CSE 421/521 Midterm Solutions

—SOLUTION SET—

31 Mar 2017

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.

This year the midterm was *entirely drawn from questions that appeared in previous years*.

Multiple Choice

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

   (a) What does Carl say sometimes?
   √ **right** √ **on that** √ **ummm** √ **thing-in-a-Bob**

   (b) Intra-process (within) communication is easier than interprocess (between) communication.
   √ **True** ◯ False

   (c) 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.**

   (d) 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.**

   (e) Normal users are not aware of laptop
   ◯ responsiveness. ◯ screensavers. √ **resource allocation.** ◯ weight.

   (f) Process terminate by calling
   ◯ `kill_me_now()`. ◯ `kthanxbai()`. √ **`_exit()`.** ◯ `terminate()`.

   (g) Which of the following is the simplest scheduling algorithm to implement?
   ◯ Rotating staircase. ◯ Multi-level feedback queue. ◯ Round-robin.
   √ **Random.**

   (h) Which of the following is a requirement for deadlock?
   ◯ A single resource request. √ **A cycle in the dependency graph.** ◯ Unrestricted access to shared resources. ◯ Preemptible resources.

   (i) Which of the following is *not* an example of an operating system mechanism?
   ◯ Virtual to physical address translation. √ **Using priorities to choose the next thread to run.** ◯ An interrupt handler. ◯ Identifying interactive threads by observing their wait patterns.

   (j) 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.**

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) Given a simple MLFQ approach that rewards any thread that sleeps or yields before its quantum expires, first describe a way that a computationally-intensive thread can take advantage of a weakness in this approach to remain in one of the top queues (2 points). Second, propose a modification to MLFQ that addresses this problem (3 points).

This question appeared on the 2016 midterm.

Solution:

The problem is that if a thread can determine how long its quantum is, even approximately, it can arrange to call `yield` right before its quantum expires. As a result, it has done almost as much computation as it would have normally, but it is not demoted to a lower-priority queue as it should be.

The fix is simple. Instead of looking at whether a thread blocks or yields before its quantum ends, consider the percentage of its quantum it has used before it sleeps or yields. Either establish a stronger threshold for not being demoted (can't use more than 10% of the quantum), or use the percentage itself to make a more nuanced decision (if within 10% maintain, if between 10–50% demote one level, if over 50% demote two levels, or whatever).

Another way to fix this problem is simply to not reward threads that yield and only consider those that block. However, this isn't quite sufficient, since a thread may be able to block on something that it knows will return immediately—like a read from `/dev/random`—or arrange to have a second thread within the same process release it. Looking at the overall quantum usage is a safer bet.

3. (5 points) Describe two changes to the OS virtual memory system that you would need or might want to make to accommodate 48-bit virtual and physical addresses (1 point each). For each, describe how it would affect the computation or memory overhead of virtual to physical translation (2 points each).

This question appeared on the 2015 midterm.

Solution:

There were a few options here:

1. **Page table entry size.** You clearly need to do something about your page table entries, since whatever you did to bitpack for 32-bit virtual addresses is going to change with 48-bit virtual addresses. Almost inevitably, you're going to end up storing more state.
2. **Address space size.** You don't necessarily have to change the address space size, since you can simply fit more content from 32-bit wide address spaces into a system with 48-bit virtual addresses. You may want to change the break point between userspace and the kernel, however, since there is no point wasting any of the 4 GB wide virtual address space on the kernel at this point. Everything from `0x0` to `0xFFFFFFFF` should be used for the process, with kernel addresses outside of that range (¿ `0x100000000`). This doesn't necessarily have any direct effect on the overheads, although it may affect the size of other data structures (PTEs or page tables).
3. **Larger pages.** You may really want to increase the 4K page size at this point, with the concordant tradeoff between TLB faults and spatial locality. Larger pages may reduce the number of translations required.
4. **Different page table structures.** It could be time to add a third level to the existing two-level page tables, particularly if you decide to expand the the address space size. The goal would be to reduce the amount of space required for the page tables for common address space layout patterns.
5. **MMU changes.** Clearly the MMU and TLB need to be modified to help map 48-bit virtual addresses to 48-bit physical addresses. No effect on the kernel overheads here, but any speculation about the affect on hardware (wider addresses might imply fewer entries) would be accepted.

4. (5 points) We have discussed several cases where operating systems provide a useful illusion to processes. Name one (1 point), describe why it is useful (2 points), and briefly explain how it is provided, identifying any hardware cooperation required (2 points).

This question appeared on the 2014 midterm.

Solution:

There were multiple possible answers to this question. Here are a few of the obvious ones (although you may have received credit for another if you convinced us it was legitimate):

1. **Concurrency.** One single-core systems concurrency is an illusion, since there are never really two threads running at once. This is useful since it provides users with the illusion that multiple things are happening at once—music is playing while the web page is updating while they are typing into the terminal. Concurrency is provided by rapidly switching between threads fast enough to human perceptual limitations. This requires the ability to perform context switches between threads, stopping one's use of the processor to allow another to continue, and then returning the first to exactly where it was stopped.
2. **Atomicity.** When we discussed synchronization we identified cases where we wanted to make multiple actions that actually occurred separately happen all at once, or *atomicly*. An example is a modification to a shared data structure that requires multiple operations that should all happen at once—imagine adding an entry to a synchronized linked list, which requires updating multiple pointers. Atomicity is more than useful, and in certain cases (like our example) is actually required for correctness. One way we provide the illusion of atomicity is by locking shared data structures, which forces other users to sleep while we make our modifications, allowing them to all look atomic.
3. **Address spaces.** Address spaces comprise multiple illusions: that processes have access to a large (1) amount of private (2) contiguous (3) memory (4). In reality, (1) the size of the address space may be larger than the amount of available physical memory. In addition, the same physical memory may be temporally-multiplexed between processes, making it not entirely private (2). And it is certainly not contiguous, with any virtual page mapping to any physical page (3), and may not even be memory at all (4) if the page has been swapped to disk or the virtual address is set up to point to a file by `mmap`. But the address space abstraction is useful because it simplifies how processes deal with memory. They can lay out their memory the same way each time and logically separate regions, such as the stack and heap, in ways that allow them to grow dynamically. At some level all of these illusions are provided by translating virtual addresses into physical addresses, with the extra level of indirection allowing the OS to move memory around and reuse it for other purposes as long is the virtual addresses obey the memory interface.

5. (5 points) Provide one example of a exception that would terminate a running process (1 point) and one example of an exception that would not (1 point). Describe what happens when an exception occurs (3 points).

This question appeared on the 2013 midterm.

Solution:

A divide-by-zero or attempt to use a privileged instruction are exceptions that would terminate the processes, while using a memory address that the MMU doesn't know about but for which there is a valid mapping would not.

When an interrupt or exception is triggered the processor:

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

6. (5 points) We've presented synchronization primitives that use both active (or busy) and inactive (or blocking) waiting. First, explain the difference (3 points). Second, for each describe a scenario in which that form of waiting is more efficient and why (1 point per scenario).

This question appeared on the 2013 midterm.

Solution:

Busy waiting occurs when a thread continues to occupy the processor while waiting for something to happen. Blocking waiting occurs when a thread yields and asks the kernel to wake it when something specific has happened.

Busy waiting can be more efficient on multi-core systems when the critical section is short. In this case, the overhead to perform the context switch required for a blocking wait is larger than the time spent spinning waiting for another thread (or hardware device) to complete.

Blocking waiting can be more efficient when critical sections are long, and is required on single-core systems when the action that a thread is waiting on requires another thread to access the processor. In this case, the time spent spinning would waste a great deal of processor time while waiting for the other thread (or hardware device) to complete.

7. (5 points) Operating systems require special privileges to multiplex memory. Below, describe:

   - **what** special privileges are required (1 point),
   - **how** they are used (2 points),
   - and **why** they are needed (2 points).

This question appeared on the 2012 midterm.

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.

Long Answer

Choose 1 of the following questions to answer. **Do not answer both questions.** If you do, we will only read the shorter one. If you need additional space, continue and clearly label your answer on other exam sheets.

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

   1. **Wait Time Prediction.** When discussing schedulers, one of the aspects of future that we wanted to be able to predict was the wait time when a thread blocks. To be more concrete, when a thread blocked and was moved to the waiting queue, we might want to know *how long* the wait will take.

      First, discuss how online prediction of wait times might be accomplished on a real system—no crystal balls allowed (10 points). Identify a few things that a thread might block waiting for, and for each discuss whether a prediction algorithm makes sense and how it might be implemented. You might think about applying some of the system design principles we've discussed in class.

      Second, discuss how to incorporate this information into a scheduling algorithm (10 points). Feel free to choose one of the scheduling algorithms we've discussed in class in order to make your solution more concrete. You should explain how to use the output of your wait predictor and argue that it can improve some aspect of scheduler performance.

---

   2. **Jumbo Pages.** While operating system pages have traditionally been 4K, some modern operating systems support "jumbo" pages as large as 64K. Based on your excellent and fast implementation of virtual memory for CSE 421 ASST3 you are hired as a kernel developer for the new Lindows © operating system company. Unfortunately, your boss didn't take CSE 421 and derives most of his understanding of operating systems from the movie "Her"[1]. At present, Lindows does not support jumbo pages, but once your boss hears about them he is desperate to include them into Lindows © version 0.0.0.2. He asks you for help.

      First, explain why how and in what cases 64K pages would improve or degrade OS performance (10 points). What information about virtual memory use could help the OS decide whether to locate content on a jumbo or regular-sized page? Second, explain how, in certain cases, you can implement jumbo-page-like functionality on top of an existing system that supports 4K pages *without* changing the underlying memory management hardware (10 points). What MMU features are required for this to work? Which benefits of jumbo pages are preserved or lost by your approach?

[1] Her is a 2013 American science fiction romantic comedy-drama film centering on a man who develops a relationship with an intelligent OS with a female voice and personality. (Wikipedia)

Solution: Wait Time Prediction

This question appeared on the 2013 midterm.

There were two parts to this question: predicting waits, and using those predictions. To predict waiting times, we apply our chanted scheduling matram of *use the past to predict the future*. Consider a disk. We will discuss later in the semester that disk read and write latencies can vary for spinning disks. (Flash drives may be more predictable.) However, you might still be able to establish a distribution of wait times for disk reads, modeled based on the amount of data that is read or written. If you wanted to be more clever, you could incorporate information about the disk geometry and current head position, but that's not stuff we've covered so it's an optional part of the solution. The idea is to point out that certain kinds of waits are probably predictable to a certain degree, and the way to predict them is to identify them by type and keep some history about how long that kind of wait has taken in the past.

You definitely need to break this down by type of wait, however. Waiting for interactive input is likely to be entirely unpredictable. Waiting for network packets may be predictable to a degree, but network latencies vary quite a bit and so this might also be difficult to track. Waiting on synchronization primitives is another case, and here the wait predictability and time might vary quite a bit depending on the primitive and what it is used for. A lock predicting a critical section of known length, for example, might have very predictable waiting times assuming you can count the number of threads waiting.

There are probably multiple ways to use this information during scheduling. Here is one. When choosing the next thread to run, the scheduler can check the waiting queue to estimate the expecting shortest waiting time of blocked threads. If there is a thread that may be about to wake up, it may be best to schedule something that the scheduler thinks might also block, so that as soon as the thread on the waiting queue is ready to run it can be rescheduled. (This could be particularly important if the thread about to wake up is a high priority or interactive thread.) On the other hand, if all of the threads on the waiting queue are predicted to block for a long while, it might be preferable to schedule a long-running CPU-bound task that can run while the blocked threads wait. This way, the operating system can avoid the overhead of running the scheduler over multiple time steps.

Another option is to use the predicted waiting time directly to determine when to next run the scheduler. So every time the scheduler runs, it predicts the shortest wait on the waiting queue, sets a timer for that time interval and then chooses a thread to run. The next time the scheduler runs, it will either be because the timer fired—in which case we hope one of the blocked threads is now on the ready queue and the scheduler can make a new scheduling decision—or the currently-running thread blocked, in which case the scheduler must choose a new thread anyway.

Solution: Jumbo Pages

This question appeared on the 2014 midterm.

First, how and in what cases do 64K pages improve or degrade performance? This is related to the discussion we had in class regarding page size. A straightforward benefit of 64K pages is that they allow the TLB to map more memory with the same number of entries, potentially leading to fewer TLB misses and associated TLB faults. Since 64K pages are larger than 4K ones, the question reduces to in what case do larger pages help and hurt? They help when there is sufficient *spatial locality* in memory accesses within the contents of the 64K page and they hurt when they do not.

One way to think about it is once I touch one byte of memory on the 64K page, what is the likelihood I will touch bytes on the 15 other 4K pages inside the 64K page? If it's high, I might as well make room for the whole page as soon as I touch any byte inside of it; if it's low, I'm going to be moving a lot of memory around without any gain over locating the byte on a standard 4K page[2] So spatial locality is what the OS would like to know. Maybe the best way of finding out would be to ask the process to tell me directly, so if it has a data structure that spans multiple 4K pages it would be better stored on a 64K page. (In addition, I might be able to tell from accesses on groups of 4K pages that they really all belong to one larger page, but it's not clear if the content can be relocated at this point.)

So how do we implement pseudo-64K pages without actual hardware support, i.e. on 4K underlying pages? Easy: whenever we have a TLB fault within a 64K region identified as a 64K page we load entries (and page contents) for not only the 4K page that caused the fault but for all of the other 15 pages within the 64K page. In a way, we can be even more flexible than real 64K pages, since we could load N extra pages on each side of the faulting page (for example, although $2N + 1$ is hard to make even). Clearly this requires a software-managed TLB to do in all cases, since a hardware-managed TLB will load the page translation when it is in memory and not alert the OS. (You could still implement the same behavior on page faults, but that's not exhaustive enough to cover all cases.)

The benefits of 64K pages that we preserve are fewer TLB faults since all 15 extra 4K pages on the 64K page are now loaded in the TLB and into memory. Assuming sufficient spatial locality, this is a good thing! The benefit that is lost is that the amount of memory that the TLB can map is still limited by the underlying 4K page size.