CSE421 Midterm Solutions

—SOLUTION SET—

08 Mar 2013

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.

Statistics:

- **82** students took this exam.
- **33** was the median score.
- **32.55** was the average score.
- **7.92** was the standard deviation of the scores.

Multiple Choice

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

   (a) In the story of the angry diva, she was upset by the color of what candy?
   ◯ Skittles. ◯ Reeses' Pieces. √ **M&Ms.** ◯ Sour Patch Kids.

   (b) Which of the following is an inter-process (IPC) communication mechanism?
   ◯ Address spaces. ◯ Virtual addresses. √ **Pipes.** ◯ Grapes.

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

   (d) What does `mmap()` do to the process file table?
   ◯ Copies it. ◯ Opens `STDIN`, `STDOUT` and `STDERR`. √ **Nothing.** ◯ Waits for it.

   (e) Which of these schedulers cannot be implemented?
   ◯ Rotating Staircase. ◯ Multi-Level Feedback Queues. √ **Shortest-Job First.** ◯ Random.

   (f) 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.

   (g) Which of the following is *not* an example of an operating system mechanism?
   ◯ Context switching. ◯ Address translation. √ **Preferring interactive threads.** ◯ `panic()`.

   (h) An example of a continuity-oriented computer process would be
   ◯ backing up files. ◯ responding to a click event. ◯ opening a new Firefox tab. √ **watching a movie.**

   (i) The bounded buffer producer-consumer synchronization problem can be solved using
   √ **two semaphores.** ◯ one semaphore. ◯ locks. ◯ recursive locks.

   (j) Address spaces are *not* usually
   ◯ big. √ **fragmented.** ◯ sparse. ◯ organized.

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) We have discussed two cases where operating systems add a level of indirection to achieve a desirable property. Identify each, and explain why indirection is helpful in both cases.

Graded by Zihe Chen.

Rubric:

- **+1 point** per case correctly identified.
- **+3 points** for identifying the reasons for indirection correctly.

Solution:

1. **File descriptors.** We introduced the *file handle* as an extra level of indirection between the file descriptor (visible to the process) and the file object (maintained by the file system). We did this in order to allow file descriptors to be private, file handles to be shared after `fork()`, and file objects to be shared potentially system-wide.
2. **Virtual addresses.** By introducing *virtual addresses* as an extra level of indirection between processes and physical memory, we gain flexibility (virtual addresses can point to many types of storage) and control (the operating system controls the mapping).

Statistics:

- **48 out of 82** students answered this question.
- **3** was the median score.
- **3.5** was the average score.
- **1.26** was the standard deviation of the scores.

3. (5 points) Identify three similarities or differences between Multi-Level Feedback Queues (MLFQ) and the Rotating Staircase Deadline Scheduler (RSDL).

Graded by Zihe Chen.

Rubric:

- **+2 points** per correct similarity or difference.

Solution:

1. **Similarity:** both organize processes into queues at different priority levels.
2. **Similarity:** both use round-robin or some other scheme to schedule processes with equivalent priority.
3. **Similarity:** both try to reward threads that sleep often, hopefully interactive threads.
4. **Difference:** MLFQ as we presented it doesn't really have a notion of *static*, or user-set, priorities. Priorities for threads are entirely a result of their behavior. RSDL, on the other hand, uses static priorities to determine the starting level for each thread and then uses their behavior to determine how they move along the staircase.
5. **Difference:** MLFQ can suffer from starvation, while RSDL ensures that each iteration of the scheduler is bounded before it begins, allowing for thread behavior to be more easily modeled.
6. **Difference:** threads in MLFQ can move both up and down in priority, whereas in RSDL threads only fall *down* the staircase.

Statistics:

- **59 out of 82** students answered this question.
- **3** was the median score.
- **3.2** was the average score.
- **1.07** was the standard deviation of the scores.

4. (5 points) Provide one example of a exception that would terminate a running process and one example of an exception that would not. Describe what happens when an exception occurs.

Graded by Zihe Chen.

Rubric:

- **+1 point** for a fatal example.
- **+1 point** for a non-fatal example.
- **+3 points** for identifying what happens when an exception occurs.

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.

Statistics:

- **52 out of 82** students answered this question.
- **3** was the median score.
- **3.1** was the average score.
- **1.38** was the standard deviation of the scores.

5. (5 points) Describe the process life cycle and related system calls.

Graded by Zihe Chen.

Rubric:

- **-1 point** for every missing or incorrectly-described system call.

Solution:

- **Birth (`fork()`):** `fork()` creates a new process that is an identical copy of it's parent.
- **Change (`exec()`):** `exec()` allows a process to load a new image, replacing its current address space and other state.
- **Death (`_exit()`):** `_exit()` terminates the running process, passing an exit code that can be retrieved by...
- **Seance (`waitpid()`):** `waitpid()` retrieves the exit code after a process exits, potentially blocking until the process exits if it has not already done so when `waitpid()` is called.

Statistics:

- **70 out of 82** students answered this question.
- **5** was the median score. (This question was too easy.)
- **4.4** was the average score.
- **0.88** was the standard deviation of the scores.

6. (5 points) Distinguish between *atomicity* and *concurrency*. Both can be real, or illusions—describe examples of each, and describe how the illusion is implemented in each case.

Graded by Zihe Chen.

Rubric:

- **+1 point** for distinguishing between them.
- **+1 point** for each real example.
- **+1 point** for each illusion.

Solution:

Atomicity describes the fact that multiple things appear to happen all at once. Real atomicity, at least as visible to the operating system, can be provided by hardware instructions such as test-and-set or compare-and-swap. The illusion of atomicity, which is important to correctness when maintaining shared data structures, can be implemented by using critical sections to prevent concurrency access to a shared data structure. A thread inside the critical section can then make multiple modifications requiring many instructions, which all appear atomic to other threads outside the critical section.

Concurrency describes the fact that multiple things seem to be happening simultaneously. Real concurrency is present on any multi-core or multi-processor system. The illusion of concurrency can be produced by rapidly switching between tasks at rates faster than humans can perceive.

Statistics:

- **63 out of 82** students answered this question.
- **3** was the median score.
- **2.56** was the average score.
- **0.89** was the standard deviation of the scores.

Starting Virtual Address	Bound	Base Physical Address	Permissions
0	100	150	R,W
348	56	2040	W
3678	510	260	R,W,E
288	16	10,000	R

Table 1: **Segmentation Table.**

7. (5 points) Given the current MMU segment table above, indicate the result of the following five load, stores, and fetches (load and execute.) **Note: to make things easier on everyone the question uses base-10 arithmetic.**

Rubric:

Graded by Zihe Chen.

- **+1 point** for each translation.

Solution:

1. fetch 78: causes an exception. The address is valid, but the segment does not have executable permissions set.
2. load 388: causes an exception. Again, the address is valid, but the segment does not have read permissions set. (Are there cases where we might mark a segment as *write-only*? Yes, for security reasons, as well as midterm questions.)
3. fetch 4000: results in the instruction at physical address 582 being loaded and executed. (Yes, I needed a calculator.)
4. store 244: causes an exception, as no segment is loaded for this address.
5. store 0: results in a store to physical address 150.

Statistics:

- **65 out of 82** students answered this question.
- **5** was the median score. (This question was too easy.)
- **4.51** was the average score.
- **0.68** was the standard deviation of the scores.

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. **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. 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. 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. **M:N Threading Implementations.** We've discussed both the M:1 threading model, where all threads are implemented by userspace libraries, and the 1:1 threading model, where all threads are implemented by the kernel. Another model exists: the M:N threading model, where the kernel supports multiple threads (N) but userspace libraries use them to support $M - N$ more. (We consider M to be strictly greater than N.)

      First, explain why this could be preferable to both the M:1 and the 1:1 threading models. Which benefits of each are preserved? Which are lost?

      Second, discuss how to implement the M:N threading model. Describe the threading interface available to processes, interaction between the threading library and the kernel, and how thread scheduling is performed.

Rubric:

Graded by Aditya Wagh.

Both long answers were divided into two 10-point sections, each of which covers several issues. Based on your answer, you might receive:

- **10 points** for an excellent discussion of all or most of the issues,
- **8 points** if you covered most of the issues but made minor mistakes,
- **5 points** if some of your points are wrong,
- **3 points** for attempts that cover at least one issue,
- **0 points** for nothing or a completely irrelevant answer.

Statistics:

- **10** was the median score.
- **11.09** was the average score.
- **4.79** was the standard deviation of the scores.

Solution: Wait Time Prediction.

Rubric:

- **10 points** for predicting waits.
  - Did you use the past to predict the future?
  - Did you identify things that a thread might wait for?
  - Did you discuss in which cases we could predict wait times?
- **10 points** for using the predictions.
  - How do you select the next thread to run?
  - How do you determine when to run the scheduler?
  - How does your algorithm improve on existing solutions?

Solution:

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, for example. We will discuss later in the semester that disk read and write latencies can vary, particularly 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: M:N Threading Implementation.

Rubric:

- **10 points** for comparing with the M:1 and 1:1 models.
  - Do you identify the benefits preserved in each case?
  - What about the benefits lost?
- **10 points** for the implementation.
  - What is the thread interface you will provide?
  - What interaction (if any) is required between the threading library and kernel?
  - How is thread scheduling performed?
  - Is your implementation properly synchronized?

Solution:

First, how would the M:N model be preferable? Recall that one of the big problems with the M:1 model was that a multithreaded user process was unable to exploit the *actual* concurrency available on the machine, particularly by using multiple cores. Given that even smartphones today ship with multiple cores, this is a problem, and the M:N model works around that. You might set N to be equal to the number of cores on the system, allowing a multithreaded process to potentially be scheduled across all available cores assuming they are available. So this is an example of a benefit preserved from the 1:1 model.

A second benefit preserved from the M:1 model is the low scheduling overhead and control of scheduling provided to the user process. For the 1:1 model, the kernel chooses not only *how many* threads will run, but which will run and in what order. Given that kernel has less visibility into the user process, it may make poor scheduling decisions. The M:N scheduling model essentially says to the user library: "I'll allow you to run up to N threads at once, but you get to pick which ones from your pool of M." So the kernel chooses to allow *a* user thread to run, and the user threading library chooses which one and potentially switches between them during what the kernel thinks is a single scheduling quantum.

Implementing this model requires merging portions of the user- and kernel-level threading interface. First, the userspace threading library may want to give the user program control over what kind of thread is created: a `pthread_create` for user-level threads and a `clone` for kernel-level threads. Or it may not, and support only a user-level thread creation command while seamlessly (and transparently) multiplexing the user threads onto kernel threads. Essentially, the library may allow the process to control N directly, or it may not expose N but set up new kernel threads as needed by count of M userspace threads. (It may not make sense, for example, to create N kernel threads for a process that is single-threaded.)

Thread scheduling is now a collaborative effort between the kernel and the threading library. The threading library will probably set up signals to be delivered to each of the N threads it controls in order to allow the userspace thread scheduler to run. Note that the kernel can still interfere with thread scheduling: for example, imagine that there are N kernel threads, but the kernel only picks one to run. Is it the one the userspace scheduler would have picked? If so, we are lucky. If not, the userspace scheduler will have to hope that the time fires quickly and gives it a chance to choose the correct thread (out of the pool of M) to replace the currently running thread with.

A final consideration is that the threading library must now be thread safe. In the M:1 model, the kernel was essentially guaranteeing that any private process library was thread safe. Now, all of the functions and data structures used by the threading library must be thread safe.