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

Spring 2024 John Kubiatowicz

Midterm II
SOLUTION

March 14th, 2024
CS162: Operating Systems and Systems Programming

Your Name:	
Your SID:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 2 pages of notes (both sides). You have 2 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming. Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	18	
2	16	
3	14	
4	13	
5	19	
6	20	
Total	100	

[Happy π Day!]

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]: With the Round Robin scheduling policy, the larger the time quantum, Q, the higher the throughput of execution.

[✓ True] ☐ False

Explain: *With a larger time quantum, threads are interrupted less frequently. Consequently, there is less overhead (saving and restoring of registers, pipeline interruption/draining) and processor mechanisms for single-threaded performance have more chance to work (caching, branch prediction, etc.)*

Problem 1b[2pts]: If the Banker's algorithm finds that it's safe to allocate a resource to an existing thread, then all threads will eventually complete.

☐ True [✓ False]

Explain: *Banker's algorithm only prevents deadlocks caused by resource-related dependency cycles. Threads could still fail to complete for a variety of other reasons (such as entering an infinite loop).*

Problem 1c[2pts]: Suppose that a shell program wants to execute another program and wait on its result. It does this by creating a thread, calling `exec()` from within that thread, then waiting in the original thread.

☐ True [✓ False]

Explain: *If the shell only creates a new thread, then calling `exec()` within that thread will completely replace the shell with the new program (and hence there won't be an original thread to do any waiting). The shell program needs to use `fork()` to create a new process in which to call `exec()`, not a new thread.*

Problem 1d[2pts]: Because the CFS scheduler uses the same mechanism to choose the next running thread regardless of whether a thread is interactive or computational, it has poor interactive behavior.

☐ True [✓ False]

Explain: *Interactive threads have short bursts and spend most time sleeping; consequently they will not accumulate as much virtual time as computational threads and will thus run preferentially when the are ready to run (e.g. when a user has just pressed a key on their keyboard).*

Problem 1e[2pts]: A 2-way set-associative cache has a faster hit time than a direct-mapped cache.

☐ True [✓ False]

Explain: *Unlike a direct-mapped cache, the 2-way set-associative cache must (1) compare tags after the cache-bank lookup using the index, (2) use the result to select the direction of a mux and (3) wait for the data to route through the mux. All of these combine to make a 2-way set-associative cache slower than a direct-mapped cache.*

Problem 1f[2pts]: The Shortest Remaining Time First (SRTF) algorithm is the best preemptive scheduling algorithm that can be implemented in an Operating System.

☐ True [✓ False]

Explain: *SRTF cannot be implemented, only approximated (it relies on perfectly predicting the future). Consequently, this statement is false regardless of your definition of "best."*

Problem 1g[2pts]: The size of an inverted page table grows with the size of the virtual address space that it is supporting.

☐ True [✓ False]

Explain: *The inverted page table size grows with the size (in pages) of the physical memory. This is because it only needs to hold the PTEs that are marked "valid" or "present". During virtual⇒physical mapping we can treat a missing entry as if it is an invalid PTE.*

Problem 1h[2pts]: Anything that can be done with a monitor can also be done with semaphores.

[✓ True] ☐ False

Explain: *This is true because you can implement your own lock(), wait(), signal() operations using semaphores, then insert them into you algorithm. The crucial aspect is that condition variables can be implemented by combining a semaphore with an extra integer (use to count number of sleeping threads, synchronized using monitor mutex).*

Problem 1i[2pts]: Page Tables have an important advantage over simple Segmentation Tables (i.e. using base and bound) for virtual address translation in that they eliminate external fragmentation in the physical memory space.

[✓ True] ☐ False

Explain: *Since every page in physical memory is exactly the same size, they are all interchangeable when placed into page tables. Consequently, free holes in physical memory can always be used regardless of how long the system have been running; i.e. there is no external fragmentation.*

Problem 2: Multiple Choice [16pts]

Problem 2a[2pts]: When a process asks the kernel for resources that cannot be granted without risking deadlock (as determined by the Banker's algorithm), the kernel must (*choose one)*:

A: ◯ Preempt the requested resources from another process and give them to the requesting process, thereby avoiding cycles in the resource dependency graph.

B: [✓ ⊗] Put the requesting process to sleep until the resources become available.

C: ◯ Send a SIGKILL to the requesting process to prevent deadlock from occurring.

D: ◯ This question does not make sense because the Banker's algorithm is run only when a new process begins execution.

E: ◯ Reboot the system, since there is no way to avoid deadlock

A: FALSE. Preempting resources from a process could lead to incorrect behavior.
B: TRUE. At each resource request, the Banker's Algorithm works by simulating what would happen if requested resources were granted, i.e. whether granting requests would lead to an UNSAFE state. If an UNSAFE state would result, the Banker's Algorithm would have to put the thread to sleep, since it cannot grant the resources (and the thread would have an incorrect understanding of its resources if it ran).
C: FALSE. The Banker's algorithm aims to avoid deadlock and allow all threads to complete.
D: FALSE. The Banker's algorithm runs on each resource request, not just at the start of a process.
E: FALSE. The Banker's algorithm puts threads to sleep before the system reaches a state in which deadlock is unavoidable.

Problem 2b[2pts]: Suppose that two chess programs are running against one another on the same CPU with equal "nice" values. Also assume that the programs are given real-time limits on the total time they spend making decisions (similar to a real chess tournament). Why might one of the programs want to perform a lot of superfluous I/O operations? (*choose one):*

A: ◯ The extra I/O operations fool the virtual memory system into giving extra physical memory to the chess program, thereby speeding up its overall execution and allowing great search depth in the fixed time.

B: ◯ The random completion time for these I/O operations will serve as an important source of randomness, thereby improving the chess playing heuristics.

C: [✓ ⊗] This will split up the long-running heuristic computation into many short bursts, thereby causing the chess program to be classified as "interactive" by the scheduler and thus receiving higher priority than the competing program.

D: ◯ This question does not make sense, since the extra I/O operations will only slow down chess program.

E: ◯ To hide the fact that the program was cheating by receiving information from a real chess master over the network.

A: FALSE. The paging heuristics used by the memory system either do not take I/O into account or .
B: FALSE. There are better sources of randomness, so this is not the best answer
C: TRUE. In class, we talked about a classic case of a contest between Othello-playing programs in which one of the contestants broke up their execution by sprinkling printf operations throughout the code, thereby getting the scheduler to misclassify the program and run it at higher priority.
D: FALSE. I/O operations shorten overall burst time which will raise priority for some schedulers.
E: FALSE. Presumably this option would be prevented by restricting outside network connections.

Problem 2c[2pts]: Which of the following are true about priority inversion? (*choose all that apply):*

A: ☐ Priority inversion could not occur if there were only two possible priorities for threads.
B: [✓] Priority donation is a solution to priority inversion.
C: [✓] Priority inversion could result in starvation of high priority threads.
D: ☐ Priority inversion is only a problem on a multi-core system.
E: [✓] Priority inversion would not happen in a first come first serve scheduler.

A: FALSE. We showed a two-thread inversion in class: If a low-priority thread grabs a lock, and a high-priority threads spins in a loop waiting for the lock to be freed, we have a priority inversion.
B: TRUE. Priority donation makes sure that the priority of a thread waiting for a lock is temporarily donated to a low-priority lock holder, thus preventing the low priority thread from indefinitely holding up a high-priority thread.
C: TRUE. By definition, priority inversion is a form of starvation of a high-priority thread.
D: FALSE. Priority inversion can happen with a single core.
E: TRUE. Since an FCFS scheduler has no priorities (or only a single priority), there can be no priority inversion.

Problem 2d[2pts]: Assume we make a single memory access on a processor which has no caches and no TLB. All else being equal, which of the following are true? (*choose all that apply):*

A: [✓] Base-and-bound (with translation) will, on average, be faster than a single-level page table.
B: ☐ An inverted page table will, on average, be faster than a single-level page table.
C: [✓] A single-level page table will, on average, be faster than a multi-level page table.
D: ☐ A multi-level page table will, on average, be faster than an inverted page table.
E: ☐ None of the above

A: TRUE. Translation using a single-level page table requires one more memory access (for the page table lookup) than base-and-bound (with translation).
B: FALSE. Allowing for collisions in the hash, the inverted page table will require slightly more than one access for each lookup, thus being slower than the single-level page table (or if there are no collisions, the inverted page table will be as fast, making this FALSE still).
C: TRUE. The multi-level page table will require more than one access to memory for each lookup, while the single-level lookup requires only a single access.
D: FALSE. The inverted page table will not, in general, encounter a hash collision for every lookup, thus requiring less than two accesses on average for each lookup.
E: FALSE. For obvious reasons.

Problem 2e[2pts]: Earliest Deadline First (EDF) scheduling has an important advantage over other scheduling schemes discussed in class because (*choose one*):

A: ○ It can hand out more total processor cycles to an asynchronously arriving mix of real-time and non-realtime tasks than other scheduling schemes.
B: [✓] It schedules tasks based on deadlines and can thus provide realtime guarantees that other schemes we discussed in class cannot.
C: ○ It can operate non-preemptively and is thus simpler than many other scheduling schemes.
D: ○ It can provide the lowest average responsiveness to a set of tasks under all circumstances—even in the presence of long-running computations.
E: ○ Because it sorts the ready queue by deadline, it can handle oversubscription of the CPU better than other schemes we discussed in class.

A: FALSE. EDF cannot schedule more processor cycles than exist. In fact, because FCFS is non-preemptive, it provides higher instruction throughput (processor efficiency) than EDF.
B: TRUE. Assuming that the set of threads satisfy the scheduling sufficiency criteria, the EDF scheduler can guarantee that it can meet the deadlines for threads, thus providing realtime guarantees. None of the other schedulers we discussed in provide such guarantees.
C: FALSE. EDF is a preemptive scheduler.
D: FALSE. EDF guarantees that deadlines are met, which is a different requirement than responsiveness.
E: FALSE. No scheduler handles oversubscription well. In fact, EDF ceases to provide guarantees and suffers from a type of cascading deadline failure when the CPU is oversubscribed.

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

A: ☑ SRTF does not suffer from the Convoy effect.
B: ☑ The CFS scheduler implemented in Linux takes O(log n) time to pick the next task to run (where "n" is the total number of threads on the ready queue).
C: ☐ FCFS provides an optimal (i.e. minimum) total completion time (wall clock time) for any set of tasks.
D: ☑ Multi-level schedulers can provide good interactive responsiveness by modeling future behavior of threads.
E: ☑ A strict priority scheduler can experience starvation.

A: TRUE. The Convoy effect occurs when a short-burst thread gets scheduled ("stuck") behind a thread with a long burst. By definition, SRTF won't let that happen.
B: TRUE. The CFS scheduler in Linux uses a red-black tree to sort tasks by their virtual time. Operations in RB trees are `O(Log n)`, including finding and removing the smallest element. NOTE: Although the answer is TRUE, we also accepted FALSE because there have been confusing answers in previous test solutions and class slides.
C: We accepted both TRUE and FALSE, since "total completion time" is confusing:
TRUE. If "total completion time" just means time to complete all tasks, then this true, since FCFS makes the fewest context switches/highest throughput execution.
FALSE. If "total completion time" is the sum of completion times (part of computing "average completion time"), then FCFS will provide the opposite of optimal completion time (i.e. maximal time) if tasks arrive in reverse order of their burst times (i.e. longest first).
D: TRUE. Multilevel schedulers try to figure out the historical burst time for threads by floating short running threads to the top queues and long running threads to the bottom queues. It thus predicts the future behavior of threads based on the queue they are in.
E: TRUE. The continuous arrival of high priority threads can prevent the execution of lower-priority threads, thus starving the low-priority threads.

`**Problem 2g[2pts]:** Debugging in Pintos can often be very difficult due to the complexity of the code base and often just staring at the code is an ineffective strategy. Which of the following are good strategies for debugging in Pintos? (*choose all that apply):*

A: ☑ Setting breakpoints in GDB to walk through specific function calls.
B: ☑ Setting memory watchpoints in GDB to watch for variable modifications.
C: ☑ Checking function pre- and post-conditions with ASSERT statements.
D: ☑ Checking execution progress with PANIC statements.
E: ☑ Checking variables with print statements.

A-E: TRUE. These are all valuable debugging techniques.

Problem 2h[2pts]: Consider a computer system with the following parameters:

Variable	Measurement	Value
P_{TLB}	Probability of TLB miss	0.1
P_F	Probability of a page fault when a TLB miss occurs on user pages (assume page faults do not occur on page tables).	0.0002
P_{L1}	Probability of a first-level cache miss for all accesses	0.1
T_{TLB}	Time to access TLB (hit)	1 ns
T_{L1}	Time to access L1 cache (hit)	5ns
T_M	Time to access DRAM	100ns
T_D	Time to transfer a page to/from disk	10 ms = 10,000,000 ns

You can assume that TLB lookup *does not occur in parallel with cache access.* You can also assume that the TLB is refilled automatically by the hardware on a miss. The 2-level page tables are kept in physical memory and are cached like other accesses. Assume that the costs of the page replacement algorithm and updates to the page table are included in the T_D measurement. Also assume that no dirty pages are replaced and that pages mapped on a page fault are not cached.

What is the effective access time (the time for an application program to do one memory reference) on this computer? Assume physical memory is 100% utilized and ignore any software overheads in the kernel. (*choose one):*

A: ○ $T_{L1} + P_{L1} \times T_M + P_{TLB} \times (T_{TLB} + P_F \times T_D)$

B: ○ $T_{TLB} + (T_{L1} + P_{L1} \times T_M) + P_{TLB} \times \{2(T_{L1} + P_{L1} \times T_M) + P_F \times T_D\}$

[✓ C:] $T_{TLB} + (1 - P_{TLB}) \times (T_{L1} + P_{L1} \times T_M) +$
$P_{TLB} \times \{2(T_{L1} + P_{L1} \times T_M) + (1 - P_F) \times (T_{L1} + P_{L1} \times T_M) + P_F \times (T_{L1} + T_M + T_D)\}$

D: ○ $T_{TLB} + (T_{L1} + P_{L1} \times T_M) + P_{TLB} \times (2T_M + P_F \times T_D)$

E: ○ $(1 - P_{TLB}) \times (T_{L1} + P_{L1} \times T_M) + P_{TLB} \times 2T_M + P_F \times T_D$

C: TRUE. Since all accesses have to perform a successful TLB lookup (either before or after handling the TLB miss), we have one T_{TLB} time guaranteed. Then, we have two options: (1) we hit in the TLB with probability (1-P_{TLB}) or (2) miss in the TLB with probablility P_{TLB}. In the first case, data access time is the AMAT for a one-level cache: $(T_{L1} + P_{L1} \times T_M)$. In the second case, data access time includes page-table lookup time (i.e. two cached accesses to page table structures, $2(T_{L1} + P_{L1} \times T_M)$), and one of two options for subsequent access: (A) we do not have a page fault, with probability (1-P_F), or (B) we have a page fault, with probability P_F. In case (A), data access time is just the one-level AMAT: $(T_{L1} + P_{L1} \times T_M)$. In case (B), data access time involves the time to load from Disk⇒Memory (T_D), then from Memory⇒Cache (T_M), then from Cache⇒Processor (T_{L1}).

Problem 3: Scheduling [14pts]

Problem 3a[3pts]: How could a strict priority scheduler be used to emulate Earliest Deadline First (EDF) scheduling? How often would you need to compute priorities (assuming that we schedule periodic tasks characterized by period T and computational time of C)? *[Explain with no more than two sentences per question.]*

At any given point in time, the set of active threads need to be sorted by the order of their deadlines, and assigned priorities in deadline order (i.e. earlier deadline⇒higher priority), one priority per thread.

The priority order would need to be recomputed whenever the set of deadlines changes, namely whenever a deadline expires [the thread would be removed from the scheduler and reinserted with a deadline that is later than the current time by its particular value of T]. Note that an efficient structure such as a red-black tree would make resorting an O(log N) operation.

Problem 3b[3pts]: How does the Linux CFS ("Completely Fair Scheduler") scheduler decide which thread to run next? What aspect of its behavior is "fair"? (You can ignore the presence of priorities or "nice" values in your answer): *[Explain with no more than three sentences.]*

The Linux CFS keeps all runnable but paused threads (i.e. those on the ready queue) sorted by their virtual time, namely the amount of CPU time they have run so far. To pick a new thread to run, it simply picks the runnable thread with the smallest virtual time. (Note: in class we discussed the fact that Linux sorts threads in a red-black tree, but we didn't require you to say this).

Ignoring "nice", CFS provides a version of "fair" in which every runnable thread is given the same fraction of CPU cycles, almost like we evenly split the CPU into N chunks for N threads.

Problem 3c[2pts]: Name two differences between *interrupts* and *synchronous exceptions* (such as page faults)? *Provide only one sentence per difference:*

There are a number of things you could mention. Here are three:

1. *Interrupts are asynchronous (occurring between instructions) while synchronous exceptions are synchronous (occurring at the same place in the instruction stream every time).*
2. *Interrupts can be disabled with the interrupt disable mechanisms, whereas synchronous exceptions cannot be disabled.*
3. *Interrupts originate from events external to the pipeline, whereas synchronous exceptions represent conditions originating locally to the pipeline.*

Problem 3d[6pts]:
Consider the following table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process A	0	2
Process B	1	6
Process C	4	1
Process D	7	4
Process E	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1. *Assume that context switch overhead is 0, that new processes are available for scheduling as soon as they arrive, and that new processes are added to the* ***head*** *of the queue except for FCFS, where they are added to the tail.*

Time Slot	FCFS	SRTF	RR
0	*A*	*A*	*A*
1	*A*	*A*	*B*
2	*B*	*B*	*A*
3	*B*	*B*	*B*
4	*B*	*C*	*C*
5	*B*	*B*	*B*
6	*B*	*B*	*B*
7	*B*	*B*	*D*
8	*C*	*B*	*E*
9	*D*	*E*	*B*
10	*D*	*E*	*D*
11	*D*	*E*	*E*
12	*D*	*D*	*B*
13	*E*	*D*	*D*
14	*E*	*D*	*E*
15	*E*	*D*	*D*

Problem 4: DataBase Access [13pts]

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

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

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

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

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

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

Problem 4a[3pts]: Above, we show a *modified* version of the Readers-Writers example given in class. It uses two condition variables, one for waiting readers and one for waiting writers. Suppose that *all* of the following requests arrive in very short order (while W_1 is still executing):

Incoming stream: W_1 R_1 R_2 R_3 W_2 W_3 R_4 R_5 R_6 W_4 R_7

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).

W_1 { R_1 R_2 R_3 R_4 R_5 R_6 R_7 } W_2 W_3 W_4

Problem 4b[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 R_1 R_2 R_3 W_2 W_3 R_4 R_5 R_6 W_4 would be processed as W_1{R_1 R_2 R_3} W_2 W_3 {R_4 R_5 R_6} W_4 *regardless of the speed with which these requests arrive.* Explain why the above algorithm does not satisfy this constraint.

The above code does not satisfy this constraint because it gives priority to servicing reads over writes: If a write comes in while any type of access is going on or while readers are waiting, it puts the write to sleep. Writes will subsequently get to run only when there are no readers either running or sleeping. Consequently, reads are reorder before writes (as shown in answer to 4a).

Problem 4c[8pts]: Suppose we wanted a logical processing order as defined in **Problem 4b**. 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, as defined by the order in which requests acquire the lock at line #7. Rather than separate methods for `Reader()` and `Writer()`, we provide a single access method, `DBAccess()` which takes a `type` argument (0 for read, 1 for write).

Let's continue to assume that our system uses Mesa scheduling and that condition variables have FIFO wait queues. The `waitQueue` condition variable will keep *unexamined* requests in FIFO order. The `onDeckQueue` keeps a *single* request that is currently incompatible with requests that are executing. Complete the DBAccessor code, below, assuming that you want to allow as many parallel requests to the database as possible, subject to the constraint of obeying logical arrival order and maintaining the readers/writers correctness constraint. You do not have to use every blank line, but you are also not allowed to add additional semicolons (';') to any of the lines either.

```
 1. Lock MonitorLock;                    // Methods: acquire(),release()
 2. Condition waitQueue, onDeckQueue;    // Methods: wait(),signal(),broadcast()
 3. int numQueued = 0, onDeck = 0;       // Counts of sleeping threads
 4. int curType = 0, numAccessing = 0;   // Info about threads in DataBase.

 5. DBAccess(int type) {               // type = 0 for read, 1 for write
 6.     /* Monitor to control entry so that one writer or multiple readers */
 7.     MonitorLock.acquire();
 8.     if (numQueued > 0 || onDeck > 0                                    ) {
 9.        numQueued++;
10.        waitQueue.wait();
11.        numQueued--;
12.     }
13.     while ( (numAccessing > 0) && (type == 1 || curType == 1) )         {
14.        onDeck++;
15.        onDeckQueue.wait();
16.        onDeck--;
17.     }
18.     numAccessing++                                                      ;
19.     curType = type                                                      ;
20.     waitQueue.signal()                                                  ;
21.     MonitorLock.release();

22.     // Perform actual data access
23.     AccessDatabase(type);

24.     // Thread exit code
25.     MonitorLock.acquire();
26.     numAccessing--                                                      ;
27.     onDeckQueue.signal()                                                ;
28.     ____________________________________________________________________;
29.     MonitorLock.release();
30. }
```

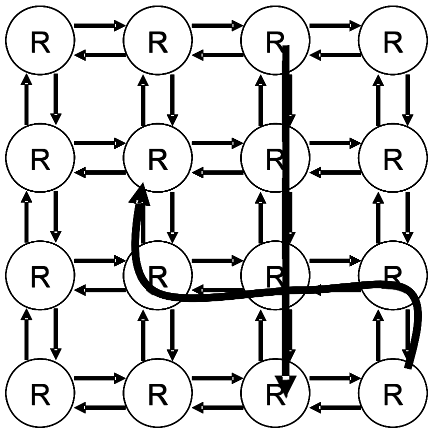
NOTES: We want to group requests in arrival order, so line 8 will queue a request on `waitQueue` if there are any requests in front of it that aren't executing yet (either in `waitQueue` or in `onDeckQueue`). Consequently, requests access the database in the same order they acquire the lock in line 7. In line 13, we hold up one request if it is incompatible with the group (one or more) of requests currently executing. To perform that check, we keep track of the current type of the executing group (using '`curType`').

Other NOTE: The above solution uses a "`while`" loop at line 13. An alternate solution would be to use an "`if`" instead of "`while`" on line 13, then change line 27 to:

```
if (numAccessing == 0) onDeckQueue.signal()
```

Problem 5: Deadlock [19pts]

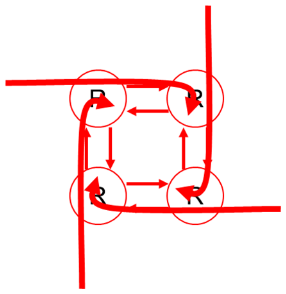
Problem 5a[2pts]:



The figure at the right illustrates a 2D mesh of network routers. Each router is connected to each of its neighbors by two network links (small arrows), one in each direction. Messages are routed from a source router to a destination router and can stretch through the network (i.e. consume links along the route from source to destination). Messages can cross inside routers.

Assume that no network link can service more than one message at a time, and that each message must consume a continuous set of channels (like a snake). Messages always make progress to the destination and never wrap back on themselves. The figure shows two messages (thick arrows).

Assume that each router or link has a very small amount of buffer space and that each message can be arbitrarily long. Show a simple situation (with a drawing) in which messages are deadlocked and can make no further progress. *[Explain with no more than two sentences]:*



Answer: The simplest deadlock example is a set of four messages in a loop (as shown in the figure at the left). Each message is blocked attempting to make a counter-clockwise turn by a message utilizing the target channel.

Problem 5b[3pts]:

Define a routing policy that avoids deadlocks in the network of **5a**. Sketch a proof that deadlocks cannot happen when using this routing policy. *Hint: assume that a deadlock occurs and show why that leads to a contradiction in the presence of your new routing policy.*

The simplest policy was given in class: every message must route along the X direction first, then along the Y direction. (The is called "dimension order routing" but you weren't required to name it).

Proof that this is deadlock free by contradiction: assume there is a deadlock cycle. That would mean that there is a set of messages deadlocked in a cycle. However, such a cycle would involve at least one message that routed from the Y direction to the X direction, which is disallowed by our policy. Consequently, there is no deadlock.

Problem 5c[3pts]: Pwnage Games decided to purchase Super Smash Bros. for Wii U -- a popular fighting video game -- in the hope that it would draw customers to the business. However, due to limited resources, the store could only buy one copy of the game. Luckily, the owners know Gill Bates -- a Cal EECS undergrad – who figured out how to allow multiple consoles to play the game at the same time. Unfortunately, she is forced to impose some conditions on the gameplay:

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

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

```
 1. pthread_mutex_t barracks_lock;  // Global lock, initialized elsewhere
 2. typedef struct fighter {
 3.    char *name;
 4.    pthread_mutex_t lock;
 5.    ...
 6. } fighter_t;
 7. void smash(fighter_t* first, fighter_t* second) {
 8.    pthread_mutex_lock (&first->lock);
 9.    pthread_mutex_lock (&second->lock);
10.    fight (first, second);
11.    pthread_mutex_unlock (&second->lock);
12.    pthread_mutex_unlock (&first->lock);
13. }
```

Despite Gill's effort, her algorithm has an obvious flaw: `smash()` can lead to deadlock! Present an example of how this can happen. *Explain with no more than two sentences.*

A simple example is one in which two consoles request the same two fighters (as defined by their `fighter_t*` pointers) but in opposite orders. If the scheduler allows the first console to lock the first fighter and the second console to lock the second fighter, then the system will be deadlocked.

Problem 5d[3pts]: *Adding no more than two lines*, redesign the `smash()` function to avoid deadlock. Make sure that multiple game consoles can still play at the same time. You can assume the presence of a global lock, called "`barracks_lock`", that has been initialized. Write your new version in the space below, ***then explain in one sentence how one of the four conditions of deadlock is eliminated by your solution.*** (You can copy a line or lines from **5c** by writing "Line X" or "Lines X-Y", but must include all of lines 8—12 from original function).

```
void smash(fighter_t* first, fighter_t* second) {

   pthread_mutex_lock(&barracks_lock);
   Lines 8-9
   Pthread_mutex_unlock(&barracks_lock);
   Lines 10-12


}
```

This solution eliminates "circular wait" because the only time that consoles would wait to acquire locks on players would be after *they already hold the `barracks_lock`. Since there can only be one thread holding the `barracks_lock` at a time, there is no way to have two threads in a cyclic waiting pattern.*

Problem 5e[2pts]: Assume that *fighter's names are unique* and that you have access to a string comparison function with the following signature:

```
// Returns: 0 if s1==s2, -1 if s1 < s2, and 1 if s1 > s2
int strcmp(char *s1, char *s2);
```

Write a deadlock free version of `smash()` that does not need to use the extra lock of **5d** by filling in the blank lines below. You do not need to use all lines and must not add extra semicolons (";").

```
void smash(fighter_t* first, fighter_t* second)
{
   if (strcmp(first->name, second->name) < 0) {
      pthread_mutex_lock (&first->lock);
      pthread_mutex_lock (&second->lock);
   } else {
      pthread_mutex_lock(&second->lock);
      pthread_mutex_lock(&first->lock);
   }
   fight (first, second);
   pthread_mutex_unlock (&second->lock);
   pthread_mutex_unlock (&first->lock);
}
```

Problem 5f[1pt]: Which version of your code (i.e. **5d** or **5e**) behaves better and why? *[No more than two sentences]:*

The version of code in 5e behaves better than the code in 5d because the only time consoles are blocked from playing in 5e is when they try to request a player that is currently in use by another console. In contrast, the code in 5d could block a console requesting players that are not in use by anyone simply because it cannot acquire the `barracks_lock`, which is held by some other console waiting for a player to be available.

Problem 5g[2pts]: Suppose that we utilize the Banker's algorithm to determine whether or not to grant resource requests to threads. The job of the Banker's algorithm is to keep the system in a "SAFE" state. It denies resource requests by putting the requesting thread to sleep if granting the request would cause the system to enter an "UNSAFE" state, waking it only when the request could be granted safely. What is a SAFE state? *[Define in no more than two sentences]:*

A SAFE state is one in which there exists a deadlock-free schedule for all threads to complete – without requiring threads to prematurely give up the resources they already have and regardless of their future patterns of allocation requests (not exceeding their specified maximum needs).

(In contrast, an UNSAFE state would be one that is either deadlocked or could deadlock from some pattern of allocation requests that fall within the specified maximums given by threads).

Problem 5h[3pts]: Suppose that we have the following resources: A, B, C and threads T1, T2, T3, T4. The total number of each resource is:

Total		
A	B	C
12	9	12

Further, assume that the processes have the following maximum requirements and current allocations:

Thread ID	Current Allocation			Maximum		
	A	B	C	A	B	C
T1	2	1	3	4	9	4
T2	1	2	3	5	3	3
T3	5	4	3	6	4	3
T4	2	1	2	4	8	2

Is the system in a safe state? If "yes", show a non-blocking sequence of thread executions. Otherwise, provide a proof that the system is unsafe. Show your work and justify each step of your answer.

Answer: Yes, this system is in a safe state.
To prove this, we first compute the currently free allocations:

Available		
A	*B*	*C*
2	*1*	*1*

Further, we compute the number needed by each thread (Maximum – Current Allocation):

Thread ID	*Needed Allocation*		
	A	*B*	*C*
T1	*2*	*8*	*1*
T2	*4*	*1*	*0*
T3	*1*	*0*	*0*
T4	*2*	*7*	*0*

Thus, we can see that a possible sequence is: T3, T2, T4, T1:

Thread ID	*Available Before*			*Needed Allocation*			*Current Allocation*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T3	*2*	*1*	*1*	*1*	*0*	*0*	*5*	*4*	*3*
T2	*7*	*5*	*4*	*4*	*1*	*0*	*1*	*2*	*3*
T4	*8*	*7*	*7*	*2*	*7*	*0*	*2*	*1*	*2*
T1	*10*	*8*	*9*	*2*	*8*	*1*	*2*	*1*	*3*

Problem 6: Address Translation [20 pts]

Problem 6a[2pts]: In class, we discussed the "magic" address format for a multi-level page table on a 32-bit machine, namely one that divided the address as follows:

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

You can assume that Page Table Entries (PTEs) are 32-bits in size in the following format:

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

What is particularly "magic" about this configuration? Make sure that your answer explains why this configuration is helpful for an operating system attempting to deal with limited physical memory.

Each page is 4K in size (12-bit offset $\Rightarrow 2^{12}$ bytes). Because the PTE is 4-bytes long and each level of the page table has 1024 entries (i.e. 10-bit virtual page #), this means that each level of the page table is 4K in size, i.e. exactly the same size as a page. Thus the configuration is "magic" because every level of the page table takes exactly one page. This is helpful for an operating system because it allows the OS to page out parts of the page table to disk.

Problem 6b[2pts]: Suppose you were interested in supporting more than 4GB (i.e 2^{32} bytes) of physical memory, while still supporting a 32-bit virtual address space. Further, suppose that you wanted to stick with 4-byte PTEs with the same *hardware-defined* page control and status bits as defined above. What is the maximum amount of physical memory you could easily support? Explain.

Ignoring the Physical Page # bits, there are 8 access control bits above, from "Large Page" down to "Valid" and 4 other bits that are either software defined or treated as 0. So, we could add up to 4 more bits to the physical page #, yielding a maximum number of physical pages = $2^{24} \approx 16$ million entries. So, with 4KB pages, our total physical memory would be: $2^{24} \times 2^{12} = 2^{36} = 64$GiB

Problem 6c[2pts]: As discussed in lecture, the Translation Lookaside Buffer (TLB) caches translations between virtual and physical page numbers. Without caching translations, each memory operation would incur the overhead of a multi-level page-table access. For some processor architectures, however, you must occasionally flush (invalidate) all of the TLB entries, even though this will temporarily slow down the processor. Explain when and why we might need to do this:

We may have to flush out the TLB when we switch from one process to another in order to avoid incorrect translation of virtual addresses to physical addresses. As a simple example, address "0" leads to a different translation from process to process. Thus, when we switch we would need to make sure to flush out any TLB entry that might map "0" to the old physical page for the old process. The same is true for all other addresses.

Note: for processors that do not have a process ID in each TLB entry, we would have to always flush the TLB when we switched processes. For processors that have a process ID in each TLB entry, we would potentially have to flush the TLB when we reused the process ID (since it typically has a small and finite number of bits). Consequently, the above answer is correct in both cases.

Problem 6d[2pts]: Consider a multi-level memory management scheme using the following format for *virtual addresses*, including 2 bits worth of segment ID and an 8-bit virtual page number:

Virtual seg # (2 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Virtual addresses are translated into 16-bit *physical addresses* of the following form:

Physical Page # (8 bits)	Offset (8 bits)

Page table entries (PTE) are 16 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory):

Physical Page # (8 bits)	Kernel	Nocache	0	0	Dirty	Use	Writeable	Valid

How big is a page in the above scheme? Explain.

Since the offset is 8 bits, a page is 2^8=256 bytes

Problem 6e[2pts]: In the above scheme, the top two bits of the address are used to select a segment. If we were using an x86 processor, instead, where would the segment identifier come from?

The segment identifier comes from instruction opcode bits rather than the address.

Problem 6f[10pts]: Using the scheme from (6d) and the Segment Table and Physical Memory table on the next page, state what will happen with the following loads and stores. Addresses below are virtual, while base addresses in the segment table are physical. If you can translate the address, make sure to place it in the "Physical Address" column; otherwise state "**N/A**".

The return value for a load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. If there is an error, say which error. Possibilities are: "**bad segment**" (invalid segment), "**segment overflow**" (address outside segment), or "**access violation**" (page invalid/attempt to write a read only page). A few answers are given:

Instruction	Translated Physical Address	Result (return value)
Load [0x30115]	**0x3115**	**0x57**
Store [0x10345]	**0x3145**	**Access violation**
Store [0x30557]	***0x3557***	***ok***
Test&Set [0x11213]	***N/A***	***segment overflow***
Load [0x31202]	***0xE002***	***0xAA***
Store [0x31231]	***0xE031***	***access violation***
Load [0x01315]	***0xF015***	***0x66***

Segment Table (Segment limit = 3)

Seg #	Page Table Base	Max Pages in Segment	Segment State
0	0x2030	0x20	Valid
1	0x1020	0x10	Valid
2	0xF040	0x40	Invalid
3	0x4000	0x20	Valid

Physical Memory

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
0x0010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
0x1010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x1020	40	03	41	01	30	01	31	01	00	03	00	00	00	00	00	00
0x1030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x1040	10	01	11	03	31	03	13	00	14	01	15	03	16	01	17	00
….																
0x2030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
0x2040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
0x2050	10	00	31	01	F0	03	F0	01	12	03	30	00	10	00	10	01
….																
0x3100	01	12	23	34	45	56	67	78	89	9A	AB	BC	CD	DE	EF	00
0x3110	02	13	24	35	46	57	68	79	8A	9B	AC	BD	CE	DF	F0	01
0x3120	03	01	25	36	47	58	69	7A	8B	9C	AD	BE	CF	E0	F1	02
0x3130	04	15	26	37	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
….																
0x4000	30	00	31	01	11	01	F0	03	34	01	35	07	43	38	32	79
0x4010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
0x4020	23	03	20	03	E0	01	E1	08	E2	86	28	03	48	25	34	21
….																
0xE000	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55
0xE010	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A	A5	5A
….																
0xF000	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0xF010	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
0xF020	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	11
….																

[This Page Intentionally Left Blank]

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]