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

Spring 2026 John Kubiatowicz

Midterm I SOLUTION

February 24th, 2026
CS162: Operating Systems and Systems Programming

Your Name:	
SID:	
CS162 Login:	
TA Name/Section Time	
Person to Left, Right, Front, and Back of you:	

This is a **closed book** exam. You are allowed 1 page of notes (both sides). You have 170 minutes 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	30	
4	19	
5	15	
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]: A thread attempting to access another thread's stack will always lead to segmentation fault.

☐ True [✓ False]

Explain: *If one thread accesses the stack of another thread in the same process, this activity will not necessarily cause a segmentation fault because the two threads are in the same address space. (Note: if the stack is corrupted in the process, subsequent activity may cause a violation, but this is not guaranteed.)*

Problem 1b[2pts]: Suppose that a process overrides the default `SIGINT` signal handler with a signal handler that does not call `exit()`, using `sigaction()`. In this situation, the process prevents itself from being terminated.

☐ True [✓ False]

Explain: *The process can still be terminated with a SIGKILL signal, which cannot be caught (since its default hander cannot be overridden).*

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

[✓ True] ☐ False

Explain: *This is true because a thread pool provides a fixed maximum number of active threads. When used to handle incoming connections from a web server, a thread pool can limit the number of simultaneous connections and thus limit memory usage.*

Problem 1d[2pts]: In the common error message "Segmentation Fault, Core Dumped", the phrase "Core Dumped" means that a particular CPU (core) in the processor is "dumped" or removed from the scheduling queue and restarted.

☐ True [✓ False]

Explain: *The word "Core" in this context refers to "Core Memory", the most common type of memory used in mainframe computers before solidstate RAM became available. "Core Dumped" simply means that the contents of process memory is written out to disk.*

Problem 1e[2pts]: Assume that the primary thread of a process is blocked waiting for the completion of I/O. When the I/O completes and it becomes unblocked, it will begin running immediately.

☐ True [✓ False]

Explain: *When I/O completes, the waiting thread is simply placed back on the ready queue to be run by the scheduler at a later time. Thus, it merely becomes unblocked. It doesn't run immediately.*

Problem 1f[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: *The scheduler can be invoked under a variety of circumstances, not just when an interrupt occurs. Examples include internal events such as `yield()` and I/O of all sorts.*

Problem 1g[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: *For security reasons, we can never allow the user to run arbitrary code with the kernel-mode bit set or even know the addresses of kernel handlers. So, a system call has to atomically, in hardware, (1) set the kernel mode bit while (2) switching to a kernel stack and (3) jumping to a numbered handler in the kernel.*

Problem 1h[2pts]: In the following code, assume all calls to fork() succeed:

```
int main(void) {
   if (fork() == 0) {
      fork();
      exit(0);
   }
   if (wait(NULL) == -1) {
      return 1;
   }
   if (wait(NULL) == -1) {

      return 1;
   }
   return 0;
}
```

This program is guaranteed to return 0:

☐ True [✓ False]

Explain: *This program returns 1: the child from the first fork() enters the first if clause, then generates a new child (call this a grandchild). Both child and grandchild exit immediately. Meanwhile, the parent executes the first wait(NULL), which returns the childs PID. The second wait(NULL) returns -1, since there are no additional children ⇒ return 1.*

Problem 1i[2pts]: A process's heap is contiguous in physical memory.

☐ True [✓ False]

Explain: *A process's heap is contiguous in virtual memory, not physical memory, where it can be spread across many, non-contiguous physical pages.*

Problem 2: Multiple Choice [18pts]

For the multiple choice, below, selecting an item when it is inappropriate or not selecting an item when it is appropriate will result in loss of points.

Problem 2a[2pts]: Which of the following needs to be saved and restored on a context switch between two threads in two different processes (*choose all that apply*):

A: ☑ Computational registers (integer and floating point).

B: ☑ The Page-table root pointer.

C: ☐ Buffered data in streams that have been opened with `fopen()`.

D: ☐ Contents of the file descriptor table

E: ☐ The Page Table.

A: TRUE. The computational registers are altered during thread execution. If registers from the first thread are not saved, they will be altered by the second thread (and vice-versa).

B: TRUE. Since we are switching between different processes, we need to change the address space and thus the active page table. We do this by changing which page table is in use.

C-E: FALSE. Information that is stored in memory does not need to be saved and restored during a context switch, since pointers to this information changes with PCB/Address translation.

Problem 2b[2pts]: Kernel mode differs from User mode in what way? (*choose all that apply)*:

A: ☑ The CPL (current processor level) is 0 in kernel mode and 3 in user mode for Intel processors.

B: ☑ In Kernel mode, additional instructions become available, such as those that modify page table registers and those that enable/disable interrupts.

C: ☐ Specialized instructions for security-related operations (such as for AI acceleration) are only available from Kernel mode.

D: ☑ Control of I/O devices (such as the timer, or disk controllers) are only available from kernel mode.

E: ☑ Pages marked as Kernel-mode in the PTEs are only available in kernel mode.

A: TRUE. Intel processors have 4 "rings" of protection levels, with 0 at the highest level. Most operating systems on Intel processors use 0 for kernel and 3 for user-mode.

B: TRUE. One of the ways that protection is enforced within a kernel is by restricting access to operations/instructions that could compromise the security model.

C: FALSE. Whatever type of security is enhanced by AI, this is not part of the dual-mode kernel/user distinction. These instructions would be available for user code.

D: TRUE. I/O devices must be carefully controlled so as to enforce the integrity of data (i.e. ownership of files/sequence of blocks within files) and integrity of the kernel operation (i.e. timer). Hence, this access must be restricted to the kernel.

E: TRUE. The kernel memory must be protected from user access to make sure that user processes cannot compromise kernel operation.

Problem 2c[2pts]: Consider the following pseudocode implementation of a `lock_acquire()`.

```
lock_acquire() {
    interrupt_disable();
    if (value == BUSY) {
         put thread on wait queue;
         Go to sleep();
    } else {
         value = BUSY;
    }
   interrupt_enable();
}
```

Which of the following are TRUE? Assume we are running on a uniprocessor/single-core machine. (*choose all that apply*):

A: ☐ For this implementation to be correct, we should call `interrupt_enable()` before sleeping.

B: ☐ For this implementation to be correct, we should call `interrupt_enable()` before putting the thread on the wait queue.

C: ☑ For this implementation to be correct, `sleep()` should trigger the scheduler and the next scheduled thread should enable interrupts.

D: ☐ It is possible for this code to be run in user mode.

E: ☐ None of the above.

A: FALSE. If we reenable interrupts just before sleeping, then it is possible that the lock would be freed and thread taken back off the wait queue – only for use to go immediately to sleep, possibly to not wake again.

B: FALSE. If we reenable interrupts just before putting thread on wait queue, then the releaser would notice us (and thus try to wake us up) – and we would proceed to put ourselves on the wait queue and go to sleep, possibly never to wake up again.

C: TRUE. This solution is correct because we maintain the atomicity of the lock while we put ourselves on the wait queue and context switch to another thread. The other thread will then get loaded and reenable interrupts cleanly.

D: FALSE. This code cannot run in user mode because it enables and disables interrupts.

E: FALSE. For obvious reasons.

Problem 2d[2pts]: In the below program, several threads are created.:

```
int global;
void* foo(void* arg) {
    printf("%p\n", &foo);
    printf("%p\n", &global);
    printf("%p\n", &arg);
    printf("%p\n", arg);
    return NULL;
}
int main() {
    void* hmem = malloc(1);
    for (int i = 0; i < 3; i++) {
        pthread_t pid;
        pthread_create(&pid, NULL, foo, hmem);
    }
}
```

Which of the following statements in `foo()` always print the same memory address when evaluated by different threads in the same process?

A: ☑ `printf("%p\n", &foo);`

B: ☑ `printf("%p\n", &global);`

C: ☐ `printf("%p\n", &arg);`

D: ☑ `printf("%p\n", arg);`

E: ☐ None of the above

A: TRUE. The function `foo()` is at a static place in memory.
B: TRUE. The global variable `global` is at a static place in memory.
C: FALSE. Each thread has a different stack, thus the address of argument '`arg`' is different for each of the threads.
D: TRUE. Unlike the address of argument '`arg`', the value of '`arg`' is the same for all threads (i.e. `arg == hmem`).
E: FALSE. For obvious reasons.

Problem 2e[2pts]: Which of the following are true about interrupts?

A: ☑ The x86 hardware saves the values for the user's stack pointer and program counter before jumping to the interrupt handler.

B: ☐ After interrupting a user thread, interrupts create a new kernel thread to run the interrupt handler.

C: ☐ Disabling interrupts is guaranteed to implement mutual exclusion.

D: ☑ Lower priority interrupts may be put on hold by a higher priority interrupt.

E: ☐ None of the above.

A: TRUE. All processors, including the x86, must save the program counter before jumping to an interrupt handler. In addition, the x86 will save the user's stack pointer and load the kernel stack as part of the hardware interrupt handling.
B: FALSE. The interrupt handler runs on the kernel stack of the interrupted user thread and must be short and non-blocking. On finishing, the interrupt handler causes the user thread to continue where it left off. No new thread is created.
C: FALSE. On any system with multiple hardware threads, such as a multicore processor, interrupt disable is not guaranteed to implement mutual exclusion because the other hardware threads keep running after interrupts are disabled.
D: TRUE: Typical interrupt handling will reenable higher-priority interrupts after setting up their environment (i.e. after disabling any interrupts that have the same priority or lower as the currently executing handler).
E: FALSE. For obvious reasons.

Problem 2f[2pts]: Which of the following are true about condition variables? (*choose all that apply*):

A: ☑ `cond_wait()` can only be used when holding the lock associated with the condition variable.

B: ☐ In practice, Hoare semantics are used more often than Mesa semantics.

C: ☐ Mesa semantics will lead to busy waiting in `cond_wait()`.

D: ☑ Each condition variable has its own wait queue for sleeping threads.

E: ☐ All of the above.

A: TRUE. The Monitor paradigm requires `cond_wait()` to be executed only while holding the monitor lock.
B: FALSE. Mesa semantics are much more common in practice that Hoare semantics because they are easier to implement. Among other things, `signal()` or `broadcast()` can be implemented simply by taking threads off a wait queue and placing them on run queue.
C: FALSE. Although `cond_wait()` occurs in a loop, this does not constitute busy-waiting because the thread is put to sleep until the next signaling event.
D: TRUE. This is the definition of a condition variable – it is a queue for waiting threads.
E: FALSE. For obvious reasons.

Problem 2g[2pts]: Select all true statements regarding Pintos (*choose all that apply*):

A: ☑ Virtual address 0xc0004567 always maps to the same physical address in Pintos
B: ☑ Even in user mode, kernel memory is always mapped.
C: ☑ Virtual address 0x08045671 may refer to multiple locations in physical memory, depending on the currently running user process.
D: ☐ The kernel can read from any user virtual address in Pintos, including unmapped user addresses.
E: ☐ The PCB of a process is stored on the stack of its main thread.

A: TRUE. The virtual address 0xc0004567 is part of the kernel address space, which maps to the same kernel page in all user processes (since the kernel expects the same contents regardless of the user process that is currently running).
B: TRUE. Memory addresses above 0xc0000000 are always mapped by page table entries to the kernel memory, even in user mode. These page table entries are marked as "kernel only" so that attempts to access them in user mode cause page faults.
C: TRUE. Because 0x08045671 is a user address, it can be mapped to a different physical page for each user process.
D: FALSE. The kernel cannot access pages that do not have a mapping in the page table.
E: FALSE. The PCB is stored in kernel space, in a table of PCBs, not on the user-level stack.

Problem 2h[2pts]: Which of the following are true about syscalls in Pintos? (*choose all that apply)*:

A: ☐ User programs directly call the syscall functions in the file src/userprog/syscall.c.
B: ☑ Arguments passed into system calls must be validated before they are used.
C: ☐ Syscalls in Pintos are run in user mode.
D: ☑ The wait syscall returns the exit code of the child process specified in the argument to the function.
E: ☐ None of the above.

A: FALSE. Directly calling the syscall functions in the file src/usrprog/syscall.c would not work because user code cannot access kernel memory. Instead, user code must invoke a system call with a special instruction that traps into the kernel, entering kernel mode, and jumping to a predefined handler based on the syscall number placed in a register.
B: TRUE. The kernel must validate all syscall arguments because it cannot trust user code.
C: FALSE. Syscalls are run in kernel mode.
D: TRUE. As defined in Pintos, the `wait()` syscall does return the exit code of the specified child process (note that this is different for Linux, but the question specified Pintos).
E: FALSE. For obvious reasons.

Problem 2i[2pts]: Suppose we wanted to redirect the output of one program, as input to another program (piping). The parent process P1 creates a pipe, and then forks to create P2. (*choose all that apply)*:

A: ☐ The pipes in both P1 and P2 have the same file descriptor numbers, but do not point to the same file descriptions

B: ☐ The pipes in P1 and P2 have different file descriptor numbers, but point to the same file descriptions.

C: ☐ If P1 closes the read end of pipe and P2 closes the write end, then P2 may pass information unidirectionally to P1.

D: [✓] It is acceptable for neither P1 nor P2 to close any descriptors.

E: ☐ None of the above.

A-B: FALSE. Not only do the pipes in both P1 and P2 have the same file descriptor numbers, but they also point to the same file descriptions.

C: FALSE. Given the specifications here, it would be possible to send information from P1 ⇒ P2 (P1 writes, P2 reads), but the opposite direction P2⇒P1 is not supported (P2 needs to write, P1 needs to read).

D: TRUE. P1 and P2 are each allowed to do anything they want with the file descriptors after `fork()` has completed.

E: FALSE. For obvious reasons.

[This page intentionally left blank]

Problem 3: Atomic Synchronization and the Fair Lock [30pts]

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) {`	`SWAP(&addr, expr) {`	`CAS(&addr, expr1, expr2) {`
`  int result = M[addr];`	`  int result = M[addr];`	`  if (M[addr] == expr1) {`
`  M[addr] = 1;`	`  M[addr] = expr;`	`     M[addr] = expr2;`
`  return (result);`	`  return (result);`	`     return true;`
`}`	`}`	`  } else {`
		`     return false;`
		`  }`
		`}`

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. 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));
```

Problem 3a[2pts]: Explain what it means for these operations to be atomic. Why is this behavior desirable enough that all modern processors after MIPS have some variation of the above operations? *Please answer in 3 sentences or less.*

What it means for these operations to be "atomic" is that they cannot be interleaved with other instructions or operations, i.e. at any given time, either all of the subcomponents occur as a unit or none of them do. These operations are typically implemented in hardware as single instructions. The existence of these operations is important for synchronization at user level: without them (such as with MIPS), the only other option for synchronizing between user-level threads is to do something really complex/non-symmetric with load/store operations such as with the "Got Milk" example from class.

Problem 3b[2pts]: Show how to implement a spinlock `acquire()` with a single while loop using CAS instead of TSET. Fill in the arguments to CAS below. *Hint: need to wait until can grab lock.*

```
void acquire(int *mylock) {

  while (!CAS(mylock, __0__________________, __1________________ ));
}
```

As discussed in class, `Futex()` is a *system call* that allows a thread to *put itself to sleep,* under certain circumstances, on a queue associated with a user-level address. It also allows a thread to ask the kernel to wake up threads that might be sleeping on the same queue. Recall that the function signature for `Futex` is:

```
int futex(int *uaddr, int futex_op, int val);
```

In this problem, we focus on two `futex_op` values:

1. For `FUTEX_WAIT`, the kernel checks atomically as follows:

   if (`*uaddr == val`): the calling thread will sleep and another thread will start running
   if (`*uaddr != val`):the calling thread will keep running, i.e. `futex()` returns immediately
2. For `FUTEX_WAKE`, this function will wake up to val waiting threads.

Problem 3c[2pts]: Fill out the missing blanks, in the `acquire()` function, below, to make a version of a lock that does not busy wait. The corresponding `release()` function is given for you:

```
void acquire(int *thelock) { // Acquire a lock
   while (TSET(thelock)) {

      futex(thelock, ___FUTEX_WAIT__________, ______1_________________);
   }
}
void release(int *thelock) { // Release a lock
   *thelock = 0;
   futex(thelock, FUTEX_WAKE, 1);
}
```

In the remainder of this problem, we will develop a queue-lock version of `acquire()` and `release()` that has the following properties:

1. Threads will be granted access to the lock in FIFO order, based on the time that they enter `acquire()`. The order of threads entering simultaneously will be undefined.
2. Threads that are waiting on the lock will be put to sleep (*not spinning).* So, there should be no spin waiting at all in this problem!

To build a queue, we will need to link waiting threads together somehow, which will require individual links in our queue. To make this interesting, we will build our queue lock without calling malloc() or free(), but rather using stack-allocated local structures that are automatically released when a procedure exits. Note that we will be building a FIFO queue in which the head of the queue is attached to the "anchor" element, and in which items are added to the tail. Here are the definitions we will use to make this work:

```
// Possible lock states
typedef enum {
   UNLOCKED, LOCKED
} LockState;

// The actual structure of a queue lock!  Every lock has one of these.
typedef struct QueueLock {
   LockEntry *tail;
   LockEntry anchor;
   LockState lockvar;
} QueueLock;

// Every linked thread has one of these structures while it is waiting.
typedef struct LockEntry {
   LockEntry *next;
   int waitlock;
   int waitnext;
} LockEntry;

// This is our actual queue:
QueueLock ourLock;
```

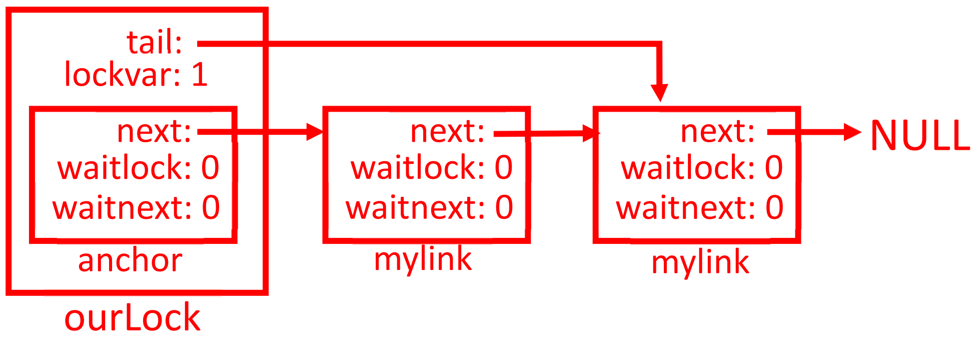
Problem 3d[2pts]: Fill out the following lines to complete the initialization for the queue. Remember – you are trying to initialize this queue as *empty.* Note that we have already added the semicolons, so you shouldn't need any additional ones. Hint: a structure element of type "`struct foo`" can be initialized to all zero with "`foo = {};`"

```
void LockInit(QueueLock *myLock) {

   myLock->tail = &(myLock->anchor);_________________________;

   myLock->anchor = {};______________________________________;

   myLock->Lockvar = UNLOCKED;
}
```

Now, *ignoring concurrency* for a moment, our lock `acquire()` and `release()` routines are going to look like the following code sketch. We have numbered code lines for reference in later problems. Note that the acquire version spins on its own local `waitlock` entry, which means that any shared memory traffic generated by it will not interfere with the spinning of any other threads:

```
    void acquire(QueueLock *myLock) {
1:     LockEntry mylink = {};       // Stack allocated queue link

       // Get our position in line by enqueuing us at the end of the list
2:     LockEntry *oldtail = myLock->tail;
3:     myLock->tail = &mylink;
4:     oldtail->next = &mylink;

       // Wait for lock to become free
5:     If (myLock->anchor.next != &mylink || myLock->lockvar == LOCKED) {
6:        while (!mylink.waitlock);
7:     }
8:     myLock->lockvar = LOCKED;

       // dequeue ourself from the queue.  We know we are at head of queue
9:     if (myLock->tail == &mylink) {
10:       myLock->tail = &(myLock->anchor);
11:       myLock->anchor.next = NULL;
12:       return;
13:    }
14:    myLock->anchor.next = mylink.next
15:    return;
    }

    // Release – either give to next thread in line, or release lock
    void release(QueueLock *myLock) {
17:    if (myLock->tail != &(myLock->anchor)) {
          // Wake up the thread
18:       myLock->anchor.next->waitlock = 1;
19:    } else {
20:       myLock->lockvar = UNLOCKED;
21:    }
    }
```

Problem 3e[2pts]: Draw a picture of the structure of the queue after 2 items are linked into it. Use a box for each LockEntry item, and arrows for pointers:



Problem 3f[3pts]: For each of the following potential context switch points, state whether or not a context switch at that point could cause incorrect behavior. List the potential issues with switching at that point. (Each point may have multiple issues.) Choose from: ***No Problem, Bad Enqueue, Bad Dequeue, Bad Release.*** Explain your choice.

```
               void acquire(QueueLock *myLock) {
Point 1 ---->     LockEntry mylink = {};
Point 2 ---->     LockEntry *oldtail = myLock->tail;
Point 3 ---->     myLock->tail = &mylink;
                  oldtail->next = &mylink;
```

Point 1: *No Problem. At this point, there is no issue with switching and returning, since no global state has been altered yet.*

Point 2: *Bad Enqueue. At this point, if we switch out and return, the tail pointer (myLock->tail) may have changed, so we will link things incorrectly.*

Point 3: *Bad Dequeue/Bad Release. At this point, if we switch out, we will have successfully made ourselves visible as the tail of the list. However, we haven't updated the previous element to point at us, so dequeueing will fail and/or an attempt by the previous lock holder to inform us during release() will fail.*

Problem 3g[3pts]: Although you cannot fix all of the problems you identified in problem **(3f)**, you can at least make sure that every `acquire()` operation properly enqueues the threads on the FIFO queue, thereby choosing the final order of lock acquisition regardless of the concurrency of the operations. We say that the `acquire()` operations are properly serialized in those circumstances. Rewrite lines 1-4 to guarantee proper serialization (*HINT: you will use a* `do/while` *loop with CAS*). We started you off with the first two lines:

```
   void acquire(QueueLock *myLock) {
1:    LockEntry mylink = {};
A:    do {
B:        *oldtail = myLock->tail;
C:    } while(!CAS(&(myLock->tail), oldtail, &mylink);
D:    oldtail->next = &mylink
```

Note: The C language doesn't allow us to declarare `oldtail` on line B because it goes out of scope in the "while" condition. So, ideally, we would have another line before "do" to declare "`LockEntry *oldtail;`" We will not penalize someone who does declare "LockTail `*oldtail=`" here, however!

Problem 3h[4pts]: Why is the dequeue code at lines 9-13 problematic when concurrent threads are trying to `acquire()` the queue lock? Explain the problem and produce an alternative solution to fix this problem in the case of concurrency with acquire(). Note that you are coming up with code that will *replace* lines 9-13. You do not have to use every line here, and can do this with TWO semicolons. *Hint: make sure to think about concurrent switch points with new* `acquire()` *requests.*

```
       // Original version of lines 9-13
9:     if (myLock->tail == &mylink) {
10:       myLock->tail = &(myLock->anchor);
11:       myLock->anchor.next = NULL;
12:       return;
13:    }
```

Explain Issue: *The issue here is that things might change after we decide that we are last on queue (in Line 9). Two issues: (1) we might delete new queue entry by accident if there is a switch between Lines 9 and 10 or (2) we might destroy link from anchor to new entry if there is a switch between Lines 10 and 11.*

Note: Our solution will be clever here in that we know before Line 9 that we are pointed at by `myLock->anchor.next` *and can clear that link, but later fix it in Line 14+ if we aren't last in queue.*

```
       // Thread-safe version of original lines 9-13
E:      myLock->anchor.next = NULL;  // will fix in Line 14 if needed
F:      if (CAS(myLock->tail, &mylink, &(myLock->anchor))
G:           return;
H:
```

Problem 3i[4pts]: The `acquire()` code busywaits threads waiting for the lock. Identify the line responsible for busywaiting, and fix the code to remove busywaiting. You do not need to use every line, below, in your updated version of `release()`. *Hint: be careful because there is a race condition in release() as well, but it is easy to fix – think back to your answer in problem* ***(3g)***.

Line # to replace: *6*
What should you replace it with:

```
I:    futex(mylink->waitlock, FUTEX_WAIT, 0);
```

```
    // Original release code
    void release(QueueLock *myLock) {
17:    if (myLock->tail != &(myLock->anchor)) {
          // Wake up the thread
18:       myLock->anchor.next->waitlock = 1;
19:    } else {
20:       myLock->lockvar = UNLOCKED;
21:    }
    }
```

Updated version of the release code:

```
    void release(QueueLock *myLock) {
J:     if (myLock->tail != &(myLock->anchor)) {
K:         If (mylink->anchor.next != NULL) {
L:             myLock->anchor.next->waitlock = 1;
M:             futex(myLock->anchor.next.waitlock,FUTEX_WAKE, 1);
N:         }
19:    } else {
20:       myLock->lockvar = UNLOCKED;
21:    }
    }
```

Problem 3j[2pts]: The code in lines 14-15 is problematic when concurrent threads are trying to `acquire()` the queue lock. Explain what the problem is and how you need to fix it from a high level. You will write actual code in problem **(3k)**, below. *Hint: think about premature freeing of objects on the stack. Your fix will likely involve waiting (notice unused element in `LockEntry`).*

```
14:    myLock->anchor.next = mylink.next
15:    return;
```

Explain Issue: *After we return (i.e. Line 15), the `mylink` structure will be freed, since the stack frame is destroyed. We cannot do that until we are sure that `mylink` is not going to be used by anbother thread. By Line 14, we know that some other LockEntry structure is either already fully linked to us or in the process of being linked (because of Lines 9-13). If `mylink.next == NULL`, we know that some other thread will still set it., so we must wait until `mylink.net != NULL`. We will use `LockEntry->waitnext` as a signal.*

Problem 3k[4pts]: Combine everything you have learned to make working code for `acquire()`. You have already given the new version of release() in problem **(3k)**. In the following, you may repeat code from the original solution by naming the line or a range of lines without rewriting them. For instance, if you wanted to repeat line 1 you would just say "`Line 1`". Or, if you wanted to repeat lines 1-4 you would just say "`Lines 1-4`". For all the other lines, you should fill them. You are also allowed to reference solutions to problems **(3g)** – **(3i)** with lines A-I.

Make sure to use *ALL* of lines 1-21 except those that were replaced by your answers to problems **(3g)** – **(3i)**. We started you out with 5 lines. You are not required to use all of the spaces:

```
void acquire(QueueLock *myLock) {
    Line 1
    Lines A-D
    if (oldtail->waitnext)
        futex(&oldtail->waitnext, FUTEX_WAKE, 1);
    Line 5
    Line I
    Lines 7-8
    Lines E-G
    if (mylink->next) {
        mylink->waitnext = 1;
        futex(&mylink->waitnext, FUTEX_WAIT, 0);
    }
    Lines 14-15
}
```

Problem 4: Short Answer Potpourri [19pts]

For the following questions, provide a concise answer of NO MORE THAN 2 SENTENCES per sub-question (or per question mark), unless instructed otherwise.

Problem 4a[2pts]: Why do we switch from the user's stack to a kernel stack when we enter the kernel (e.g. for a system call)? What is the advantage of associating 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).

Problem 4b[2pts]: In 1-2 sentences, explain the use of the `loadusersymbols` macro in GDB. How might this be employed in PintOS?

The `Loadusersymbols` macro, as the name suggests, loads the symbol table of a given user program into GDB so that, instead of raw addresses, GDB can actually display function names, variables, and line numbers for user processes. Without it, we would not map user executable symbols and, thus, be unable to do breakpoints, stack traces, etc.

Problem 4d[3pts]: Suppose the grep program is invoked under Pintos with the following command line: `grep help cs162-midterm.txt`. Using a diagram for illustration, explain what the call stack will look like when the program starts. How much total memory will be needed to set up this stack?

Address	*Name*	*Data*	*Type*
0xBFFFFFEE	*argv[2][...]*	*cs162-midterm.txt\0*	*char[18]*
0xBFFFFFE9	*argv[1][...]*	*help\0*	*char[5]*
0xBFFFFFE4	*argv[0][...]*	*grep\0*	*char[5]*
0xBFFFFFE0	*argv[3]*	*0*	*char **
0xBFFFFFDC	*argv[2]*	*0xBFFFFFEE*	*char **
0xBFFFFFD8	*argv[1]*	*0xBFFFFFE9*	*char **
0xBFFFFFD4	*argv[0]*	*0xBFFFFFE4*	*char **
0xBFFFFFC8	*-------*	*(12 bytes undefined)*	*<padding>*
0xBFFFFFC4	*argv*	*0xBFFFFFD4*	*char ***
0xBFFFFFC0	*argc*	*3*	*int*
0xBFFFFFBC	*return address*	*0*	*(void (*)())*

This stack needs a total of 68 bytes.

Problem 4e[2pts]: What is "hyperthreading"? Does it allow threads to execute "concurrently" or "in parallel"? Explain.

Hyperthreading is a computer architectural technique that allows multiple loaded threads to simultaneously share functional units in a modern, out-of-order pipeline. Each thread has its own set of registers and makes forward progress independently of other threads in the pipeline. The threads thus operate both concurrently and in parallel. [Remember parallel⇒concurrent

Problem 4f[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?

The processor transitions from user mode to kernel mode when (1) the user executes a system call, (2) the processor encounters an synchronous exception such as divide by zero or page fault, (3) the processor responds to an interrupt. The user cannot execute arbitrary code because entry into the kernel is through controlled entry points (not under control of the user).

Problem 4g[3pts]: Assume that we have the following piece of code:

```
int main() {
   printf("Starting main\n");
   int file_fd = open("test.txt", O_WRONLY | O_TRUNC | O_CREAT, 0666);
   int new_fd = dup(STDOUT_FILENO);
   dup2(file_fd, STDOUT_FILENO);
   pid_t child_pid = fork();
   if (child_pid != 0) {
      wait(NULL);
      printf("In parent\n");
   } else {
      dup2(new_fd, STDOUT_FILENO);
      printf("In child\n");
   }
   printf("Ending main: %d\n", child_pid);
}
```

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 4h[2pts]: Consider the following code (assume there are no errors from `pthread_create()`):

```
int global = 0;
void myincr(int *var) {
   *var = *var + 1;
}
void* mythread(void *myglobal) {
   int i;
   for (i=0; i<10; i++) {
      myincr((int *)myglobal);
   }
}
int main() {
   pthread_t mythreads[4];
   for (int i = 0; i < 4; i++) {
      pthread_create(&mythreads[i], NULL, mythread, (void*)&global);
   }
   for (int i = 0; I < 4; i++) {
      pthread_join(mythreads[i],NULL);
   }
   printf("Final value: %d\n",global);
   return 0;
}
```

What is the range of values that this code could output for its final value? Explain your answer.

If the `myincr()` routine were atomic, then the final value after threads join would be

$\#\, threads \times \#\, iterations = 4 \times 10 = 40$

However, because of the unprotected race condition in `myincr()` routine, the increment operations interfere with one another. In fact, the results can vary from "Final value: 2" to "Final value: 40". We know that the smallest it can be is 2 because the last thread to write the value did so by adding 1 to a value written by another thread, and that thread added 1 to a number that cannot be smaller than 0 $\Rightarrow$ at least 2.

Problem 4i[2pts]: Write a new version of myincr() so that you can be sure that the code outputs "Final value: 40", using no more than **TWO** semicolons (we will grade for conciseness). You can assume that variables can be declared mid-procedure, if that is helpful. *Note: you are not allowed to hardcode the solution in any way.*

```
void myincr(int *var) {

   do {  // one of the following lines will have a 'while':

      int oldval=*var;

   } while (!CAS(var, oldval, oldval+1);
}
```

[This page intentionally left blank]

Problem 5: Readers-Writers Access to Database [15 pts]

```
Reader() {
   //First check self into system
   lock.acquire();
   while ((AW + WW) > 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 (WW > 0){
      okToWrite.signal();
   } else if (WR > 0) {
      okToRead.broadcast();
   }
   lock.release();
}
```

Problem 5a[2pts]: Above, we show the Readers-Writers example given in class. It used 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 R_1 and R_2 are still executing):

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

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.

SOLN: Processed as follows: { R_1 R_2 R_3 } W_1 W_2 W_3 W_4 W_5 W_6 W_7 {R_4 R_5 R_6 R_7 R_8 R_9 R_{10} }

Since this algorithm gives precedence to writes over reads, although the first two reads get to execute together and right away, the remaining reads are deferred until each write is processed (one at a time). After all writes are finished, then the reads execute as a group.

Problem 5b[2pts]: Let us define the *logical arrival order* by the order in which threads first acquire the monitor lock. Suppose that we wanted the results of reads and writes to the database to be the same as if they were processed one at a time – in their logical arrival order; for example, W_1 W_2 R_1 R_2 R_3 W_3 W_4 R_4 R_5 R_6 would be processed as W_1 W_2 {R_1 R_2 R_3} W_3 W_4 {R_4 R_5 R_6} *regardless of the speed with which these requests arrive.* Explain why the above algorithm does not satisfy this constraint.

Once it starts buffering items (i.e. putting requests to sleep on condition variables), the above algorithm does not keep track of the arrival order. The above algorithm loses all ordering between buffered reads and buffered writes. Thus, it cannot guarantee that reads execute after earlier writes and before later writes (as defined by logical arrival order).

Assume that our system uses Mesa scheduling and that condition variables have FIFO wait queues. Below is a sketch of a solution that uses only two condition variables and that *does* return results as if requests were processed in logical arrival order. Rather than separate methods for `Reader()` and `Writer()`, we have a single method `AccessDB()` which takes a "`newType`" variable (0 for read, 1 for write):

```
 1: pthread_mutex_t monitorLock;            // Monitor lock
 2: pthread_cond_t stallQueue, onDeckQueue; // Staging queues
 3: int numStalled = 0, numOnDeck = 0;      // Counts of sleeping threads
 4: int curType = 0, numAccessing = 0;      // Active threads in DB

    // Control entry so that only one writer or multiple readers
 5: void AccessDB(int newType) {     // type = 0 for read, 1 for write
 6:    pthread_mutex_lock(&monitorLock);

 7:    /* Missing wait condition */
 8:    {  numStalled++;
 9:       pthread_cond_wait(&stallQueue, &monitorLock);
10:       numStalled--;
11:    }
12:    /* Missing wait condition */
13:    {  numOnDeck++;
14:       pthread_cond_wait(&onDeckQueue, &monitorLock);
15:       numOnDeck--;
16:    }
17:    /* Missing code */
18:    pthread_mutex_unlock(&monitorLock);

19:    // Perform actual data access
20:    AccessDatabase(NewType);

21:    /* Missing code */
22: }
```

The `stallQueue` condition variable keeps *unexamined* requests in FIFO order. The `onDeckQueue` keeps a *single* request that is currently incompatible with requests that are executing. W*e want to allow as many parallel requests to the database as possible, subject to the constraint of obeying logical arrival order.* Here, logical arrival order is defined by the order in which requests acquire the lock at line #6.

Problem 5c[2pts]: Explain why you might not want to use a "while()" statement in Line #7 – *despite the fact that the system has Mesa scheduling*:

Because a while loop will potentially reorder requests on the `stallQueue` condition variable – and we want to keep requests in their original logical arrival order. Thus whatever we do, we do not want to execute `pthread_cond_wait(&stallQueue, &monitorLock)` more than once per thread.

Problem 5d[2pts]: What is the missing code in Line #7? *Hint: it is a single "if" statement.*

```
if (numStalled + numOnDeck > 0)
```

Reasoning: If anyone in the system is waiting, then we want to wait so that we don't get ahead of others (i.e. so that we preserve the logical arrival order).

Problem 5e[2pts]: What is the missing code in Line #12? *This should be a single line of code.*

```
while ((numAccessing > 0) && (newType == 1 || curType == 1))
```

*Reasoning: If there are requests currently in progress and the new request is imcompatible with the set of executing requests, then we have to wait. Note that you could also use an "if" here, but then you would need to conditionalize the "`onDeckQueue`" signal statement in **(5g).** See below.*

Problem 5f[2pts]: What is the missing code in Line #17? *You should not have more than three (3) actual lines of code.*

```
numAccessing++;        // Have another thread accessing DB
curType = newTpe;      // In case we are first of new group
pthread_cond_signal(&stallQueue,&monitorLock); // Let new thread on deck
```

Problem 5g[3pts]: What is the missing code in Line #21? *You should not have more than five (5) actual lines of code.*

```
pthread_mutex_lock(&monitorLock);
numAccessing--;        // One less thread accessing DB
pthread_cond_signal(&onDeckQueue);
pthread_mutex_unlock(&monitorLock);
```

Note: If you use an "if" in (3h), then need this instead of the above signaling:

```
if (NumAccessing == 0)
    pthread_cond_signal(&onDeckQueue); // Wake if someone waiting
```

[This page intentionally left blank]

[Function Signature Cheat Sheet]

```
/******************************* pThreads ******************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_init(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

/****************************** Processes *******************************/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/*************************** High-Level I/O *****************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/******************************** Sockets *******************************/
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);


/*************************** Low-Level I/O ****************************/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
      whence=>SEEK_SET, SEEK_CUR, or SEEK_END
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
```

[*Function* Signature Cheat Sheet (con't)]

```
/****************************** Pintos ********************************/
void list_init(struct list *list);
struct list_elem *list_head(struct list *list)
struct list_elem *list_tail(struct list *list)
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
struct list_elem *list_pop_front(struct list *list);
struct list_elem *list_push_front(struct list *list);


void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);
```

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]