University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 2015 John Kubiatowicz

# Midterm I

March 11th, 2015
CS162: Operating Systems and Systems Programming

| | |
|---|---|
| Your Name: | |
| SID Number: | |
| Discussion Section: | |

General Information:
This is a **closed book** exam. You are allowed 1 page of **hand-written** notes (both sides). You have 3 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

| **Problem** | **Possible** | **Score** |
|---|---|---|
| 1 | 18 | |
| 2 | 18 | |
| 3 | 24 | |
| 4 | 20 | |
| 5 | 20 | |
| Total | 100 | |

[ This page left for π ]

3.14159265358979323846264338327950288419716939937510582097494 4

# Problem 1: TRUE/FALSE [18 pts]

In the following, it is important that you *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT*.

**Problem 1a[2pts]:** The kernel on a multiprocessor can use the local disabling of interrupts (within one CPU) to produce critical sections between the OSs on different CPUs.

True / False
Explain:

**Problem 1b[2pts]:** Simultaneous Multithreading is a hardware mechanism that can switch threads every cycle.

True / False
Explain:

**Problem 1c[2pts]:** In a multi-process HTTP server (like in HW#2), only the child process is responsible for closing the client socket (e.g. the file descriptor returned by `accept()`), since the parent doesn't know when the child is done using the socket.

True / False
Explain:

**Problem 1d[2pts]:** A user-level library implements each system call by first executing a "transition to kernel mode" instruction. The library routine then calls an appropriate subroutine in the kernel.

True / False
Explain:

**Problem 1e[2pts]:** A thread can be blocked on multiple condition variables simultaneously.

True / False
Explain:

**Problem 1f[2pts]:** Floating point numbers are not used in Pintos because floating point operations are too slow and have rounding issues.

True / False
Explain:

**Problem 1g[2pts]:** In Pintos, implementing priority scheduling for semaphores will also take care of priority scheduling for locks and condition variables. This is because locks and condition variables are implemented using semaphores.

True / False
Explain:

**Problem 1h[2pts]:** The only way to resolve a resource deadlock is to reboot the system.

True / False
Explain:

**Problem 1i[2pts]:** Calls to `printf()` always enter the kernel to perform an output to `stdout`.

True / False
Explain:

# Problem 2: Short Answer [18pts]

**Problem 2a[3pts]:** Name at least two disadvantages to using interrupts to serialize access to a critical section. When does it make sense to use interrupt disable/enable around a critical section?

**Problem 2b[2pts]:** What is the difference between Mesa and Hoare scheduling for monitors? How does this affect the programming pattern used by programmers (be explicit)?

**Problem 2c[2pts]:** What needs to be saved and restored on a context switch between two threads in the same process? What if we have two different processes?

**Problem 2d[3pts]:** Name three ways in which the processor can transition from user mode to kernel mode. Can the user execute arbitrary code after the transition?

**Problem 2e[2pts]:** What is the difference between fork() and exec() on Unix?

**Problem 2f[2pts]:** List two reasons why overuse of threads is bad (i.e. using too many threads for different tasks). Be explicit in your answers.

**Problem 2g[2pts]:** What is the default scheduler in PintOS?

**Problem 2h[2pts]:** In PintOS, the code for `thread_unblock()` contains a comment that says "This function does not preempt the running thread". Explain why you should not modify `thread_unblock()` in a way that could cause it to preempt the running thread.

# Problem 3: Atomic Synchronization Primitives [24pts]

In class, we discussed a number of *atomic* hardware primitives that are available on modern architectures. In particular, we discussed "test and set" (TSET), SWAP, and "compare and swap" (CAS). They can be defined as follows (let "expr" be an expression, "&addr" be an address of a memory location, and "M[addr]" be the actual memory location at address addr):

| Test and Set (TSET) | Atomic Swap (SWAP) | Compare and Swap (CAS) |
|---|---|---|
| TSET(&addr) {<br>  int result = M[addr];<br>  M[addr] = 1;<br>  return (result);<br>} | SWAP(&addr, expr) {<br>  int result = M[addr];<br>  M[addr] = expr;<br>  return (result);<br>} | CAS(&addr, expr1, expr2) {<br>  if (M[addr] == expr1) {<br>  M[addr] = expr2;<br>  return true;<br>  } else {<br>  return false;<br>  }<br>} |

Both TSET and SWAP return values (from memory), whereas CAS returns either true or false. Note that our &addr notation is similar to a reference in c++, and means that the &addr argument must be something that can be stored into (an "lvalue"). For instance, TSET could be used to implement a spin-lock acquire as follows:

```
int lock = 0; // lock is free

// Later: acquire lock
while (TSET(lock));
```

CAS is general enough as an atomic operation that it can be used to implement both TSET and SWAP. For instance, consider the following implementation of TSET with CAS:

```
TSET(&addr) {
   int temp;
   do {
      temp = M[addr];
   } while (!CAS(addr,temp,1));
   return temp;
}
```

**Problem 3a[3pts]:**
Show how to implement a spinlock acquire with a single while loop using CAS instead of TSET. You must only fill in the arguments to CAS below:

```
// Initialization
int lock = 0;  // Lock is free


// acquire lock

while (  !CAS(          ,          ,          )   );
```

**Problem 3b[2pts]:**
Show how SWAP can be implemented using CAS. Don't forget the return value.

```
SWAP(&addr, reg1) {









}
```

**Problem 3c[2pts]:**
With spinlocks, threads spin in a loop (busy waiting) until the lock is freed. In class we argued that spinlocks were a bad idea because they can waste a lot of processor cycles. The alternative is to put a waiting process to sleep while it is waiting for the lock (using a blocking lock). Contrary to what we implied in class, there are cases in which spinlocks would be more efficient than blocking locks. Give a circumstance in which this is true and explain why a spinlock is more efficient.

An object such as a queue is considered "lock-free" if multiple processes can operate on this object simultaneously without requiring the use of locks, busy-waiting, or sleeping. In this problem, we are going to construct a lock-free FIFO queue using the atomic CAS operation. This queue needs both an `Enqueue` and `Dequeue` method.

We are going to do this in a slightly different way than normally. Rather than `Head` and `Tail` pointers, we are going to have "`PrevHead`" and `Tail` pointers. `PrevHead` will point at the last object returned from the queue. Thus, we can find the head of the queue (for dequeuing). If we don't have to worry about simultaneous Enqueue or Dequeue operations, the code is straightforward:

```
// Holding cell for an entry
class QueueEntry {
     QueueEntry next = null;
     Object stored;

     QueueEntry(Object newobject) {
          stored = newobject;
     }
}

// The actual Queue (not yet lock free!)
class Queue {
     QueueEntry prevHead = new QueueEntry(null);
     QueueEntry tail = prevHead;

     void Enqueue(Object newobject)  {
          QueueEntry newEntry = new QueueEntry(newobject);
          QueneEntry oldtail = tail;
          tail = newEntry;
          oldtail.next = newEntry;
     }

     Object Dequeue() {
          QueueEntry oldprevHead = prevHead;
          QueueEntry nextEntry = oldprevHead.next;
          if (nextEntry == null)
               return null;
          prevHead = nextEntry;
          return nextEntry.stored;
     }
}
```

**Problem 3d[3pts]:**
For this non-multithreaded code, draw the state of a queue with 2 queued items on it:

**Problem 3e[3pts]:** For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior of Enqueue(); Explain!

```
        void Enqueue(Object newobject)  {
1 ----> QueueEntry newEntry = new QueueEntry(newobject);
2 ----> QueueEntry oldtail = tail;
3 ----> tail = newEntry;
            oldtail.next = newEntry;
        }
```

Point 1:

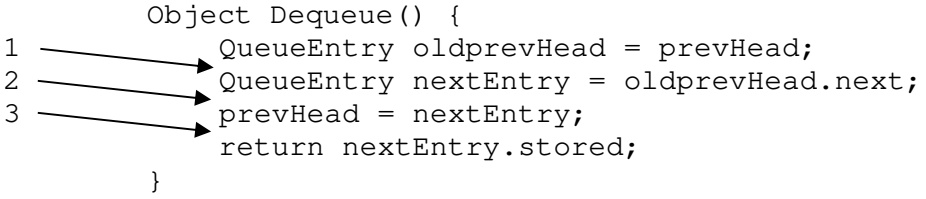
Point 2:

Point 3:

**Problem 3f[4pts]:** Rewrite code for Enqueue(), using the CAS() operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** The solution is tricky but can be done in a few lines. We will be grading on conciseness. Do not use more than one CAS() or more than 10 lines total (including the function declaration at the beginning). *Hint: wrap a do-while around vulnerable parts of the code identified above.*

```
    void Enqueue(Object newobject)  {
        QueueEntry newEntry = new QueueEntry(newobject);

        // Insert code here









    }
```

**Problem 3g[3pts]:** For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior of Dequeue(); Explain! (Note: *Assume that the queue is not empty when answering this question, since we have removed the null-queue check from the original code):*




Point 1:

Point 2:

Point 3:

**Problem 3h[4pts]:** Rewrite code for Dequeue(), using the CAS() operation, such that it will work for any number of simultaneous Enqueue and Dequeue operations. You should never need to busy wait. **Do not use locking (i.e. don't use a test-and-set lock).** The solution can be done in a few lines. We will be grading on conciseness. Do not use more than one CAS() or more than 10 lines total (including the function declaration at the beginning). You should correctly handle an empty queue by returning "null". *Hint: wrap a do-while around vulnerable parts of the code identified above and add back the null-check from the original code.*

```
Object Dequeue() {

    // Insert code here









}
```

# Problem 4: Scheduling and Deadlock [20 pts]

**Problem 4a[2pts]:** How could a priority scheduler be used to emulate Earliest Deadline First (EDF) scheduling? Would computing of priorities be an expensive operation (assume that we schedule periodic tasks characterized by period T and computational time of C)? Explain.

**Problem 4b[2pts]:** What is a multi-level feedback scheduler and how can it approximate SRTF?

**Problem 4c[3pts]:** What is priority donation? What sort of information must the OS track to allow it to perform priority donation? Is priority donation targeted at preventing a deadlock or a livelock?

**Problem 4d[3pts]:** Suppose that you utilize a scheme that schedules threads within a process at user level. Why might a naïve scheduling scheme run into problems when accessing I/O? Can the operating system help resolve this problem? Explain

Pwnage Games, a fairly unknown arcade in Downtown Berkeley, decided to purchase Super Smash Bros. for Wii U -- a popular fighting video game -- in the hope that it would draw customers to the business. However, due to limited resources, the store could only buy one copy of the game. Luckily, the owners know Gill Bates -- a Cal EECS undergrad -- who offers her help in exchange for free arcade credits. Her job is to allow multiple consoles to play the game at the same time. Thanks to her hacking skills, Gill completes the task in no time, but she is forced to impose some conditions on the gameplay:

- each console only allows for two players to fight at a time;
- the same character cannot be used by more than one player at a time.

The enforcement of these conditions is handled after character selection. That is, all fighters appear available at all times, and the following function loads the fight. Each character has a global fighter_t* representing it across consoles.

```
void smash (fighter_t* first, fighter_t* second)
{
   pthread_mutex_lock (&first->lock);
   pthread_mutex_lock (&second->lock);
   fight (first, second);
   pthread_mutex_unlock (&second->lock);
   pthread_mutex_unlock (&first->lock);

}
```

**Problem 4e[4pts]:** Despite Gill's effort, her algorithm has an obvious flaw: it can lead to deadlock! Present an example of how this can happen. List the four conditions for deadlock and show how they are satisfied by this example:

**Problem 4f[3pts]:** Redesign the `smash()` function to avoid deadlock. Write your new version in the space below. Which of the four conditions are now missing? Name one downside of your approach.

**Problem 4g[3pts]:** Explain how the Banker's algorithm could prevent the deadlock identified in Problem (4e) and what changes would need to be made to the code to support it. Clearly identify the behavior that would result, and why the four conditions for deadlock are not simultaneously satisfied. Would this solution be better or worse than your solution to Problem (4f)?

[ This page intentionally left blank ]

# Problem 5: Address Translation [20 pts]

Consider a multi-level memory management scheme with the following format for virtual addresses:

| Virtual Page # (10 bits) | Virtual Page # (10 bits) | Offset (12 bits) |
|---|---|---|

Virtual addresses are translated into physical addresses of the following form:

| Physical Page # (20 bits) | Offset (12 bits) |
|---|---|

Page table entries (PTE) are 32 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory):

| Physical Page # (20 bits) | OS Defined (3 bits) | 0 | Large Page | Dirty | Accessed | Nocache | Write Through | User | Writeable | Valid |
|---|---|---|---|---|---|---|---|---|---|---|

Here, "Valid" means that a translation is valid, "Writeable" means that the page is writeable, "User" means that the page is accessible by the User (rather than only by the Kernel). *Note: the phrase "page table" in the following questions means the multi-level data structure that maps virtual addresses to physical addresses.*

**Problem 5a[2pts]:** How big is a page? Explain.

**Problem 5b[4pts]:** Draw a picture of the page table. What good property(s) result from dividing the address into three fields in this way (i.e. 32 bits = 10bits + 10bits + 12bits)?

**Problem 5c[2pts]:** Suppose that we want an address space with one physical page at the top of the address space and one physical page at the bottom of the address space. How big would the page table be (in bytes)? Explain.

**Problem 5d[2pts]:** What is the maximum amount of physical memory that can be addressed by this page table. Explain.

**Problem 5e[10pts]:** Assume the memory translation scheme from (5a). Use the Physical Memory table given on the next page to predict what will happen with the following load/store instructions. Assume that the base table pointer for the current ***user level process*** is `0x00200000`.

Addresses in the "Instruction" column are virtual. You should translate these addresses to physical address (i.e. in middle column), then attempt to execute the specified instruction on the resulting address. The return value for a load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. Possible errors are: **invalid, read-only, kernel-only.**
*Hints: (1) Don't forget that Hexidecimal digits contain 4 bits! (2) PTEs are 4 bytes!*

| Instruction | Physical Address | Result |
|---|---|---|
| Load [0x00001047] | 0x00002047 | 0x50 |
| Store [0x00C07665] | 0xEEFF0655 | ok |
| Store [0x00C005FF] | 0x112205FF | ERROR: read-only |
| Load [0x00003012] | | |
| Store [0x02001345] | | |
| Load [0xFF80078F] | | |
| Load [0xFFFFF005] | | |
| Test-And-Set [0xFFFFF006] | | |

## Physical Memory [All Values are in Hexidecimal]

| Address | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | +8 | +9 | +A | +B | +C | +D | +E | +F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 0E | 0F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D |
| 00000010 | 1E | 1F | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 2A | 2B | 2C | 2D |
| …. | | | | | | | | | | | | | | | | |
| 00001010 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F |
| 00001020 | 40 | 03 | 41 | 01 | 30 | 01 | 31 | 03 | 00 | 03 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00001030 | 00 | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | AA | BB | CC | DD | EE | FF |
| 00001040 | 10 | 01 | 11 | 03 | 31 | 03 | 13 | 00 | 14 | 01 | 15 | 03 | 16 | 01 | 17 | 00 |
| …. | | | | | | | | | | | | | | | | |
| 00002030 | 10 | 01 | 11 | 00 | 12 | 03 | 67 | 03 | 11 | 03 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00002040 | 02 | 20 | 03 | 30 | 04 | 40 | 05 | 50 | 01 | 60 | 03 | 70 | 08 | 80 | 09 | 90 |
| 00002050 | 10 | 00 | 31 | 01 | 10 | 03 | 31 | 01 | 12 | 03 | 30 | 00 | 10 | 00 | 10 | 01 |
| …. | | | | | | | | | | | | | | | | |
| 00004000 | 30 | 00 | 31 | 01 | 11 | 01 | 33 | 03 | 34 | 01 | 35 | 00 | 43 | 38 | 32 | 79 |
| 00004010 | 50 | 28 | 36 | 19 | 71 | 69 | 39 | 93 | 75 | 10 | 58 | 20 | 97 | 49 | 44 | 59 |
| 00004020 | 23 | 03 | 20 | 03 | 00 | 01 | 62 | 08 | 99 | 86 | 28 | 03 | 48 | 25 | 34 | 21 |
| …. | | | | | | | | | | | | | | | | |
| 00100000 | 00 | 00 | 10 | 67 | 00 | 00 | 20 | 67 | 00 | 00 | 30 | 00 | 00 | 00 | 40 | 07 |
| 00100010 | 00 | 00 | 50 | 03 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| … | | | | | | | | | | | | | | | | |
| 00103000 | 11 | 22 | 00 | 05 | 55 | 66 | 77 | 88 | 99 | AA | BB | CC | DD | EE | FF | 00 |
| 00103010 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | AA | BB | CC | DD | EE | FF | 00 | 67 |
| … | | | | | | | | | | | | | | | | |
| 001FE000 | 04 | 15 | 00 | 00 | 48 | 59 | 70 | 7B | 8C | 9D | AE | BF | D0 | E1 | F2 | 03 |
| 001FE010 | 10 | 15 | 00 | 67 | 10 | 15 | 10 | 67 | 10 | 15 | 20 | 67 | 10 | 15 | 30 | 67 |
| … | | | | | | | | | | | | | | | | |
| 001FF000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 65 | 00 | 00 | 10 | 67 | 00 | 00 | 00 | 00 |
| 001FF010 | 00 | 00 | 20 | 67 | 00 | 00 | 30 | 67 | 00 | 00 | 40 | 65 | 00 | 00 | 50 | 07 |
| … | | | | | | | | | | | | | | | | |
| 001FFFF0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 10 | 00 | 00 | 67 | 00 | 10 | 30 | 67 |
| … | | | | | | | | | | | | | | | | |
| 00200000 | 00 | 10 | 00 | 07 | 00 | 10 | 10 | 07 | 00 | 10 | 20 | 07 | 00 | 10 | 30 | 07 |
| 00200010 | 00 | 10 | 40 | 07 | 00 | 10 | 50 | 07 | 00 | 10 | 60 | 07 | 00 | 10 | 70 | 07 |
| 00200020 | 00 | 10 | 00 | 07 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| … | | | | | | | | | | | | | | | | |
| 00200FF0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 1F | E0 | 07 | 00 | 1F | F0 | 07 |
| … | | | | | | | | | | | | | | | | |

[ This page intentionally left blank ]

[ This page left for scratch ]