CSE 421/521 Midterm Solutions

—SOLUTION SET—

23 Mar 2016

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) What is GWA's special talent?
   ◯ French ◯ C++ coding √ **Haircut detection** ◯ Haruspication

   (b) Which of the following is a requirement of a critical section?
   √ **progress** ◯ concurrency ◯ mutual inclusion ◯ idleness

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

   (d) Which of the following requires communication with the operating system?
   ◯ Switching between two threads ◯ Inverting a matrix ◯ Recursion
   √ **Creating a new process**

   (e) 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**

   (f) The Rotating Staircase Deadline Scheduler is most similar to which other scheduling algorithm?
   ◯ Lottery scheduling √ **Multi-level feedback queues** ◯ Round-robin
   ◯ Random

   (g) What would probably be stored in a page table entry?
   √ **the physical memory address** ◯ the virtual memory address ◯ the process ID ◯ the file name

   (h) Address translation allows the kernel to implement what abstraction?
   ◯ Files ◯ Threads ◯ Processes √ **Address spaces**

   (i) Con Kolivas was particularly interested in improving what aspect of Linux scheduling?
   ◯ Overhead ◯ Throughput √ **Interactive performance** ◯ Awesomeness

   (j) Which is probably *not* a privileged operation?
   ◯ Changing the interrupt mask ◯ Loading an entry into the TLB ◯ Modifying the exception handlers √ **Adding two registers and placing the result in a third register**

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 (3 points). Second, for each describe a scenario in which that form of waiting is more efficient and why (1 point each).

Rubric:

- **+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.

3. (5 points) Consider a 32-bit system with 4K pages uses multi-level page tables as described in class:

   1. 10 bits for the first-level index,
   2. 10 bits for the second-level index,
   3. and a 12-bit offset.

   (Assume that page table entries (PTEs) are 32-bits and so can be stored directly in the second-level table.)

   If a process has 10 contiguous code pages (including global variables), 4 contiguous heap pages, and a single 1 page stack. What is the minimum number of pages required for the process's page table (1 point)? What is the maximum (2 points)? In both cases, briefly explain your answer (1 point each).

Rubric:

- **+1 points**: for the minimum.
- **+1 point**: for the pseudo-maximum.
- **+2 points**: for the actual maximum.
- **1 point**: for each explanation.

Solution:

Each first- or second-level table consumes one page, since it has 1024 ($2 * *10$) 4-byte entries. So at minimum, this process's page table requires two pages: one of the top-level table and a second for the second-level table if the code, heap, and stack are close enough together (within 4MB) to all lie within the same second-level table. (Note that this is not common and not how we have typically showed processes laid out in class.)

One attempt at calculating the maximum that we will give partial credit for (1 point) gives four: one page for the top-level table plus three pages for the second-level tables, one covering the 10 contiguous code pages, one covering the 4 contiguous heap pages, and a third covering the single stack page. However, the true maximum is 6, since both the the code segment and heap may overlap on to two second-level pages each if they are set up just right (wrong, actually) in the address space. (The stack cannot, since it is only one page.)

Index	SLT
0x0	0x800
0x12	0x10400
0x34	0x32000
0x45	0x10000
0x7f	0x100000

(a) **TLT**

VPN	PPN
0x0	0x3200
0x23	0x1040
0xfd	0x320
0xff	0x1

(b) **SLT at 0x800**

Index	PPN
0x0	0x1048
0xcf	0x7888
0xdf	0x9000

(c) **SLT at 0x100000**

Index	PPN
0x0	0x1048
0x10	0x7888
0x11	0x7888
0x17	0x7888
0xfe	0x9000

(d) **SLT at 0x10040**

Table 1: **Process Page Tables**

4. (5 points) The `HappyShawn` architecture has byte-addressable memory with 64K virtual pages. The `HappyShawnOS` operating system uses two-level pages tables to support 2GB virtual address spaces by dividing the virtual address into a 64K offset and two 8-bit first- and second-level page table indices.

   First, given the virtual address `0x7f001000`, identify the virtual page number, the offset, and the first and second level page table indices (1 point).

   Second, given the top-level page table (TLT) and second-level page tables (SLT) above for the currently-running process provided above, indicate the result of the following four virtual to physical page translations (1 point each).

Solution:

For `0x7f001000`, the virtual page number is `0x7f00`, the offset is `0x1000`, and the first and second level page indices are `0x7f` and `0x0`. Note that while the addresses were in hexadecimal, there was actually no hexadecimal math required to answer this question because the choice of page size and index size split the hexadecimal addresses very cleanly.

Translations follow:

1. `NULL (or 0x0)` → `0x32000000`
2. `0x7fffabcd` → Invalid (no entry in second-level table).
3. `0x22ff0020` → Invalid (no entry in top-level table).
4. `0xff000000` → Invalid (outside 2GB address space, and also missing a top-level entry.)

5. (5 points) Identify one separation between OS mechanism and policy that we have discussed this semester (1 point). Describe the mechanism (2 points) and one policy (2 points).

Rubric:

- **+1 point**: identify the policy-mechanism case.
- **+2 points**: describe the mechanism.
- **+2 points**: describe the policy.

Solution:

One example is the separation between the ability to stop and start threads (mechanism) and scheduling (policy). The mechanism is preemptive context switches using the timer to give the kernel a chance to run regularly and interrupt the running thread if needed. One policy is to schedule threads at random.

Another example is the separation between the virtual address translations performed by the TLB or MMU (mechanism) and the mapping between virtual and physical memory set by the kernel and running process (policy). The mechanism is the TLB's ability to cache translations and map virtual addresses to physical addresses. One policy is having a 2GB virtual address space where the code starts at `0x10000`, the heap starts at `0x10000000` and grows upward, and the stack starts at `0x80000000` and grows down. But any potential address space layout would earn you credit here.

6. (5 points) Identify three system calls that allocate new virtual addresses (i.e., allow the process to use them) and provide a brief description for each.

Rubric:

- **+1 point** for identifying each system call.
- **+1 point** for each system call if properly described.

Solution:

This one came pretty much directly from the lecture slides.

1. `exec()`: loads content from the ELF file and sets up virtual addresses mainly that point to the process's code area and any statically-declared variables.
2. `fork()`: copies the parent's entire address space, including all the code pages, the entire heap, and one (or more) thread stacks, allowing the child to use all of these virtual addresses which will (eventually) have to point to private physical addresses.
3. `sbrk()`: extends the "break point", or the top of the process's heap, allocating new virtual addresses that point to physical memory. Not usually called by processes directly, but instead by `malloc` libraries when their subpage allocator runs out of space.
4. `mmap()`: asks the kernel to map a portion of virtual addresses to point to a region of a file.

7. (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).

Rubric:

- **+2 points** for describing the problem.
- **+3 points** for fixing the problem.

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.

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. **A New Synchronization Primitive.** We have introduced semaphores, spin and sleep locks, condition variables, and reader-writer locks. However, many other useful synchronization primitives exist. First, describe one additional synchronization primitive (4 points). Provide a complete interface for it in C pseudo-code (2 points) and describe how to implement it (4 points).

      Second, provide two different use cases for your new synchronization primitive (5 points each). Feel free to use pseudo-code as well as English to describe why your new primitive is useful.

---

   2. **Scheduling Core Count and Frequency.** We framed the scheduling problem as a question of what threads to run on which cores. However, the modern scheduling problem is much more complicated—particularly on energy-constrained battery-powered devices like laptops and smartphones.

      These devices introduce many new wrinkles. First, cores can be powered off to save energy, since even a completely idle core consumes some amount of power. Second, the speed of each core can be throttled up and down at runtime. Generally, when cores run faster they are *less efficient*—consuming more energy to perform the same set of instructions more quickly. (Third, both these transitions take a non-trivial amount of time, but you can ignore that fact for now.)

      First, describe how these capabilities complicate the original scheduling problem. Identify two new decisions that need to be made (2 points each) and two new tradeoffs that result (3 points each).

      Second, describe a scheduling algorithm for this type of common multicore system (10 points). It can be a variant of one of the algorithms we described in class, or something completely new. However, it should do something intelligent to manage the new capabilities described above. (Put another way, random earns you no credit.)

Solution: A New Synchronization Primitive

There are a bunch of possible answers to this question. Here are two.

The first one is called a *barrier* and is typically used by two threads. Whichever thread arrives at the barrier first waits. Once the second arrives, both continue, and the barrier is destroyed. (You could do a variant with more than two threads, but it's not clear it's that useful.) Here is the interface:

```
/*
 * Create and return a barrier. Note that there is no barrier_destroy, since
 * that is done automatically once the second thread arrives as a safety
 * feature.
 */

struct barrier *barrier_create(const char *name);

/*
 * Wait on a barrier. If the thread is the first to arrive, wait. If it the
 * second to arrive, free the waiting thread, destroy the barrier, and return
 * immediately.
 */

void barrier_wait(struct barrier *);
```

Implementing a barrier is not that difficult. It's quite similar to a semaphore. For a binary barrier we simply set a flag during the first `barrier_wait` and put the thread on a wait channel. When the second thread arrives, we clear the flag, release the first thread, and destroy the structure. We use a spin lock to synchronize the internal barrier state and bridge to the wait channel. (Technically a wait channel is overkill for a binary barrier since there is only at most one thread ever waiting.) For a counting barrier we keep a count of the number of threads that have called `barrier_wait` and wake all once the count is reached.

Barriers are commonly used to synchronize initialization between two threads. As one example, in `fork` the parent may want to ensure that it completes some initialization before the child exits, but the child may be able to do some work safely before reaching the barrier. So whatever the child can do before the parent runs goes above the barrier, and the parts that require the parent to run go below it. The parts that the parent has to complete go above the barrier. If the child hits the barrier first, it waits. If the parent hits the barrier first, it waits. Once both arrive, both continue.

A second use case for a counting barrier occurs during the initialization of code in our test suites. Frequently we initialize a bunch of threads which can do some initialization but need to wait for the menu thread to finish creating the other test runners before continuing. A counting barrier allows the menu to create all of the threads and then free them all at once once they have all finished their initialization. (Note that the menu thread may not be the one to free them, since it may run ahead of some of the threads initialization routines.)

The second one is a synchronized channel—a primitive that combines synchronization and communication in a nice way. (And is one of the building blocks of synchronization in Go.) Ideally a channel is typed, but let's imagine that we create one limited to sending 4-byte values (maybe pointers). Threads can both send and receive from a channel. The channel has a capacity, and if the channel is full (or empty) the channel will block until another thread calls receive (or send). Here is the interface:

```
/*
 * Create and return a channel with the given capacity. Note that zero is a
 * valid capacity and will cause all sends and receives on the channel to be
 * blocking until the complementary operation has completed.
 */

struct channel *channel_create(int capacity, const char *name);

/* Destroy the channel. It should be empty. */

struct channel *channel_destroy(const char *name);

/*
 * Send the void * pointer on the channel. If the channel is full, the send
 * blocks until channel_receive is called. Otherwise it returns immediately.
 */

void channel_send(struct channel *, void *);

/*
 * Receive a void * pointer from the channel. If the channel is empty, the
 * receive blocks until channel_send is called. Otherwise it returns immediately.
 */

void * channel_receive(struct channel *);
```

Implementing a channel essentially reduces to the bounded buffer problem that we discussed in class, so I won't describe it further here. The only additional complication is storing the passed `void *` when a bunch of threads are blocked on a full channel. So the implementation may need some flexible storage to handle this case. Or it can fail `channel_send` calls once some maximum is reached.

Channels are useful whenever two threads need to communicate information in a synchronized fashion. Returning to the `fork` example from ASST2, a channel can be used both to achieve the synchronization required and send the trapframe pointer from the parent to the child. Buffered channels can be used as a general-purpose communication mechanisms that frees the communicating threads from worrying about the case where they have no work to do, since they will sleep until they do.

Solution: Scheduling Core Count and Frequency

The first part of the question is fairly easy to answer. Originally we have to choose what threads should be run. Now we have to choose what cores should be running and what frequency they should be running at. In both cases, the tradeoff is speed versus energy. Having more cores online allows more tasks to proceed in parallel, but is also useful in the case where we actually have applications (or tasks) that can take advantage of concurrency. Adjusting the frequency presents the same performance v. energy consumption tradeoff. Ramping up the frequency allows us to complete work more quickly, but causes the same work to consume more energy than if we had done it more slowly.

We will accept a variety of different scheduling algorithms. Here's one idea that extends MLFQ. We map the MLFQ levels to CPU frequencies, allowing threads in the top queues to run at faster frequencies. Since these threads are (in theory) more likely to be interactive, this makes sense: allowing them to run more quickly lets them finish their work, keeps the user from waiting, and allows the system to go back to sleep—at which point the CPU frequency can be reduced or cores may even be able to be powered down.

To choose the number of cores to have active we examine the number of runnable threads in the top few queues. For example, if we have four cores and 6 threads runnable in the top queue (or second-level queue) we activate all four cores. Otherwise, we cap the number of cores at three and examine lower queues. We can apply this approach all the way down the queues, scaling things so that threads in the bottom queue must run on one core. This slows them down, since they have to run back-to-back rather than in parallel, but also allows the system to save energy.