CSE421 Alternate Midterm Solutions

—SOLUTION SET—

08 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 very 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) On Monday, March 4th, (GWA) Geoff began class with a story about which famous Saturday Night Live comedian?

   ◯ Tina Fey [✓ **Al Franken**] ◯ Sarah Silverman ◯ Dan Akroyd

   (b) Unlike threads, address spaces and files, the process abstraction is not tied to

   [✓ **a hardware component.**] ◯ another abstraction. ◯ a specific policy. ◯ user behavior.

   (c) What does the following system call do?

   ```
   waitpid(54);
   ```

   ◯ Fail. ◯ Return the exit code of process 45. ◯ Block until process 45 exits. [✓ **There is not enough information to tell.**]

   (d) What does `fork()` do to the process file table?

   ◯ Nothing. [✓ **Copies it.**] ◯ Opens `STDIN`, `STDOUT` and `STDERR`. ◯ Clears it.

   (e) What MIPS instruction do processes use to request kernel attention?

   ◯ `rfe` ◯ `lw` [✓ `syscall`] ◯ `addiu`

   (f) True or false: the following code ensures that variable `foo` is protected? (Assume `foo_lock` exists and was properly initialized.)

   ```
   lock_acquire(foo_lock);
   // modify foo
   lock_release(foo_lock);
   ```

   ◯ True. [✓ **False.**]

   (g) Which of the following is *not* an example of an operating system mechanism?

   ◯ A context switch. ◯ Using timer interrupts to stop a running thread. ◯ Maintaining the running, ready and waiting queues. [✓ **Choosing a thread to run at random.**]

   (h) Normal users are actively aware of computer

   ◯ resource allocation. [✓ **responsiveness.**] ◯ throughput. ◯ power consumption.

   (i) Who is Ingo Molnar?

   [✓ **The maintainer of the Linux scheduling subsystem.**] ◯ The author of the Rotating Staircase Deadline Scheduler (RSDL). ◯ An Australian who tried to change the Linux scheduler. ◯ A Turing award winner.

   (j) A virtual address might point to all of the following *except*

   ◯ physical memory. ◯ a disk block. ◯ a port on a hardware device. [✓ **a register on the CPU.**]

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.

2. (5 points) Describe the process of performing a context switch.

Solution: A context switch stops the currently running thread and starts another. The state private to each thread are the registers, which include the stack pointer and program counter. To save the state of the currently running thread the operating system copies the register values into what we refer to as a *trapframe*. To load the state of another thread the operating system copies the values from the trapframe created when we stopped the thread back to the appropriate CPU registers.

3. (5 points) Dewey, Cheatham, and Howe are car mechanics at the Good News Garage. One day Ray brings in his old MG which is, of course, in terrible shape. There are three distinct problems to solve. Dewey, Cheatham and Howe divide up the tasks between them, grab their tools and get to work.

   Like most car mechanics, Dewey, Cheatham and Howe share a set of tools (they're cheapskates), only pick up one tool at a time (the other hand is holding a beer), grab all the tools that they need before beginning a task (so that they don't have to move after they begin), and don't surrender a tool until they have finished whatever it is they are trying to do (or think that they have finished). As they approach the MG, Dewey sees that his problem requires the wrench and the hammer, Cheatham sees that his requires the pry bar and the wrench, and Howe sees that his requires the pry bar, hammer, and blow torch.

   Despite their best intentions, describe a situation in which Dewey, Cheatham and Howe will find themselves unable to get any work done. Propose two separate solutions that will allow them to complete the job and get Ray's car back on the road. Your solutions should allow multiple mechanics to work on the car at once when possible.

Solution: A deadlock can occur if Dewey grabs the wrench, Cheatham grabs the pry bar, and Howe grabs the hammer. At this point Dewey needs the hammer, which Howe has; Cheatham needs the wrench, which Dewey has; and Howe needs the pry bar, which Cheatham has. (Howe can still grab the blow torch.)

There are several possible solutions, each relaxing one deadlock condition:

1. **Shared resources.** The mechanics should buy more tools.
2. **No preemption.** The mechanics should surrender all tools if they see that they cannot make progress.
3. **Independent requests.** The mechanics should grab all of the tools they need at once. (Putting down, unfortunately, the beers.)
4. **Circular dependencies.** The mechanics should agree that they must always grabs tools in a fixed order, e.g. first the wrench, then the hammer, then the pry bar, then the blow torch.

4. (5 points) We discussed several approaches to doing inter-process communication, or IPC. Below, describe two IPC mechanisms. For each, provide an example of interaction between two processes well-suited to that mechanism.

Solution: We discussed at least five IPC mechanisms: exit codes, shared files, pipes, signals, and shared memory.

1. **Exit codes.** When a process exits, it can return an integer value that can be retrieved by the process that created it.

   An example of interaction suited to this mechanism is when a process launches another to perform a task for it. To know whether the task was completed properly, or if an error occurred, it can examine the exit code.

2. **Shared files.** After `fork()`, the parent and child share open file descriptors. In this case they will share the offset into the file. Alternatively, two unrelated processes can simply open the same file, in which case they will *not* share the offset. In either case, reads and writes to the file by one process will be visible be the other.

   A simple example is a background process that loads its configuration from a file. A separate interactive tool might be used to edit the configuration file, which changes visible to the background process the next time it rereads the file.

3. **Pipes.** Pipes are producer-consumer memory buffers. Each has a read and write end. Data written to the write end is immediately visible to the read end. By creating a pipe before calling `fork()` and manipulatin the read and write ends after the new process is created, buffers between related processes can be established.

   Pipes are commonly used to create complex processing tools by chaining together multiple processes each of which performs a simple task.

4. **Signals.** A signal allows one process to invoke a signal handler in another process. Processes can choose how to handle signals that they receive.

   Signals are useful as a way of one process alerting another that something has happened. Continuing with the configuration editor example above, the configuration editor might send a signal to the background process when it's configuration files changes, causing it to wake up and reconfigure itself.

5. **Shared memory.** Shared memory is similar to shared files, except that the shared memory region requires more explicit set up and acts like memory.

   Shared memory can be suitable for processes that want to emphemeral data structure that they establish in memory. As in shared files, coordination is difficult and usually requires another mechanism.

5. (5 points) Explain what happens when a page is swapped in from disk.

Solution:

1. The instruction that caused the page to be swapped in is stopped.
2. We must allocate page of memory to hold the contents of the page we are about to retrieve.
3. The operating system locates the page on disk using the PTE.
4. We copy the data from the on-disk page to the newly-allocated memory page.
5. The PTE is updated to indicate that the page is now in memory.
6. The TLB is loaded with the appropriate virtual to physical translation.
7. The instruction that caused the page fault is restarted.

6. (5 points) Operating systems require special privileges to multiplex the CPU. 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 is invoked on hardware interrupts, and in this case specifically on hardware interrupts caused by the timer.
- **How they are used:** timer interrupts are used to ensure that the operating will eventually gain control of the system again, at which point it can choose to deschedule the currently-running thread.
- **Why they are needed:** if the operating system did not have a way to *always* stop the currently-running thread it might run forever or for longer than the operating system wants it to.

7. (5 points) It's your first week working at Macrohard, makers of the newly-popular Windix operating system. You have been hired as part of the Desktop Performance Group, focused on improving the somewhat sluggish performance of Windix on consumer machines.

   Your boss Geoff Challen comes to you early one morning perplexed. He has been working on a sophisticated new page replacement algorithm utilizing AI-inspired algorithms to choose the best page to swap out when memory runs low. He has convincing evidence that his new algorithm is doing a much better job of choosing a page than the very simple approach currently used by Windix. However, when he performs performance measurements he finds that a system using his new approach is actually *slower* despite the fact that it is choosing pages to evict more effectively.

   Geoff thinks that there must be something wrong with the testing suite that he is using. You have a more convincing explanation. Detail it below.

Solution: Geoff's new algorithm is taking so long to run that any improvements to the page replacement policy are being more than cancelled by the time that the system is spending choosing which page to evict next.

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. **Compressed memory.** We have discussed swapping in class, which frees up memory by moving pages to disk. Another approach to freeing up memory compresses the content of one page allowing it to be stored in a smaller amount of memory. When the page is accessed again, the compressed page is decompressed to restore its contents.

      As an example, imagine that I have an algorithm that can reliably compress a 4K page into 2K. First, describe how your system would use this algorithm to free up system memory. Include a detailed description of when compressiond and decompression are performed. Compare the cost-benefit tradeoff of this compression-driven approach to swapping.

      Second, imagine I have a system that can either compress or swap pages to free memory. Describe how the system might choose which approach to use at a given point in time.

   ---

   2. **User-level system calls.** Context switching is expensive and creates an overhead for kernel entry and hence for making system calls from user space. For some system calls entering the kernel and running kernel code is unavoidable. For other system calls, however, the kernel transition may be avoided.

      Consider the `getpid()` system call. First, propose a way to implement `getpid()` entirely in usermode without entering the kernel. Please be detailed about how your new implementation will work and any changes to user processes or the kernel that will be required.

      Second, imagine that you have implemented your new `getpid()` which does not enter the kernel. Afterwards, however, you notice no speedup to your user programs. Explain why your change to `getpid()` results in no appreciable performance change.

Solution: Compressed memory.

Solution: Compressed memory.

Compression is similar to swapping in many ways but there are a few key differences. First, when the system needs to free up a page (4K) of memory, rather than moving one 4K page to disk (creating 4K) it must compress *two* pages of memory from a total of 8K to a total of 4K. Second, in order to create a single 4K page to use the compressed data must be moved to a separate page. So assuming the system starts with N free pages, the process works as follows:

1. Find one unallocated 4K page. ($N - 1$ free pages remain.)
2. Choose two allocated pages to compress. Compress both pages into the page allocated in Step 1. ($N - 1 + 2 = N + 1$ free pages remain.)

Note that Step 1 means that the system must always have at least one free page available in order to perform memory compression.

As far as the cost-benefit tradeoff compared to swapping, the benefit is the same: I have access to 4K of memory for some period of time. Here the time is bounded by the minimum of the two times until either page the system compressed is accessed. The cost, however, is quite different. For swapping, the cost was time and *disk bandwidth*. With compression, the cost is time and *CPU utilization*, since we are assuming that compression is a CPU-intensive task.

For the second part of the question, the key was to identify disk utilization with swapping and CPU utilization with memory compression. If disk bandwidth is limited, memory compression may be a good technique since it avoids the disk; alternatively, if CPU cycles are limited, swapping may be the better choice since it avoids the CPU.

Solution: User-level system calls.

Solution: One key thing to note is that the process PID does not change during the lifetime of the process.

Implementing it in userspace requires the kernel store the PID in a previously-agreed-on location where the process can find it without entering the kernel. The simplest solution is for the kernel to establish a virtual memory address within every process address space that always contains the process PID. Let's say that that address is `0x1000`. When each process is started, the kernel creates a mapping for `0x1000`, points it to a physical page and stores the process PID on that physical page.

At this point user processes can implement *getpid()* simply as *load 0x1000*. Note that this may require entering the kernel to translate this virtual address, but that may only happen once after which the translation will be cached by the TLB and the user-kernel boundary crossing eliminated entirely.

Regarding the lack of speedup, the explanation is simple: processes don't call `getpid()` that often! (Why would they?)