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

Spring 2024 John Kubiatowicz

Midterm I
SOLUTION

February 15th, 2024

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 have 110 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	20	
2	18	
3	22	
4	24	
5	16	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [20 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]: Because the "monitor" pattern of synchronization involves sleeping inside a critical section, it requires a special hardware support to avoid deadlock (i.e. a permanent lack of forward progress).

☐ True [✓ False]

Explain: *No special hardware is required beyond the hardware used for lock implementation. All that is necessary is for the implementation of cond_wait() release the lock when putting the thread on the sleep queue and to reaquire the lock before returning.*

Problem 1b[2pts]: In UNIX, Pipes are implemented with a buffer in user space.

☐ True [✓ False]

Explain: *UNIX Pipes are implemented in the kernel (they are constructed with the `pipe()` system call). Consequently, the buffer is in kernel space.*

Problem 1c[2pts]: Any two processes in a running system have a common ancestor process. *Clarification during exam: a process is considered an ancestor of itself.*

[✓ True] ☐ False

Explain: *All processes can ultimately be traced to the "init" process that starts up at boot time. Init starts shells and kernel daemon processes, which launch other processes.*

Problem 1d[2pts]: After the main thread in a process calls the `exit()` system call (either implicitly or explicitly), the other threads are allowed to run to completion before the process terminates.

☐ True [✓ False]

Explain: *The `exit()` system call frees up all process resources (except for memory storing the exit status), including all of the process' threads.*

Problem 1e[2pts]: A user-level library implements each system call by executing a "transition to kernel mode" instruction followed by a procedure call to an appropriate system call handler in the kernel. To make this work, the user-level library must be built with access to a symbol table that contains the kernel addresses of each system call.

☐ 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 1f[2pts]: Not every call to `fread()` must make a call the kernel using the `read()` system call to retrieve data from the underlying file.

[✓ True] ☐ False

Explain: *This is true because `fread()` makes use of a data buffer in user memory (i.e. in the `FILE` structure). It only makes a call to `read()` when the buffer is empty, but otherwise serves `fread()` requests from the user-level buffer.*

Problem 1g[2pts]: The `test&set` instruction consists of *two* independent operations: (1) read the value from memory and (2) replace it with the value "1". Thus, it must be used in a loop to protect against the case in which multiple threads execute `test&set` simultaneously and end up interleaving these two operations from different threads.

☐ True [✓ False]

Explain: *The `test&set` instruction must atomically read from memory and set the value of the memory to 1. Otherwise it would be no better than a non-atomic combination of a load and a store (and thus suffer from the limitations of the "got milk" solution #3).*

Problem 1h[2pts]: It is possible for a client machine (with IP address X) and server machine (with IP address Y) to have multiple simultaneous but independent communication channels between them – despite the fact that network packets get intermixed in the middle.

[✓ True] ☐ False

Explain: *A socket-to-socket connection over the Internet is uniquely identified by a 5-tuple of values: [Source IP, Source Port, Dest IP, Dest Port, Protocol]. Multiple unique channels between X and Y can be distinguished with different source and/or destination ports which are included in the message headers, thereby allowing messages to be sorted out at source or destination.*

Problem 1i[2pts]: When a process issues a system call in Pintos, the kernel must change the active page table (i.e., update the page table base register) so that the system call handler can access kernel memory.

☐ True [✓ False]

Explain: *PintOS achieves two separate address spaces in a single page table by marking some entries as "kernel only." Consequently, during a system call, the additional mappings for the kernel become immediately available as soon as the processor mode changes to kernel mode – without having the change the page table.*

Problem 1j[2pts]: A user-level application can build an implementation of lock `acquire()` and `release()` by using the interrupt enable and disable functions of the pthread library.

☐ True [✓ False]

Explain: *For security reasons, user code is never allowed to enable or disable interrupts. So, there is no way, even in principle, that they could build a lock using interrupt enable/disable, and there is no such functionality in the pthread library.*

Problem 2: Multiple Choice [18pts]

Problem 2a[2pts]: Which of the following statements about Base&Bound style address translation are true? (*choose all that apply)*:

A: ☐ Base&Bound cannot protect kernel memory from being read by user programs.
B: ☑ Sharing memory between processes is difficult.
C: ☐ Context switches incur a much higher overhead compared to use of a page table.
D: ☑ Base&Bound will lead to external fragmentation.
E: ☑ With Base&Bound, each process can have its own version of address "0".

A: FALSE. Base&Bound prevents user code from accessing memory outside the range from Base to Base+Bound. Consequently is can prevent reading of memory for the kernel.

B: TRUE. The simple Base&Bound mechanisms involves non-overlapping memory ranges and cannot easily provide selectively shared chunks of memory.

C: FALSE. Changing from one address space to another is a simple matter of changing the base and bound registers, which is little different from changing the page table base register (certainly not "much higher").

D: TRUE. Since Base&Bound involves variable-sized chunks of memory, this will lead (over time) to small, unusable blocks of unallocated memory – that is external fragmentation

E: TRUE. Every process can have different contents for address 0 since Base&Bound adds a base pointer to a process' virtual memory address to compute the physical address.

Problem 2b[2pts]: What are some things that can cause a transfer from user mode to kernel mode? (*choose all that apply)*:

A: ☑ User code divides by zero.
B: ☑ The user executes a system call.
C: ☑ A packet is received from the network.
D: ☑ The application uses malloc() to allocate memory from the heap.
E: ☑ The timer goes off.

A: TRUE. Integer divide by zero causes a divide by zero trap and enters the kernel through the appropriate entry in the interrupt vector.

B: TRUE. A system call is a software trap instruction that transfers into the kernel.

C: TRUE. Reception of packets from the network can generate interrupts, which will force a transition from user mode to kernel mode to handle the incoming packets..

D: TRUE. Calls to `malloc()` allocate data on the heap, which can enter the kernel to ask for more memory (via the `sbrk()` system call) to be assigned to the part of the address space assigned to the heap..

E: TRUE. The timer generates an interrupt which causes a transition from user mode to kernel mode.

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]: Which of the following are true about semaphores (*choose all that apply):*

A: ☐ Semaphores can be initialized to any 32-bit value in the range -2^{31} to $2^{31}-1$

B: ☑ If there is at least one thread sleeping on a given semaphore, then the result of performing a `Semaphore.V()` will be to wake up one of the sleeping threads rather than incrementing the value of the semaphore.

C: ☐ Semaphores cannot be used for locking.

D: ☐ The *interface* for `Semaphore.P()` is specified in a way that prevents its *implementation* from busy-waiting, even for a brief period of time.

E: ☑ The pure semaphore interface does not allow querying for the current value of the semaphore.

A: FALSE. Semaphores represent non-negative integers and so cannot be negative.

B: EITHER. This behavior can occur in whatever way we want to be atomic. Originally, we intended this answer to be TRUE, since the fact that we wake a thread means that any internal change in the value of the semaphore would be invisible to users (e.g. any internal increment done by V() must be immediately undone to allow the woken thread to pass through P()). Nonetheless, we decided to accept FALSE as well, since one could consider implementations in which V() simply increments the semaphore and wakes the sleeping thread, which must then compete with other threads to be the lucky one to decrement the value back to zero.

C: FALSE. A semaphore initialized to 1 can be used as a lock, with `P()`⇒`acquire()` and `V()`⇒`release()`, assuming we adhere to a strict disciple of only release a lock if we have already acquired it.

D: FALSE. The interface for Semaphore just says that a thread entering P() is prevented from exiting if the semaphore is 0. It actually doesn't require that the thread sleep(); A spin-waiting lock would be a bad implementation but correct semaphore.

E: TRUE. Although some implementations allow you to query the value of the semaphore, it is not part of the pure semaphore interface (and is not really synchronized so can change out from under you anyway).

Problem 2e[2pts]: In PintOS, every "user thread" has a chunk of memory that serves as a user-level stack and that is matched one-for-one with a corresponding kernel stack in kernel-protected memory. What is true about this arrangement (*choose all that apply*):

A: ☑ When a user-thread makes a system call that must block (*e.g.* a read to a file that must be retrieved from disk), the user thread can be put to sleep at any time by pushing CPU state onto its kernel stack and putting that stack onto an appropriate wait queue inside the kernel.

B: ☐ The kernel stack is often identified in figures as a "kernel thread" because it represents an independent computational entity that can run at the same time as the user thread.

C: ☐ The kernel stack can grow to an arbitrary size (on demand) using virtual memory.

D: ☑ The kernel gains safety by the presence of the kernel stack because it does not have to rely on the correctness of the user's stack pointer register for correct behavior.

E: ☐ None of the above.

A: TRUE. This is the whole point of having a separate kernel stack for every thread. It means that all of the state of the thread – including its sequence of calls in the kernel – are stored on the kernel stack. By pushing the remaining state on the stack, we can just put this stack (and probably the TCB as well) on the given wait queue because it represents everything we need to restart thread from its previous position.

B: FALSE. There is no "kernel thread" that is independent of the "user thread." There is only one thread that is either executing in kernel mode or user mode.

C: FALSE. Typically, the kernel stack is of limited size, since the kernel does not do arbitrarily deep recursive computation. In fact, the PintOS kernel combines both the TCB and kernel stack into a 4KB page. Linux combines the TCB and kernel stack into a single 8KB chunk.

D: TRUE. The contents of the user's segment register and stack pointer cannot be trusted to be useable or large enough to handle kernel code. Further, the kernel cannot use a stack that is in user space without risking having it corrupted by other threads.

E: FALSE. For obvious reasons.

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

A: ☑ If a process calls `execv()`, it does not need to free any of its allocated heap variables.

B: ☐ Using IPC (such as a pipe), a child process is able to share the address of a stack-allocated variable directly with its parent process so that the two processes can communicate directly using load and store instructions.

C: ☐ Each process has its own instance of user and kernel memory.

D: ☐ If a parent process has multiple running threads at the time of `fork()`, then the child process will have multiple running threads immediately after `fork()`.

E: ☐ Immediately after fork(), a child has a *duplicate* file descriptor table with pointers to *duplicated* file description structures for files that are open in the parent.

A: TRUE. The `execv()` *system call replaces the complete contents of the address space, including the heap. Consequently, there is no reason to free heap contents.*

B: FALSE. Since the parent and child process have separate, isolated memory spaces, simply sharing an address between parent and child is insufficient to allow sharing. You would also have to also setup shared memory (which would not normally involve the stack).

C: FALSE. There is only one kernel, shared among all processes.

D: FALSE. The `fork()` *system call only launches a single thread in the child, namely a duplicate of the thread that ran* `fork()` *in the parent.*

E: FALSE. Because the file descriptor table of the parent is copied in the child, all the file descriptors in the child point at the same file descriptions that they did in the parent. Consequently, this is false because the file descriptions are not duplicated.

Problem 2g[2pts]: Which of the following statements about files are true? (*choose all that apply):*

A: ☑ The same file descriptor number can correspond to different files for different processes.

B: ☐ The same file descriptor number can correspond to different files for different threads in the same process.

C: ☐ Reserved 0, 1, and 2 (`stdin`, `stdout`, `stderr`) file descriptors cannot be overwritten by a user program.

D: ☑ File descriptions keep track of the file offset.

E: ☑ An `lseek()` within one process may be able to affect the writing position for another process.

A: TRUE. Each process has a unique file descriptor table whose contents depends on the history of `open()` *and* `close()` *operations executed by threads in that process.*

B: FALSE. File descriptors are a process-level resources, so different threads in a process use the same (identical) file descriptor table.

C: FALSE. These file descriptors can be closed and reassigned easily using `dup2()`.

D: TRUE. The kernel tracks the file offset so that a sequential set of `read()` *operations will walk through the contents of the file automatically.*

E: TRUE. Since a parent and child share open file descriptions immediately after `fork()`, *an lseek() in one of these will change the file offset in the underlying file description and hence the writing position for the other process.*

Problem 2h[2pts]: Select all true statements about the x86 Calling sequence (*choose all that apply):*

A: ☐ Parameters are pushed onto the stack in the order that they are declared in the function.

B: ☐ Right before the Caller jumps to the Callee function, the ESP must be 16-byte alligned.

C: ☐ If padding is necessary for alignment, it should be added at a lower address than the parameters.

D: ☑ In the Callee, space is allocated for local variables via subtracting from the ESP.

E: ☐ None of the above.

A: FALSE. Parameters are pushed on the stack in reverse order.

B: FALSE. While it is true that the ESP should be 16-byte aligned during the prologue, it should already be at this alignment prior to executing the call instruction.

C: FALSE. Padding should be added at a higher address than the parameters.

D: TRUE. The compiler is able to calculate how much of the stack is used by local variables and subsequently subtracts the right amount from ESP.

E: FALSE. Obviously

Problem 2i[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. This is a required part of the monitor pattern.

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. Obviously

Problem 3: Synchronization [22pts]

Consider the following two threads, to be run concurrently in a shared memory (all variables are shared between the two threads):

Thread A	Thread B

`for (i=0; i<5; i++) {`
`    x = x + 1;`
`}`
	`for (j=0; j<5; j++) {`
	`    x = x + 1;`
	`}`

Assume a single-processor system, that load and store are atomic, and that x must be loaded into a register before being incremented (and stored back to memory afterwards). *Assume that x is initialized to zero before either Thread A or Thread B start.*

Problem 3a[3pts]: Give a *concise* proof why `x≠1` when both threads have completed. *Hint: consider the fact that each store is one greater than the value it loaded, even when threads interleave.*

The argument goes as follows:

1. *Each statement x=x+1 is guaranteed to store back a value that is one greater than it loaded.*
2. *Imagine the last store from any thread (i.e. the final value of x). By #1, the last value of x will be one greater than whatever was loaded by that thread in iteration 5 of its loop.*
3. *The last value loaded by this thread (to produce the last value) is the result of some other store, not the initial value 0: If we line up all of the stores prior to this load, we know that there are at least 4 of them (i.e. from the same thread). There may be others from the other thread. By #1, any store will store a value ≥ 1.*
4. *So – the last store is 1 more than the value it loaded, which must be at least 1, i.e. x ≥ 2*

Incidentally, although we didn't ask you this, there is actually an interleaving that produces x = 2!

Problem 3b[2pts]: What is a *critical section* and what is the role of locking in enforcing correct behavior for a multithreaded program?

A critical section is a sequence of operations which must be executed as a unit (without interleaving by multiple threads) in order to avoid incorrect behavior of the program. To enforce this correctness criteria, we can use a lock to prevent multiple threads from entering the critical section at the same time, i.e. we make threads `acquire()` the lock on entry to the critical section and `release()` the lock on exit of the critical section.

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)

`int TSET(&addr) {`
`  int result = M[addr];`
`  M[addr] = 1;`
`  return(result);`
`}`
	`int SWAP(&addr,expr) {`
	`  int result = M[addr];`
	`  M[addr] = expr;`
	`  return (result);`
	`}`
		`bool CAS(&addr,expr1,expr2) {`
		`  if (M[addr] == expr1) {`
		`     M[addr] = expr2;`
		`     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 implement a spin-lock acquire as follows:

```
int lock = 0;          // lock is free
while (TSET(&lock)); // Later: acquire lock
```

Problem 3c[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_________ ));
}
```

Problem 3d[3pts]: In class we argued that spinlocks were a bad idea because they can waste a lot of processor cycles (busy waiting). There is, however, one case we mentioned in which spinlocks would be more efficient than blocking locks. When is that? Can you say why the spinlock of **Problem 3c** might be even better in this circumstance than the standard spinlock built with TSET?

Spin locks might make sense in a multiprocessor application in which there are less active threads than cores. In that case, the time from releasing a lock to signaling a spinning thread to continue could be extremely fast. The lock in (3c) only writes if it can actually acquire the lock, which has better cache behavior that a basic test&set lock (much less cache coherence traffic).

Problem 3e[2pts]: Fill in the blanks, to construct a lock-free implementation of a singly-linked list "push" operation. *Hint: We need to retry if someone changes the root point out from under us.*

```
typedef struct node {
   node_t *next;
   data_t mydata;
} node_t;
void push(node_t **rootp, node_t *newnode) {
   do {
      node_t *oldfront = *rootp;
      newnode->next = oldfront;

   } while (!CAS(rootp, ____oldfront______, ___newnode________));
}
// Sample usage:
node_t *root = NULL;       // Empty list
push(&root, mynewnode);    // push new node onto list
```

The issue with constructing locks using only user-level atomic instructions such as TSET or CAS is that we cannot block a thread (i.e. put it to sleep) when it is unable to acquire a lock. 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. ~~You can assume in this problem that val will always be <= the actual number of waiting threads.~~

Problem 3f[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);
}
```

Problem 3g[2pts]: Under which circumstances is the lock implementation in **Problem 3f** suboptimal enough to look for a different implementation?

In a situation for which the lock is mostly uncontested because there is usually only one thread attempting to run in the critical section at a time, then the above code is not optimal: every release is forced to go to the kernel to see if there is a sleeping thread. Ideally, in an uncontested situation, both the `acquire()` and `release()` could happen entirely at user level as either a single TSET (for `acquire()`) or a single store (for `release()`).

Problem 3h[3pts]: To address the problem in **Problem 3g**, we can build a lock with three states:

```
typedef enum { UNLOCKED,LOCKED,CONTESTED } Lock;
Lock mylock = UNLOCKED;    // Initialize the lock in UNLOCKED state
```

Explain why we might want three states (rather than just LOCKED and UNLOCKED). In your explanation, make sure to say what each of the states indicate and under what circumstances the presence of the third state (`CONTESTED`) allows us to optimize performance.

A three-state lock could, in principle, track whether or not the lock was contested (i.e. multiple threads were trying to acquire it at the same time). Here our states represent:

- *UNLOCKED: No thread currently has the lock*
- *LOCKED: Only one thread has the lock (and no other threads are sleeping for sure).*
- *CONTESTED: The lock is locked there might be someone sleeping.*

As long as we are conservative about entering this third state (CONTESTED), we can avoid asking the kernel to see if there is someone to wake when we transition from LOCKED→UNLOCKED.

Problem 3i[3pts]: Fill in the missing blanks, below, for the three-state solution, using constants from the Lock enum. Hint: You are optimizing your lock based on the answer to **Problem 3h**:

```
typedef enum { UNLOCKED,LOCKED,CONTESTED } Lock;
Lock mylock = UNLOCKED; // Initialize the lock in UNLOCKED state

// Acquire a lock
void acquire(Lock *thelock) {
   // Fast case acquire:

   if (CAS(thelock, ________UNLOCKED__________, ________LOCKED____________))
      return;

   // Contested acquire:

   while (SWAP(thelock, ____CONTESTED_________) != _____UNLOCKED___________)
      futex(thelock, FUTEX_WAIT, CONTESTED);
}

// Release a lock
void release(Lock *thelock) {

   if (SWAP(thelock, _______UNLOCKED_________) == _____CONTESTED___________)
      futex(thelock,FUTEX_WAKE,1);
}
```

Problem 4: Short Answer Potpourri [24pts]

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

An interrupt is an asynchronous signal that comes from outside the processor pipeline. Assuming that a particular interrupt is enabled, the interrupt will cause the processor to stop what it is doing and atomically: swap in the kernel stack, save essential user registers on the new stack, change to kernel mode, and start executing a handler that is dependent on the interrupt ID. The interrupt controller is a piece of hardware that permits the OS to individually enable and disable interrupts and that will prioritize the set of enabled interrupts when there is more than one asserted interrupt.

Problem 4b[3pts]: The linked list nodes you have seen before (in 61B) stored each value alongside the previous and next pointers. However, this is not the case in PintOS. Below are the definition of the `list_elem` and `list` structures for PintOS:

```
struct list_elem
{
    struct list_elem *prev;
    struct list_elem *next;
};

struct list
{
    struct list_elem head;
    struct list_elem tail;
};
```

Provide justification for why PintOS lists are implemented this way. In an actual list built using these definitions, where are the related *values* stored?

Lists are implemented this way to provide a type independent, object-oriented (or "generic") list mechanism in C (which is decidedly not object-oriented). Lists and list elements can be manipulated by functions that do not need to know anything about the nodes that are being linked in the list. In an actual list built this way, the "list_elem" structure is embedded in an enclosing structure, which actually contains the values.

Problem 4c[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 switched automatically as a side-effect of changing out the address space and registers, since they stay in the same place in memory and do not actually have to be saved or restored.

Problem 4d[2pts]: What was the problem with the Therac-25 radiation therapy machine? Your answer should involve one of the topics of the class.

The Therac-25 radiation therapy machine was responsible for the deaths multiple cancer patients when it delivered excessive doses of radiation to these patients. Among other things, investigators discovered a race-condition/synchronization problem that caused this machine to malfunction when operators typed too quickly at the console (thereby interrupting the movement of internal mechanical components that controlled the radiation type and dosage).

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[2pts]: When a process thread executes fork(), this creates a new child process with an address space that is *separate* from the parent and that has *identical contents* to the parent process. Does an implementation of fork() require the operating system to copy all of the data of the parent process into the child process? Explain.

No, the OS does not have to copy all of the parent's data as long as it provides the same application behavior as if the contents of the parent's address space were fully copied. One possible way to do this is to use copy-on-write (COW): at the time of fork, we set all of the parent's page table entries to read-only before copying the page table for the child. To the child, this looks like a complete copy. And, if either the parent or child tries to write a page, we get a page fault and make two copies of that particular page – one for the parent and one for the child, maintaining the illusion that the child got an independent copy of the parent's address space.

Problem 4g[2pts]: When handling PintOS syscalls in `userprog/syscall.c`, how can we tell what syscall the user called, since there is only one syscall_handler function?

The first argument to the syscall_handler (pushed to the user stack in the interrupt stack frame) is an enum that specifies which syscall is needed. This is passed in from the lib/user/syscall.c file for each syscall.

Problem 4h[4pts]: Your friend tells you that they can open a file once but read it twice. Although you are skeptical, you decide to give it a try. Without utilizing another call to open, finish the following `double_read()` function. This function should read the specified file twice, storing the appended result as a duplicated, null-terminated string in the given buffer '`buffer`'. Note: *You must actually `read()` the file contents twice – do not just copy the data after reading the file once!*

In the following, assume that all system calls succeed and that the buffer is big enough to hold the result. Read the file in a maximum of CHUNK_SIZE bytes at a time. *While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.*

```
#define CHUNK_SIZE 1024
void double_read(char *filename, char *buffer) {
   int fd = open(filename, O_RDONLY);

   int offset = 0 ;

   int bytesread ;

   for (int pass = 0 ; pass < 2 ; pass++ ) {
      while(1) {

         bytesread = read(fd, buffer+offset, CHUNK_SIZE) ;

         if (bytesread == 0 ) break;

         offset += bytesread ;

      }
      lseek(fd, 0, SEEK_SET) ;
   }

   *(buffer+offset) = '\0' ;
   close(fd);
}
```

Problem 4i[3pts]: The "high-level" file I/O interface (i.e `fopen()`, `fclose()`, `fread()`, `fwrite()`, and other associated functions) is often called the "streaming I/O interface" in contrast to the "low-level" interface (i.e. `open()`, `close()`, `read()`, `write()`) . Why is this terminology/distinction appropriate? (*Hint: start by explaining what a streaming communication pattern might is).*

A streaming communication pattern is one that writes or reads a large sequential stream of bytes to or from a file using a (continuous) series of operations. It is distinguished from a random pattern of reads or writes. The key difference between the "high-level" and "low-level" interfaces is the presence of a user-level buffer in the high-level versions. Assuming a streaming I/O pattern, the high-level I/O operations can use the buffer gather together the data from a bunch of requests (from `fread()` or `fwrite()`) into a much smaller number of `read()` or `write()` operations to the kernel, thereby gaining in performance. However, this advantage is only present if the user-level buffer can be properly utilized, namely through a sequential "stream" of reads or writes that can lead to larger blocks of writes.

[This page intentionally left blank]

Problem 5: Stock Trading [16]

Stock trading is a very dynamic process with multiple traders simultaneously issuing buy and sell requests. As a result, any system that supports trading must deal with synchronization to provide correct behavior. In this problem, we will build a system to match sell requests to buyers. One essential element of our solution is that each particular stock has a `match_queue` that is used for coordinating the sales of that stock. We will build a fully synchronized solution using the monitor pattern with pthreads (i.e. using `pthread_mutex_t` and `pthread_cond_t` variables). Both sellers and buyers will be put to sleep while their corresponding trades are matched and executed.

Problem 5a[2pts]: We will represent a sell request with a `sell_request_t` structure and the queue of pending sell requests with a `match_queue_t` structure. Complete the following definitions. *Assume that we will use a PintOS list* to link sell requests into the `match_queue_t` structure. Further assume that buyers will sleep on one condition variable and sellers will sleep on a separate one, both of which are stored within the `match_queue`. Do not add any semicolons!

```
typedef struct sell_request {
   int waiting_sell;            // Remaining # shares for this seller
   struct list_elem mylink;     // Link for PintOS list
} sell_request_t;

typedef struct match_queue {
   char *stock_symbol;          // String describing stock
   int waiting_buy;             // Number waiting buyers

   struct list mlist ;

   pthread_mutex_t mlock ;

   pthread_cond_t sellwait ;

   pthread_cond_t buywait ;
} match_queue_t;
```

Problem 5b[2pts]: Complete the following `match_queue` allocator, assuming calloc succeeds). Please do not add any semicolons. Error codes for syscalls and pthread functions do not need to be checked.

```
match_queue_t *match_queue_alloc(char *stock_symbol) {
   match_queue_t *new_match_queue =
      (match_queue_t *)calloc(1,sizeof(match_queue_t));
   new_match_queue->stock_symbol = stock_symbol;

   list_init(&(new_match_queue->mlist)) ;

   pthread_mutex_init(&(new_match_queue->mlock)) ;

   pthread_cond_init(&(new_match_queue->sellwait)) ;

   pthread_cond_init(&(new_match_queue->buywait)) ;
   return new_match_queue;
}
```

Problem 5c[6pts]: Fill in the missing blanks for the buyer's routine, below. Assume that `buy_stock_match()` should not return until the total requested shares have been matched with a seller. However, busy-waiting is strictly forbidden. Make sure that sellers (represented by sell_request_t structures with a non-zero sell_waiting value are handled strictly in order. Error codes for syscalls and pthread functions do not need to be checked. Note: The matchq argument represents a pointer that has gone through your match_queue_alloc() routine, so should be properly initialized. *While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.*

Hint: The buyers and sellers work together. Thus, as you will handle in **Problem 3d**, sellers will sleep waiting for their shares to be completely matched with buyers. And buyers will sleep waiting for enough shares to become available.

```
// Buy num_shares of a stock through the given match_queue
// Assume num_shares > 0 and matchq is valid (from match_queue_alloc())
void buy_stock_match(match_queue_t *matchq, int num_shares) {

   pthread_mutex_lock(&(matchq->mlock)) ;
   while(num_shares) {
      if (list_empty(&(matchq->mlist) )) {
         matchq->waiting_buy += 1;

         pthread_cond_wait(&(matchq->buywait), &(matchq->mlock)) ;
         matchq->waiting_buy -= 1;
      } else {
         struct list_elem *fe = list_pop_front(&(matchq->mlist) );
         sell_request_t *first =
            list_entry(fe , sell_request_t , mylink );
         if (first->waiting_sell > num_shares) {

            first->waiting_sell -= num_shares ;

            num_shares = 0 ;

            list_push_front(&(matchq->mlist), fe) ;
         } else {
            num_shares -= first->waiting_sell ;

            first->waiting_sell = 0 ;

            pthread_cond_broadcast(&(matchq->sellwait) ;
         }
      }
   }
   pthread_mutex_unlock(&(matchq->mlock) ;

   ________________________________________________________________________;
}
```

Problem 5d[4pts]: Fill in the missing blanks for the seller's routine. Assume that calloc() always succeeds and that the seller should be put to sleep until all of their stock shares have been matched to buyers. Make sure that new sellers are not allowed to jump in front of existing sellers. Error codes for syscalls and pthread functions do not need to be checked. Note: The matchq argument represents a pointer that has gone through your match_queue_alloc() routine, so should be properly initialized. Busy-waiting is not allowed. Further, you should make sure that there are no memory leaks (it is up to the seller to free any sell structures they have allocated). *While you do not necessarily need to use every line here, you are also not allowed to add semicolons to existing lines.*

```
// Sell num_shares of a stock through the given match_queue
// Assume num_shares > 0 and matchq is valid (from match_queue_alloc())
void sell_stock_match(match_queue_t *matchq, int num_shares) {
   sell_request_t *req =
      (sell_request_t *)calloc(1, sizeof(sell_request_t));
   req->waiting_sell = num_shares;

   pthread_mutex_lock(&(matchq->mlock)) ;

   list_push_back(&(matchq->mlist), &(req->mylink) );
   if (matchq->waiting_buy) {

      pthread_cond_broadcast(&(matchq->buywait) ;
   }
   while (req->waiting_sell ) {

      pthread_cond_wait(&(matchq->sellwait), &(matchq->mlock)) ;
   }
   free(req) ;

   pthread_mutex_unlock(&(matchq->mlock) ;
}
```

Problem 5e[2pts]: Suppose that many sellers show up all at once, before buyers arrive. Then, suppose buyers arrive one at a time. Although still correct, the above solution will incur a lot of scheduler overhead. Explain the problem in three sentences or less and how you would fix it.

The problem here is that all sellers are treated identically – regardless of whether or not their shares have been all sold. So, each time that a sell_request_t is emptied (because its num_shares→ 0), we wake up all sellers (using a broadcast), all but one of which will check their sell structure, notice that it still contains shares, and go to sleep. We have to use broadcast because there are no guarantees of the order which threads will be woken up from a condition variable.

We fix this by putting a condition variable inside each sell_request_t. As a result, we can directly wake up only the buyer whose shares have been freed.

[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!]