CSE421 Alternate Midterm Solutions

—SOLUTION SET—

06 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 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.

Statistics:

- **6** students took this exam.
- **41** was the median score.
- **40.33** was the average score.
- **5.85** 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) Why wouldn't the diva sing?

   ◯ She needed vocal surgery. √ **Her dressing room contained inappropriately-colored M&Ms.** ◯ She was hungry. ◯ The lip-sync track was broken.

   (b) Which of the following operating system abstractions is not tied to a hardware component?

   ◯ Threads. √ **Processes.** ◯ Address spaces. ◯ Files.

   (c) What does the following system call do?

   ```
   exec("/bin/true");
   ```

   ◯ Fail. √ **Load and execute "/bin/true".** ◯ Return the exit code of process "/bin/true". ◯ Wait for process "/bin/true" to exit.

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

   ◯ Copies it. √ **Nothing.** ◯ Opens `STDIN`, `STDOUT` and `STDERR`. ◯ Clears it.

   (e) Which is probably a privileged instruction?

   ◯ Load a word from memory. √ **Change the interrupt mask.** ◯ Add two registers and place the result in a third register. ◯ Rotate the value of a register right by the value of a second register.

   (f) True or false: the following code ensures that variable `foo` is protected? (Assume `foo_lock` exists and was properly initialized; do not make any other assumptions.)

   ```
   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?

   ◯ 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.

   (h) Normal users are not aware of laptop

   ◯ responsiveness. ◯ screensavers. √ **resource allocation.** ◯ weight.

   (i) Who has to approve patches to mainline Linux?

   √ **Linus Torvalds.** ◯ God. ◯ Con Kolivas. ◯ Erasmus B. Dragon.

   (j) A virtual address might point to

   ◯ virtual memory. ◯ grapes. √ **0xdeadbeef.** ◯ 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) We've presented synchronization primitives that use both active (or busy) and inactive (or blocking) waiting. First, explain the difference. Second, for each describe a scenario in which that form of waiting is more efficient and why.

Rubric:

Graded by Zihe Chen.

- **+3 points**: describe the difference.
- **+1 point**: per scenario.

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.

Statistics:

- **6 out of 6** students answered this question.
- **5** was the median score.
- **4.83** was the average score.
- **0.37** was the standard deviation of the scores.

3. (5 points) Explain how to use the `fork()`, `pipe()`, and any other required system calls to establish the interprocess communication (IPC) between a parent and child process started by the following shell command:

   ```
   $ cat exam.tex | wc
   ```

   Drawing a diagram or set of diagrams may be helpful. You may also write pseudo-code as long as it is sufficiently clear. (It does not have to be able to compile.)

Rubric:

Graded by Zihe Chen.

- **-1 point**: for missing pipe.
- **-1 point**: for missing fork.
- **-2 point**: for the remainder after fork and explanation.

Solution:

Almost directly from the lecture slides.

```
1  int pipeEnds[2];
2
3  pipe(pipeEnds); // Initialize pipe file descriptors.
4
5  /* We'd actually need to manipulate the file table using dup
6     so that STDIN for wc and STDOUT for cat pointed at the pipe,
7     but we'll allow solutions that omit that step. */
8
9  int returnCode = fork(); // Should handle error here!
10
11 /* Parent and child roles below can be swapped. */
12
13 if (returnCode == 0) {
14         // Somebody needs to run wc.
15         close(pipeEnds[1]); # Not necessary, but improves clarity.
16         exec("/bin/wc");
17 } else {
18         // Somebody needs to run cat.
19         close(pipeEnds[0]);
20         exec("/bin/cat")
21 }
```

Statistics:

- **4 out of 6** students answered this question.
- **5** was the median score.
- **5** was the average score.
- **0.0** was the standard deviation of the scores.

4. (5 points) List and describe the three thread states we discussed in class. Next, describe four transitions between them including when and how they occur.

Rubric:

Graded by Zihe Chen.

- **+3 points** for identifying the three states.
- **+1 point** for four transitions.
- **+1 point** for when and how they occur.

Solution:

Thread states:

1. **Running:** actively executing instructions on a processor core.
2. **Ready**: not scheduled on a core, but ready to execute.
3. **Blocked or Waiting**: waiting for something to happen and not ready to run until it does.

Examples of transitions:

- **Running** $\rightarrow$ **Ready**: a context switch occurred and a thread was descheduled. Thread state is saved, thread is moved to the ready queue, and a new thread is chosen to run on the now-free core.
- **Running** $\rightarrow$ **Waiting**: a thread performed a block system call or began waiting for some other thing to happen. Context switch occurs, thread state is saved, the thread is moved to the waiting queue and a new thread is scheduled.
- **Waiting** $\rightarrow$ **Ready**: the blocking event that a thread was waiting for completed, and the thread can now continue to run. Thread is moved from the waiting queue to the ready queue.
- **Ready** $\rightarrow$ **Running**: thread was scheduled, chosen by the kernel to run on a processor core. Thread state is reloaded and thread is removed from the ready queue.

Statistics:

- **6 out of 6** students answered this question.
- **5** was the median score.
- **4.67** was the average score.
- **0.47** was the standard deviation of the scores.

Starting Virtual Address	Bound	Base Physical Address	Permissions
98	100	450	R,W,E
2	40	100	R
1040	1000	10,000	R,W
2200	500	1000	R,W,E

Table 1: **Segmentation Table.**

5. (5 points) Segmentation translation using table.

   Given the 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. load 1200 → load 10,160.
2. store 10 → exception, segment starting at 2 marked as read only. Had the segment been marked writable, would go to 108.
3. load 2080 → exception, no segment loaded for this virtual address. Kernel must either load a segment description into the MMU or kill this process if the address is not valid.
4. fetch 143 → fetch from 495.
5. store 1050 → store to 1010.

Statistics:

- **3 out of 6** students answered this question.
- **5** was the median score.
- **4.67** was the average score.
- **0.47** was the standard deviation of the scores.

6. (5 points) Differentiate between time and space multiplexing, and explain how each concept applies to processor and memory sharing.

Rubric:

Graded by Zihe Chen.

- **+2 points** for explaining the difference.
- **+3 points** for explaining how they apply.

Solution:

Time multiplexing involves sharing a resource by dividing access to it in time. At some time, you allow one process access to it; at a later time, you provide access to some other process. Multiplexing a single-core system is essentially done using time multiplexing.

Space multiplexing involves sharing a resource by dividing it into smaller pieces which are used concurrently. You give part of it to one process, part of it to another, and both may use it simultaneously. Memory is primarily shared using space multiplexing.

Bonus points for pointing out that **both** of these resources actually use a mixture of both space and time multiplexing. Multi-core systems divide cores between threads using space multiplexing. And memory systems change allocations of memory between processes over time, achieving a form of time multiplexing.

Statistics:

- **1 out of 6** students answered this question.
- **3** was the median score.
- **3** was the average score.
- **0.0** was the standard deviation of the scores.

7. (5 points) Your boss at Engitech is an radical egalitarian. You've written some clever code using two threads performing separate tasks that together accomplish your objectives of meeting the customer's every need, but he's concerned about fairness. He's worried that if one of your threads ends up doing too much work, the other thread will feel bad and become discouraged. So he's instructed you to implement a fairness policy so that your two threads must periodically meet at a fixed place in your code. If one thread arrives first, it should wait for the other, and this should work regardless of which thread arrives first. Once both threads meet together, both can continue with their separate tasks.

   Explain how to implement this egalitarian dream world using two semaphores.

Graded by Zihe Chen.

Rubric:

- **+1** for correctly initializing both semaphores.
- **+2** for correct used of `P()` and `V()`.

Solution:

Initialize both semaphores, S1 and S2, to zero. Then, to establish this barrier, at some point in the code for Thread 1 you insert the following:

```
1  // enter and wait
2  P(S1);
3  V(S2);
4  // continue
```

Similarly, at some point in the code for Thread 2 you insert the following:

```
1  // enter and wait
2  V(S1);
3  P(S2);
4  // continue
```

If Thread 1 arrives first, it will block on `S1`; similarly, Thread 2 will block on `S2` if it arrives first. If Thread 1 arrives second, `P(S1)` will not block and `V(S2)` will free Thread 2. If Thread 2 arrives second, `V(S1)` will free Thread 1 and `P(S2)` will not block, or not for long—since Thread 1 will immediately `V()` it once freed.

Statistics:

- **5 out of 6** students answered this question.
- **5** was the median score.
- **4.4** was the average score.
- **0.8** 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. **Starvation v. Utilization in Reader-Writer Locks.** Describe the tension between improving utilization and preventing starvation when implementing reader-writer locks. Outline two *conceptual approaches* (do not write code) to implementing reader-writer locks. For each, explain how it avoids starvation and the effect this has on the utilization of the resource the lock is protecting.

---

    2. **Deadlock.** List the four requirements for deadlock. In class, we solved the dining philosophers problem by eliminating one of the deadlock conditions. Describe two separate approaches to eliminating deadlock for this problem that eliminate other deadlock requirements, as well as any additional kernel support each solution requires over-and-above the primitives that exist or we have asked you to implement for OS/161.

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:

- **14** was the median score.
- **13.33** was the average score.
- **4.11** was the standard deviation of the scores.

Starvation v. Utilization in Reader-Writer Locks.

Rubric:

- **10 points** for discussing starvation and utilization.
  - How do you improve utilization?
  - How might this lead to starvation?
- **10 points** for two approaches.
  - How does the approach work?
  - How does it prevent starvation?
  - What about utilization?

Solution:

Achieving high utilization on a reader-writer lock requires allowing as many reads as possible to proceed in parallel. However, taken to the extreme this would allow prevent writers from making progress as long as their was a steady stream of readers. Preventing this kind of starvation *always* requires stopping the stream of readers and eliminating potential concurrency in order to allow writers (or a writer) to proceed. The question is the conditions under which this is done.

One potential solution to the reader-writer lock problem is to handle requests in the order in which they arrive. Imagine requests arrive in this order, and that each read or write takes one time unit to complete:

$$R_0 \to R_1 \to R_2 \to W_3 \to W_4 \to R_5 \to R_6 \to R_7 \to W_8 \to R_9$$

Then the order-of-arrival solution does the following:

1. $T = 0$: R_0, R_1, R_2.
2. $T = 1$: W_3.
3. $T = 2$: W_4.
4. $T = 3$: R_5, R_6, R_7.
5. $T = 4$: W_8.
6. $T = 5$: R_9.

In this case, this solution has handled 10 requests in 6 time units. However, if we reordered requests notice that we could have handled all 10 requests in only 4 time units. So this solution has sacrificed concurrency and utilization in order to avoid starvation.

A second solution prevents starvation by bounding the amount of time that the lock can remain in read mode while a writer is waiting. When a writer arrives and the lock

is in read mode, a countdown timer is set. Once it goes off, no additional readers are allowed to enter. When the lock clears, the writer can proceed.

In the above example, to make things concrete, the timer might allow R_6, R_7, and R_8 to complete with R_1, R_2 and R_3, even while W_4 is waiting, improving the concurrency by a single time unit. Here the idea is that we are directly trading off writer waiting time to try and increase reader concurrency. Most solutions do this to some extent or another.

Deadlock.

Rubric:

- **10 points** for correctly identifying all four deadlock conditions.
  - 8 points for three.
  - 5 points for two.
  - 3 points for one.
- **10 points** for describing two approaches to eliminating deadlock.
  - How do they work?
  - What additional kernel support is required?

Solution:

The four conditions for deadlock were:

1. **Protected access** to shared resources, which requires waiting.
2. **No resource preemption**, meaning that the system cannot take a resource from a thread holding it.
3. **Multiple independent requests**, meaning a thread can hold some resources while requesting others.
4. **Circular dependency graph**, identified in the directed graph of who's waiting for who.

In class, we relaxed the dependency graph to establish a solution. Here are other ways to do it:

1. **Protected access:** don't let threads sleep waiting on other threads while holding forks/chopsticks! You'd end up with something like this:

```
while (1) {
        // We can wait for the first resource, just not for
        // other resources once we hold one.
        lock_acquire(right_chopstick);

        if (lock_acquire_noblock(left_chopstick) != 0) {

                // We don't hold the lock, so drop the right
                // chopstick and try again.
                lock_release(right_chopstick)
        } else {
                break;
        }
}
```

Here you would clearly need the `lock_acquire_noblock` call which you currently do not have, that would have to peek at the state of the lock and return non-zero if it cannot be acquired.

2. **No resource preemption:** here we need two things. First, we need a way to remove a resource from a thread on the waiting queue, which requires a way to identify what it is waiting for, temporarily remove that resource from it (how?) and then return it later. One (ugly) way to do this would be to have the thread pass a function pointer into `lock_acquire` that could serve as a continuation point if the lock is forcibly revoked, allowing the thread to perform some cleanup.

   Note, however, that this solution also requires *a way to identify deadlock*! Deadlock detectors are non-trivial and probably the more difficult part of this approach. Combined with the difficulty of resource preemption, this is a tough road to take.

3. **Multiple independent requests:** don't grant individual resources, only sets of resources to threads that don't currently hold any. So you need a few changes. First, you need `lock_acquire` to allow you to acquire multiple, potentially many, locks. Second, you need to change the semantics of `lock_acquire` so that it keeps track of threads that hold *any* lock. A second request for *any* lock by a thread currently holding one should fail, since this constitutes a second independent request.