a web server which creates a pool of threads that are assigned to respond to incoming connections. Rendering a page requires a fair amount of both computation and I/O, and because the thread pool is established at the time that the webserver is launched creation and destruction are rare.

Solution: Design principles.

Solution:

- **Grading rubric.**
  5 points for one design principle with examples. **10 points** for two design principles with examples. **15 points** for three design principles but unclear or poorly-explained examples. **20 points** for three design principles with strong examples.
- **Separate policy from mechanism.**
  An example from class is the separation of policy and mechanism in the operating system scheduling system. Mechanism consists of the process of performing a context switch, allowing us to switch between threads, as well as using periodic timer interrupts to force context switches to occur and guarantee that the operating system will always regain control of the machine. Policy consists of determining what threads to run in what order and for how long. Well-written schedulers—such as the one in OS/161—allow the policy to be changed while utilizing the same underlying mechanisms.

  One of many examples *not* drawn from class is the movement towards software-driven routers such as OpenFlow. Historically routers have intermingled policy and mechanism, making their routing decisions difficult to control. New routers try to cleanly separate the control and data plane to enable more flexible reconfiguration. Here control and data map roughly on to policy and mechanism.
- **Keep It Simple Stupid (K.I.S.S.).**
  An example drawn from class is the random scheduler or base and bounds address translation. K.I.S.S. is intended to represent the idea of doing the simpler thing *first* and then improving it as necessary, so another example would be linked list page tables which may be simpler to implement than multi-level tables and work well as long as the number of pages is small.

  As a design principle, many of Apple's products and their interfaces embody this idea. Many iPod competitors were crowded with tiny buttons, mysteriously-color LEDs and similar complex and inane "features." Apple stripped the interface down to its K.I.S.S. components which made it not only more powerful but more beautiful. (Full disclosure—the iPod that my wife and I own is extremely old and I do not consider myself an Apple fanboy. That said, why can't other computer companies manage to make attractive products? Is it so hard?)
- **Use the past to predict the future.**
  An example drawn from class is our approach to thread scheduling. If we have identified that a thread only uses the CPU in short bursts before sleeping, we anticipate that it will continue to do that and take its past behavior into account when scheduling it.

  An example not drawn from class comes from the world of car insurance. When you are in an accident, you will immediately pay more for insurance. The insurance provider is using your past behavior (you get into accidents) to predict the future cost to insure you.

(Hopefully everyone got this one after we chanted it repeatedly in class like operating system monks.)

- **Use a cache.** Or, put a small fast thing in front of a big slow thing to make it look faster. An example drawn from class is using the TLB to cache address translations. The TLB can contain only a small number of the translations established by the operating system but it is much, much faster.

  An example not drawn from class is the tray of "freshly made" fast food out under heat lamps at your favorite fast food restaurant. The "cache" holds fewer burgers than the restaurant contains, but it is faster to grab one out from under the lamps than to make one from "scratch".

- **Add a level of indirection.** An example drawn from class is virtual to physical address translation. Rather than allowing processes to access physical memory directly, the operating system creates a level of indirection which provides the kernel more control. References can be revoked or shared and the underlying objects moved or altered.

  An example not drawn from class is canonical name (or DNS name) to IP address translation. Canonical names are not only more easy for humans to remember, but they can be translated in a variety of ways, point to different physical machines at different times, or revoked from one owner and reassigned to another.

- **Avoid doing things immediately you might never have to do.** An example drawn from class is on-demand paging. Instead of loading pages into the process's address space when the address space is laid during `exec()`, or when heap is allocated using `sbrk()`, the kernel waits until the pages are referenced for the first time by an instruction executed by the process. At that point it either loads the page or creates a new page initialized to hold zeros.

  I'm guessing that nobody had a hard time coming up with examples here. I'm not going to share mine, however, since they are all things that I really *should* have done when I was asked! (But eventually someone stopped asking or the situation resolved naturally.)