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

Spring 2019 John Kubiatowicz

Midterm I
SOLUTIONS

February 28th, 2019
CS162: Operating Systems and Systems Programming

Your Name:	
SID AND 162 Login (e.g. s042):	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 1 page of notes (both sides). You may use a calculator. You have 2 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	21	
3	27	
4	21	
5	13	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [18 pts]

Please *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]: If we disable interrupts in Pintos, the only way for another thread to run is for the current thread to die and exit, triggering the scheduler to choose another thread to run.

☐ True ☑ False

Explain: *There are a number of ways for a different thread to take over, including* `thread_yield()` *as well as a variety of blocking system calls (e.g.* `read()`*).*

Problem 1b[2pts]: Threads within the same process can share data with one another by passing pointers to objects on their stacks.

☑ True ☐ False

Explain: *Threads in the same process share an address space and can communicate via shared-memory (reading and writing to shared addresses). Thus, objects allocated on the stack for one thread have addresses that can be forwarded to other threads as pointers.*

Problem 1c[2pts]: After a forked child process calls exit(33), the child process will assign its exit code to the parent's wait status reference (int *status) via the code *status = exit_code (where exit_code is 33 in this case).

☐ True ☑ False

Explain: *The child process is in a separate address space from the parent and has no access to the "status" variable in the parent's address space. The OS returns the child's exit status to the parent when the parent calls* `wait()`*.*

Problem 1d[2pts]: A system call (such as `read()`) is implemented in the C library as a function which does two separate things: (1) executes a "transition to kernel mode" instruction that changes the processor from user level to kernel level, followed by (2) a call to the correct routine in the kernel. This process is secure because the C library involves standard code that has been audited by the designers of the kernel.

☐ True ☑ False

Explain: *Security of system calls is not ensured by the C library. There is no guarantee that any particular C library was linked into a program. Instead, entry into the kernel is through a simultaneous (atomic) process that transitions to kernel mode and vectors to a pre-determined handler in the kernel based on the system call number. The Kernel handler is then responsible for security.*

Problem 1e[2pts]: Immediately after a process has been forked, the same variable in both the parent and the child will have the same virtual memory address but different physical memory addresses.

☐ True ☑ False

Explain: *Initially, the child and parent have the same physical memory mapped into their address spaces. The page tables for this memory are marked as read-only so that a copy-on-write process will create distinct physical copies only for pages that differ between parent and child.*

Problem 1f[2pts]:."Hyperthreading" refers to the situation in which a modern operating system allows thousands of threads to access the same address space.

☐ True ☑ False

Explain: *Hypertheading refers to the functionality (usually on x86 processors) that allows multiple physical threads (with distinct registers, program counters, etc) to share the functional units in a single pipeline.*

Problem 1g[2pts]: Semaphores that operate across multiple processors (and at user level) can be implemented on machines in which the only atomic instruction operations on memory are loads, stores, and test-and-set.

☑ True ☐ False

Explain: *Similar to the alternate lock implementation given in class (using test-and-set on a guard variable), one could use test-and-set as a guard on a semaphore implementation.*

Problem 1h[2pts]: Because the "monitor" pattern of synchronization involves sleeping inside a critical section, it requires special hardware support to avoid deadlock (i.e. permanent lack of forward progress).

☐ True ☑ False

Explain: *The `wait()` operation on a condition variable atomically releases the lock and puts the process to sleep, thereby freeing up the lock to be acquired by a different thread that will `signal()` the sleeping thread. Then, before `wait()` returns to the original thread, the lock is reacquired. None of this requires special hardware.*

Problem 1i[2pts]: Thread pools are a useful tool to help prevent an overly popular web site from crashing the hosting web servers.

☑ True ☐ False

Explain: *During a flash-crowd situation, a naïve server solution would simply launch a new thread (or worse, process) for every new connection, which could over flow memory and overwhelm the OS. Thread pools can be used to limit the maximum number of threads in a process.*

Problem 2: Short Answer [21pts]

Problem 2a[3pts]: What is priority donation and why is it important?

Priority donation is a mechanism by which the scheduling priority of a high-priority process attempting to acquire a lock held by a lower-priority process is donated to the lower-priority process. Priority donation helps to prevent a priority-inversion that violates the design constraints of a program by allowing a low-priority process to block a high-priority process.

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

There are many ways that a processor can transition from user mode to kernel mode. Here are several:

a. *The user code can execute a system call.*
b. *The user code can perform an exception such as dividing by zero.*
c. *A protection violation such as a page-fault could occur*
d. *An local event such as a timer interrupt might occur*
e. *An external event such as a network message arrival could generate an interrupt*
f. *A hardware failure event such as an error correction fault in memory*

No, the user cannot execute arbitrary code in the kernel, since all entry into the kernel goes through explicit vector tables to well-defined entry points (handlers).

Problem 2c[3pts]: What happens when an interrupt occurs? What does the interrupt controller do?

If an interrupt is disabled, then nothing happens. Assuming that the interrupt is enabled, then the processor hardware disables all interrupts, saves critical state, switches to kernel mode and then dispatches to a kernel handler based on the type of interrupt. Either the hardware or handler switches to a kernel stack. The interrupt controller allows the kernel to control which interrupts are enabled or disabled at any given point in time.

Problem 2d[2pts]: Processes (or threads) can be in one of three states: **Running, Ready,** or **Waiting**. For each of the following examples, write down which state the process (or thread) is in:

a. Spin-waiting for a variable to become zero:
*Process is **Running** (Busy waiting!)*

b. Having just completed an I/O, waiting to get scheduled again on the CPU:
*Process is **Ready** (schedulable but not actually running**)***

c. Blocking on a condition variable waiting for some other thread to signal it:
*Process is **Waiting** (linked on a wait queue somewhere)*

d. Scanning through the buffer cache within the kernel in response to `read()`:
*Process is **Running** (Executing kernel code!)*

Problem 2e[3pts]: Disabling of interrupts can be very dangerous if done incorrectly. Is it possible to design an OS for modern computer hardware that never disables interrupts? Why or why not?

The answer is "NO". Since interrupts stop the processor and redirect it to do other things, one cannot avoid disabling interrupts in some parts of the kernel to achieve correctness. Examples include handlers for other interrupts and periods of time when the kernel is manipulating data structures that are touched by interrupt handlers. Note that most modern computer hardware delivers events using interrupts to achieve timeliness; consequently interrupts occur frequently (especial under high-usage situations) and would practically guarantee that race conditions would occur.

Although not allowed by the question, if one were to change the hardware so that it never invoked interrupts (so that interrupts were effectively always disabled), then an OS might poll periodically to check for pending hardware events. While the code could be made to be correct (no race conditions), this design pattern would prevent preemptive scheduling as well as timely delivery of events – the whole point of having interrupts in the first place.

Problem 2f[3pts]: What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? Be explicit.

All of the execution state – registers, program counters, stack pointers, etc – must be saved and restored on a context switch between threads in the same process. In addition, when switching between threads in different processes, the memory protection state must be switched (i.e. change the active page tables). Process-specific state such as the PCB, file-descriptor table, etc, is usually switched automatically as a side-effect of changing out the address space and registers.

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

There are a number of reasons, including:

a. *The overhead of switching between threads (scheduling, register saving and restoring, etc) can overwhelm the computation*
b. *Threads may overuse kernel resources such as memory, thereby making these resources unavailable for other things, such as file caching or virtual memory*
c. *Hardware resources such as processor caches are inefficiently utilized because they are spread too thin across too many threads.*
d. *If threads are being used for parallelism, the cost of synchronizing between them can cause overhead by forcing threads to wait a long time (think about barriers, for instance)*

Problem 2h[2pts]: Why is it possible for a web browser (such as Firefox) to have 2 different tabs opened to the same website (at the same remote IP address and port) without mixing up content directed at each tab?

Because TCP/IP connections are uniquely defined by a five-tuple of information: two IP addresses, two ports, and a protocol (i.e. TCP). Connections between the two tabs and the web server would be distinguished by the local ports chosen for the connection.

Problem 3: Synchronization Primitives [27pts]

Assume that you are programming a multiprocessor system using threads. In class, we talked about two different synchronization primitives: Semaphores and Monitors.

The interface for a Semaphore is similar to the following:

```
struct sem_t {
   // internal fields
};
void sem_init(sem_t *sem, unsigned int value) {
   // Initialize semaphore with initial value
   …
}
void sem_P(sem_t *sem) {
   // Perform P() operation on the semaphore
   …
}
void sem_V(sem_t *sem) {
   // Perform V() operation on the semaphore
}
```

As we mentioned in class, a Monitor consists of a Lock and one or more Condition Variables. The interfaces for these two types of objects are as follows:

```
struct lock_t {
   // Internal fields
};
void lock_init(lock_t *lock) {
   // Initialize new lock
   …
}

void acquire(lock_t *lock) {
   // acquire lock
   …
}
void release(lock_t *lock) {
   // release lock
   …
}
```

```
struct cond_t {
   // Internal fields
};
void cond_init(cond_t *cv, lock_t *lock) {
   // Initialize new condition variable
   // associated with lock.
   …
}
void cond_wait(cond_t *cv) {
   // block on condition variable
   …
}
void cond_signal(cond_t *cv) {
   // wake one sleeping thread (if any)
   …
}
void cond_broadcast(cond_t *cv) {
   // wake up all threads waiting on cv
   …
}
```

Monitors and Semaphores can be used for a variety of things. In fact, each can be implemented with the other. In this problem, we will show their equivalence.

Problem 3a[2pts]: What is the difference between Mesa and Hoare scheduling for monitors?

Mesa: signaler keeps lock and processor; waiter placed on ready queue and does not run Immediately

Hoare: signal gives lock and CPU to waiter; waiter runs immediately; waiter gives lock and processor back to signaler when it exits critical section or waits again.

Problem 3b[6pts]: Show how to implement Semaphores using Monitors (i.e. the `lock_t` and `cond_t` operations). Make sure to define `sem_t` and implement all three methods, `sem_init()`, `sem_P()`, and `sem_V()`. *None of the methods should require more than five lines.* Assume that Monitors are Mesa scheduled.

```
struct sem_t {
   lock_t my_lock;
   cond_t my_cond;
   unsigned int my_value;
};
void sem_init(sem_t *sem, unsigned int value) {
   lock_init(&(sem->my_lock));
   cond_init(&(sem->my_cond), &(sem->my_lock));
   sem->my_value = value;
}
void sem_P(sem_t *sem) {
   acquire(&(sem->my_lock));
   while (sem->my_value == 0)
      cond_wait(&(sem->my_cond));
   sem->my_value--;
   release(&(sem->my_lock));
}
void sem_V(sem_t *sem) {
   acquire(&(sem->my_lock));
   sem->my_value++;
   cond_signal(sem->my_cond);
   release(&(sem->my_lock));
}
```

Problem 3c[4pts]: Show how to implement the Locks using Semaphores (i.e. the sem_t operations). Make sure to define `lock_t` and implement all three methods, `lock_init()`, `acquire()`, and `release()`. None of the methods should require more than five lines.

```
struct lock_t {
   sem_t my_sem;


};
void lock_init(lock_t *lock) {
   sem_init(&(lock->my_sem),1);



}
void acquire(lock_t *lock) {
   sem_P(&(lock->my_sem));



}
void release(lock_t *lock) {
   sem_V(&(lock->my_sem));



}
```

Problem 3d[2pts]: Explain the difference in behavior between `sem_V()` and `cond_signal()` when no threads are waiting in the corresponding semaphore or condition variable:

`sem_V()` increments the semaphore, while `cond_signal()` does nothing.

Problem 3e[10pts]: Show how to implement the Condition Variable class using Semaphores (and your Lock class from 3c). Assume that you are providing Mesa scheduling. You should not use lists, queues, or arrays for this solution. Be very careful to consider the semantics of `cond_signal()` as discussed in (3d). *Hint: the Semaphore interface does not allow querying of the size of its waiting queue; you may need to track this yourself.* None of the methods should require more than five lines.

```
struct cond_t {
   lock_t *my_lock;
   sem_t my_sem;
   int queue_length; // Queue length of my_sem for signal.

};
void cond_init(cond_t *cv, lock_t *lock) {
   cv->my_lock = lock;
   sem_init(&(cv->my_sem),0);
   queue_length=0;


}
void cond_wait(cond_t *cv) {
   // We know we are in critical section.  Update queue_length before release
   queue_length++;

   // release lock, wait, reacquire lock
   // Even if another thread gets between release and sem_P, this works
   // since queue_length > 0 => any signal or broadcast will sem_V and
   // wake us up properly (i.e. sem_P will fall through immediately)
   release(cv->my_lock);
   sem_P(&(cv->my_sem));
   acquire(cv->my_lock);
}
void cond_signal(cond_t *cv) {
   // We are in critical section!
   if (queue_length > 0) {
      sem_V(&(cv->my_sem));
      queue_length--;
   }

}
void cond_broadcast(cond_t *cv) {
   // We are in critical section!
   while (queue_length > 0) {
      sem_V(&(cv->my_sem));
      queue_length--;
   }

}
```

Problem 3f[3pts]: Suppose that we implement locks using test-and-set and that we want to avoid long-term spin-waiting. Also assume that we use these locks at user level to synchronize across multiple processors. Explain why we still need to provide support for our lock implementation in the kernel and provide an interface for any system calls that might be needed.

Just as in class, we would use test-and-set to guard the lock implementation (so that spin-waiting is minimized). Both the guard variable and lock variable would be integers in memory and accessible by user code. However, we still need a way to put a thread to sleep and atomically release the guard if it is unable to acquire a lock. Further, we need an interface to wake up a sleeping thread (if it exists). Both of these operations interact with the kernel scheduler queues (e.g. ready) and must disable interrupts to work properly – such operations can only occur within the kernel.

Assume that the lock variable address is used by the kernel to define a sleep queue. Then our simple interface would involve two system calls like this:

```
/*
 * Atomically perform *guard_address=0 and put current thread to
 * sleep on queue associated with lock_address.
 */
void  sleep_and_release(int *guard_address, int *lock_address);

/*
 * Wake up one sleeping thread from sleep queue associated
 * with lock_address (if present).
 */
void wake_one(int *lock_address);
```

This solution assumes that creation of the kernel-level lock queue happens automatically the first time that the kernel sees the lock_address. The assumption is simply that the kernel uses the unique lock_address value to look up the actual wait queue in kernel memory.

NOTE: this interface is similar to but not quite the same as the Linux Futex interface! Use of Futex can be much more optimal. We encourage you to look it up.

Problem 4: Syscall Potpourri (21pts)

Problem 4a[4pts]: Assume that we have the following piece of code:

```
1.  int main() {
2.     printf("Starting main\n");
3.     int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
4.     dup2(file_fd, STDOUT_FILENO);
5.     pid_t child_pid = fork();
6.     if (child_pid != 0) {
7.        wait(NULL);
8.        printf("In parent\n");
9.     } else {
10.       printf("In child\n");
11.    }
12.    printf("Ending main: %d\n", child_pid);
13. }
```

What is the output of this program? You can assume that no syscalls fail and that *the child's PID is 1234.* Fill in the following table with your prediction of the output:

Standard out	test.txt
Starting main	*In child* *Ending main: 0* *In parent* *Ending main: 1234*

Problem 4b[4pts]: Next, assume that we have altered this code as follows:

```
1.  int main() {
2.     printf("Starting main\n");
3.     int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
4.     int new_fd = dup(STDOUT_FILENO);
5.     dup2(file_fd, STDOUT_FILENO);
6.     pid_t child_pid = fork();
7.     if (child_pid != 0) {
8.        wait(NULL);
9.        printf("In parent\n");
10.    } else {
11.       dup2(new_fd, STDOUT_FILENO);
12.       printf("In child\n");
13.    }
14.    printf("Ending main: %d\n", child_pid);
15. }
```

What is the output this time? Fill in the following table with your prediction of the output:

Standard out	test.txt
Starting main *In child* *Ending main: 0*	*In parent* *Ending main: 1234*

Problems 4c[5pts]: Consider the following fragment of a program (which is missing lines of code). Fill in the blanks in the code to ensure that the output of this program is always the following two lines in the following order (under all interleavings or schedules):

```
val = 0
val = 1
```

No more than one function call (or system call) per blank! Your solution is not allowed to make assignments to the "`val`" variable. Also, no use of `printf()` or other print statements!

```
void rectangle () {
     pid_t oval = __getppid()_____________________;
     kill(oval, SIGCONT);
}
void circle(int i) {
     val = 1;
}
int val = 0;
int main() {
     pid_t pid = fork();
     _signal(SIGCONT,circle)_____________;

     if (pid == 0) {
          _rectangle()_____________________;
     } else {
          _wait(NULL)______________________;
     }
     printf("val=%d\n",val);
}
```

Although this solution may seem counterintuitive, let's walk through this problem together. From reading the problem description, we know that we MUST always output our print statements in a certain order. This means we probably need some sort of synchronization, in this case, we see a fork() call, so we'll probably use wait() in the parent (the else block). Next, we see in the function rectangle(), there is a kill syscall. We know that from lecture and discussion that we can intercept and handle certain signals via the signal syscall. Finally, in analyzing our end output, we know that we must set the parent's value to 1. So ideally, if we had multiple lines, we could call circle() and wait() in the else block but we can't. So we use a signal handler to call circle for us and rectangle within the child to trigger the signal handler.

Another solution that students creatively came up with is a wait(NULL) call right after fork() and a circle call inside the parent. This solves our issue of fitting both wait and circle into our parent execution.

Problem 4d[3pts]: If a child process sends a SIGKILL signal to its parent, what happens to the parent process and the child process? Can the parent prevent this from happening?

The parent process will be terminated. The child process will keep running. Since SIGKILL cannot be caught, the parent cannot prevent this from happening.

Problem 4e[2pts]: Consider the following scenario. A process forks a child process and the parent process waits on it. Then, the child exits normally and the parent is unblocked. Finally, the parent makes another wait call on the child process. What happens to this last wait call and to the parent process?

Since there will be no child still running when the second `wait()` system call is made, this call will return with an error indicating that there are no children remaining (`ECHILD`).

Problem 4f[3pts]: Why do we switch from the user's stack to a kernel stack when we enter the kernel (e.g. for a system call)? Why do we associate a *unique* kernel stack for *each* user thread?

Because we do not trust the user to have a valid stack, we always switch to a kernel-allocated stack when entering the kernel.

The reason that we associate a unique kernel stack for each user thread is so that the kernel can put the thread to sleep simply by unloading the thread's registers into the TCB and putting this TCB on a wait list. When we wake up the thread later, it can continue exactly where it left off (with all its in-kernel stack state intact and ready to go).

[This page intentionally left blank!]

Problem 5: Alternate Readers-Writers Access to Database [13pts]

```
Reader() {
   //First check self into system
   lock.acquire();
   while (AW > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
   }
   AR++;
   lock.release();

   // Perform actual read-only access
   AccessDatabase(ReadOnly);

   // Now, check out of system
   lock.acquire();
   AR--;
   if (AR == 0 && WW > 0)
      okToWrite.signal();
   lock.release();
}
```

```
Writer() {
   // First check self into system
   lock.acquire();
   while ((AW + AR) > 0) {
      WW++;
      okToWrite.wait(&lock);
      WW--;
   }
   AW++;
   lock.release();

   // Perform actual read/write access
   AccessDatabase(ReadWrite);

   // Now, check out of system
   lock.acquire();
   AW--;
   if (WR > 0){
      okToRead.broadcast();
   } else if (WW > 0) {
      okToWrite.signal();
   }
   lock.release();
}
```

Problem 5a[5pts]: Above, we show a *modified* version of the Readers-Writers example given in class. It uses two condition variables, one for waiting readers and one for waiting writers. Suppose that *all* of the following requests arrive in very short order (while W_1 is still executing):

Incoming stream: $W_1\ R_1\ R_2\ R_3\ W_2\ W_3\ R_4\ R_5\ R_6\ W_4\ R_7\ W_5\ W_6\ R_8\ R_9\ W_7\ R_{10}$

In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this clearly by surrounding them with braces '{}'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue). Explain how you got your answer.

The order would be: $W_1,\ \{\ R_1\ R_2\ R_3\ R_4\ R_5\ R_6\ R_7\ R_8\ R_9\ R_{10}\ \},\ W_2,\ W_3,\ W_4,\ W_5,\ W_6\ W_7$

The reasoning behind our answer is simple: W_1 comes in first. Then, all other requests arrive while W_1 is still processing. When these other requests arrive, all the read requests get placed on the okToRead condition variable (see entry condition for Reader()—AW > 0. Further all write requests get placed on the okToWrite condition variable for a similar reason in the Writer() code. When first writer exits, it notices that WR > 0 and wakes all readers up with a broadcast, and they all start running. Finally, when the last reader finishes, the Reader() code signals the first writer on the condition variable (okToWrite.signal). Subsequent writers are signaled in order by the writer (also with okToWrite.signal).

Our solution assumes that the process of accessing the database (both reading and writing) is long relative to any scheduling quanta. If you assumed that reading the database is so fast that a group of readers (perhaps the first) finishes before all readers wake up and start reading the database (and you said so in your explanation!) we would be willing to entertain other solutions that are carefully explained. See additional discussion in solution for Problem 5e.

Problem 5b[2pts]: How is the above version of the Readers-Writers access control different from the one give in class? Your answer here should be very short.

It gives priority to readers over writers, as opposed to writers over readers like the version from class.

Problem 5c[2pts]: Explain why the above code ensures that if `AR > 0`, then `AW == 0`.

First, AR ≥ 0 and AW ≥ 0, since AR and AW start at zero and only decrement as many times as they increment (each increment associated with a thread the subsequently decrements). Further, AW only increments in the entry code of the Writer() routine, and only if AR == 0 (since both AR ≥ 0 and AW ≥ 0, and AW increments only if AR+AW == 0).

Problem 5d[2pts]: Explain why the above code makes sure that there can only be one writer at a time and that if `AW == 1`, then `AR==0`?

As above, AR ≥ 0 and AW ≥ 0. And, AW only increments in the entry of the Writer() routine if both AR and AW are 0. Consequently, AW cannot increment past one ⇒ implying that AW ≤ 1 (i.e. there can only be one writer at a time).

Further, at the time that AW increments to one (1), we know that AR==0. Subsequently, as long as AW > 0, then AR remains constant because AR only increments in the entry code of the Reader() routine and only if AW==0. So, AW==1, means that AR=0.

Problem 5e[2pts]: Finally, explain why the code for `Reader()` does not have to check to see if `WR > 0` (and as a result never needs to call `okToRead.broadcast()`)?

Consider a particular thread which is about to finish reading the database. Clearly, the code at the end of Reader() which might reasonably check for WR > 0 is only executed by an active reader (i.e. AR > 0 just prior to the closing lock.acquire()). By (5c) above, AW==0 when AR > 0. Consequently, at the time this particular thread started reading the database, we know that AW==0. Assume, for a moment, that WR > 0 at this time. We know that every one of the WR threads must have been woken in Writer() at the time that AW⇒0, since that transition executes by okToRead.broadcast() and no further reading threads will be put to sleep while AW=0. So – the Reader() code doesn't need to check WR > 0 or wake readers since they are already awake.

Some additional information (for both 5a, 5b, and 5e, here). Although the Reader code does not have to check WR > 0, there is an interesting ordering "bug" here that would alter the processing order if database reads are fast enough. Reasoning is as follows:

If database reads are fast enough, then it is possible that AR=0 with WR > 0 at the end of the the Reader() code. In that case a writer will be woken prematurely, even though there are a bunch of readers on the ready queue and about to contend for the lock. If that writer is put on the ready queue and somehow gets to run and acquire the lock before the waking readers get to run or a new writer gets to run before the readers get to run, then it might be possible for the priority of readers over writers to be violated. Note that simply checking for WR > 0 in the entry code for Writer() would fix this problem and the exit code of Reader() would still not need to check the value of WR:

```
while ((AW + AR + WR) > 0) {
   WW++;
   okToWrite.wait(&lock);
   WW--;
}
```

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]