

# Discussion 4: Scheduling

March 6, 2026

## Contents

<b>1 Scheduling</b>	<b>2</b>
1.1 Round Robin T/F . . . . .	3
1.2 Life Ain't Fair . . . . .	4
1.3 Bitcoin Mining . . . . .	6
<b>2 Starvation</b>	<b>8</b>
2.1 Concept Check . . . . .	10
2.2 Simple Priority Scheduler . . . . .	11
2.3 Banker's Algorithm . . . . .	12

# 1 Scheduling

**Scheduling** is the process of deciding which threads are given access to resources from moment to moment. Usually, scheduling pertains to CPU access times, but it can encompass any type of resource like network bandwidth or disk access.

When discussing scheduling, we typically make some assumptions for simplification. Each user is assumed to have one single-threaded program where each program is independent of each other.

## Goals and Criteria

The goal of scheduling is to dole out CPU time to optimize some desired parameters of the system. Generally speaking, scheduling policies focus on the following goals and criteria.

### Minimize Completion Time

**Completion time** is the combination of the waiting time plus the run time of a process (e.g. time to compile a program, time to echo a keystroke in an editor). Minimizing completion time is crucial for time-sensitive tasks (e.g. I/O).

### Maximize Throughput

**Throughput** is the rate at which tasks are completed. While throughput is related to completion time, they are not the same. Maximizing throughput involves minimizing overhead (e.g. context switching) and efficient use of resources.

### Maintain Fairness

**Fairness** refers to sharing resources in some equitable manner. Evidently, fairness is not very well defined unless specific parameters are specified. Maintaining fairness will usually contradict minimizing completion time.

### First Come First Serve (FCFS)

**First come first serve (FCFS)** schedules tasks in the order in which they arrive. It's simple to implement and is good for throughput since it minimizes the overhead of context switching. However, average completion time under FCFS can vary significantly according to arrival order. Additionally, FCFS suffers from the **Convoy effect** where short tasks get stuck behind long tasks.

### Shortest Job First (SJF) / Shortest Remaining Time First (SRTF)

**Shortest job first (SJF)** schedules the shortest task first. **Shortest remaining time first (SRTF)** is a preemptive version of SJF. If a task arrives and has a shorter time to completion than the current running task, the resource will be preempted. SJF and SRTF are provably optimal for minimizing average completion time among non-preemptive and preemptive schedulers, respectively. However, both of them involve the impossible idea of knowing how long a task is going to take.

Moreover, they do not maintain fairness with regards to the amount of time spent using a resource. In fact, SJF and SRTF can lead to **starvation** which is the continued lack of progress for one task due to other tasks being scheduled over it. If small tasks keep arriving, larger tasks will never get to run.

### Round Robin (RR)

**Round robin (RR)** schedules tasks such that each of them takes turns using a resource for a small unit of time known as the **time quantum** ( $q$ ). After  $q$  expires, the task is preempted and added to the end of the ready queue. When  $q$  is large, RR resembles FCFS, becoming identical if  $q$  is larger than the length of

any task. When  $q$  is small, RR resembles SJF. Generally,  $q$  must be large with respect to the cost of context switching to avoid the overhead being too high.

If there are  $n$  tasks, each task gets  $1/n$  amount of resources, ensuring fairness in terms of sharing resources. No task will wait for more than  $(n - 1)q$  time units.

In general, with RR, small scheduling quantum decreases average waiting time but increases completion time, due to the extra context switching overhead and the fact that longer jobs get "stretched out".

## Lottery

**Lottery** gives each task a certain number of lottery tickets. On each time slice, a ticket is randomly picked. On expectation, each task will be given time proportional to the number tickets it was originally given. When distributing tickets, it's important to make sure that every task gets at least one ticket to avoid starvation. If SRTF was to be approximated, shorter tasks would simply get more tickets than longer tasks.

## Multi-Level Feedback Queue (MLFQ)

**Multi-level feedback queue (MLFQ)** uses multiple queues which each have different priority. Each queue has its own scheduling algorithm. Higher-priority queues (foreground) will typically use RR while lower-priority queues (background) might use FCFS.

A task will start at the highest-priority queue. If the task uses up all the resources (e.g.  $q$  for RR), it will get pushed one level down as a penalty. If the task does not use up all the resources (e.g. less than  $q$  for RR), it will get pushed one level up as a reward. This ensures that long running tasks (e.g. CPU bound) don't hog all resources and get demoted into low priority queues while short running tasks (e.g. I/O bound) will remain at the higher-priority queues.

### 1.1 Round Robin T/F

1. The average wait time is less than that of FCFS for the same workload.

2. If a quantum is constantly updated to become the number of cpu ticks since boot, Round Robin becomes FCFS.

3. Ideally, you should set the time quanta of round robin generally to be significantly smaller than the average CPU burst time of a task.

4. Cache performance is likely to improve relative to FCFS.

5. If no new threads are entering the system, all threads will get a chance to run in the cpu every  $QUANTA * SECONDS\_PER\_TICK * NUMTHREADS$  seconds, assuming  $QUANTA$  is in ticks.

## 1.2 Life Ain't Fair

Suppose the following threads (**priorities given in parentheses**) arrive in the ready queue at the clock ticks shown. Assume all threads arrive unblocked and that **each takes 5 clock ticks to finish executing**. Assume threads arrive in the queue at the beginning of the time slices shown and are ready to be scheduled in that same clock tick. This means you update the ready queue with the arrival before you schedule/execute that clock tick. Assume you only have one physical CPU.

```

0 Taj (prio = 7)
1
2 Kevin (prio = 1)
3 Neil (prio = 3)
4
5 Akshat (prio = 5)
6
7 William (prio = 11)
8
9 Alina (prio = 14)

```

Determine the order and time allocations of execution for each given scheduler scenario. Write answers in the form of vertical columns with one name per row, each denoting one clock tick of execution. For example, allowing Taj 3 units at first looks like:

```

0 Taj
1 Taj
2 Taj

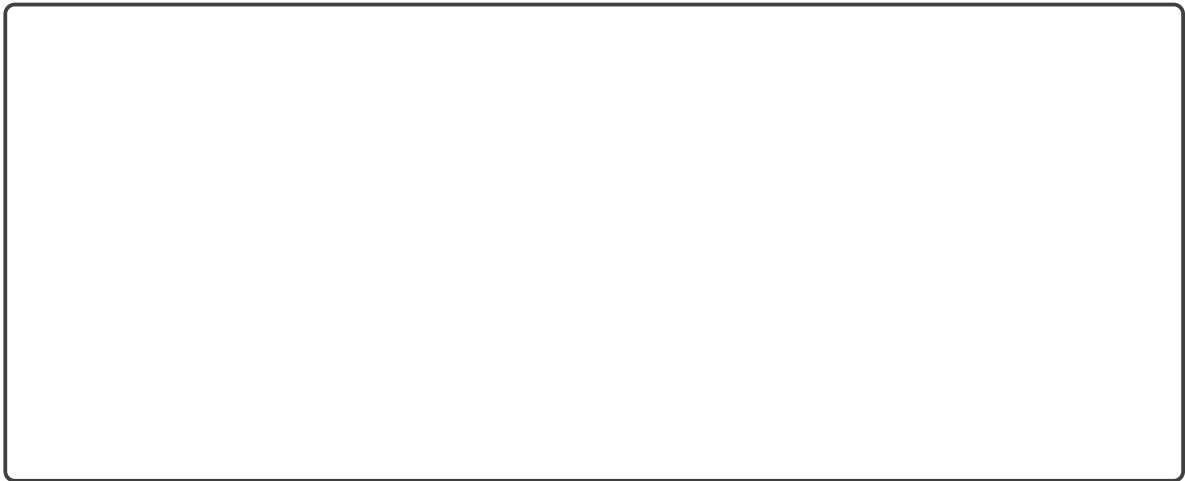
```

It will probably help you to draw a diagram of the ready queue at each tick for this problem.

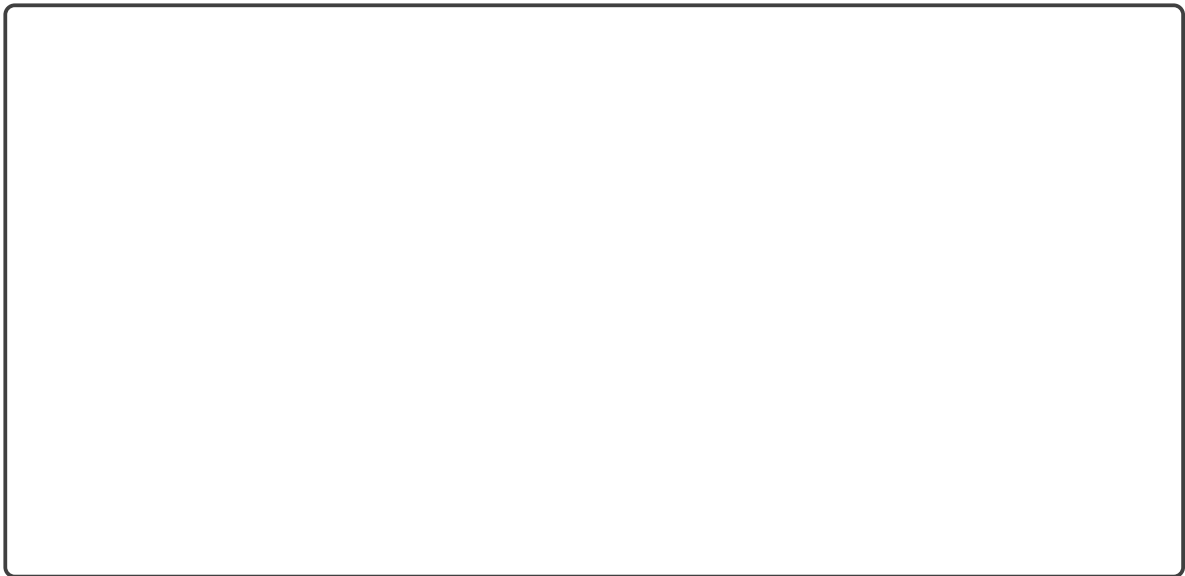
1. Round robin with time quantum 3



2. SRTF with preemptions

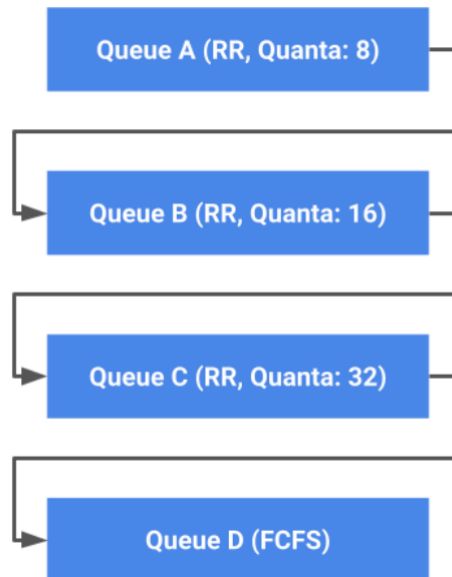


3. Preemptive priority



### 1.3 Bitcoin Mining

You are a Bitcoin miner, and you've developed an algorithm that can run on an unsuspecting machine and mine Bitcoin. You now need to write a program that will run your mining algorithm forever. While you want your mining job to be scheduled often, you also don't want to attract too much suspicion from system users or administrators. Fortunately, you know that the machines you're targeting use a MLFQS algorithm to schedule jobs, outlined below.



1. You decide that the best strategy is to guarantee that your mining job will always be placed on Queues B and C.

Assume that the CPU-intensive mining algorithm you've developed can be run in 10-tick intervals. Implement your mining program, and explain your design. The only functions you should use are `mine` (which runs for 10 ticks) and `printf`. Assume that your job is initially placed on Queue B.

```

1 void mine_forever() {
2   while(1) {
3     -----
4     -----
5     -----
6   }
7 }
  
```

bitcoin\_mine.c

2. Explain why, regardless of how you implement your mining program, your job will never be placed on Queue A twice in a row.

## 2 Starvation

When designing scheduling policies, it's important to prevent **starvation**, a situation where a thread fails to make progress for an indefinite period of time. If a scheduling policy never runs a particular thread on the CPU, the thread will starve. Threads can also starve if they wait for each other or are spinning in a way that will never be resolved.

### Strict Priority

A **strict priority scheduler** schedules the task with the highest priority. If multiple tasks have the same priority, they can be scheduled in some RR fashion. This ensures that important tasks get to run first but doesn't maintain fairness. Evidently, a strict priority scheduler suffers from starvation for lower priority threads.

### Priority Inversion

With a priority scheduler (e.g. strict priority), **priority inversion** can become a problem where a higher priority task is blocked waiting on a lower priority task. As a result, a medium priority task (in between the higher and lower priority) will be run by the scheduler, resulting in the medium priority task starving the higher priority task. A common example of this is when a higher priority task tries to acquire a mutex that the lower priority task already holds.

To address priority inversion, the scheduler can use **priority donation** where the lower priority task is *temporarily* granted the same priority as the higher priority task, so it can run. In this scenario, the lower priority thread is said to have an **effective priority** of the higher priority task. Once the higher priority task is no longer blocked on the lower priority task, the scheduler would demote it back to its **base priority**.

### Lottery

A **lottery scheduler** gives each task some number of lottery tickets. At each time slice, a random ticket is drawn, and the task holding that ticket is granted the resource. On expectation, the time each task is allocated the resource for will be proportional to the number of tickets it holds.

Ticket numbers can be assigned in a variety of schemes. For instance, the scheduler could approximate SRTF by giving more tickets to shorter jobs and less tickets to longer jobs. Essentially, the number of tickets serve as a measure of priority in some sense. To avoid starvation, it's important to make sure that every job gets at least one ticket.

### Stride

A **stride scheduler** is a deterministic version of a lottery scheduler. Like a lottery scheduler, the stride scheduler gives each task some number of tickets. Then, each task is defined a **stride** which is inversely proportional to the number of tickets. Typically, this is calculated as  $W/n_i$  where  $W$  is a really big number and  $n_i$  is the number of tickets given to task  $i$ .

On every time slice, the task with the lowest **pass** is chosen. All tasks start with **pass** equal to 0. When a task is chosen, its pass is incremented by its stride. Hence, a smaller stride will allow a task to run more often.

### Linux Completely Fair Scheduler (CFS)

The **Linux Completely Fair Scheduler (CFS)** aims to give each task an equal share of the CPU by giving an illusion that each task executes simultaneously on  $1/n$  of the CPU. Since hardware needs to give out CPU in full time slices, the scheduler will track CPU time per task and schedule tasks to match up the average rate of execution. When choosing a task to run, the thread with minimum CPU time will be chosen. To accomplish this efficiently, a heap like scheduling queue is used for efficient (logarithmic with respect to number of tasks) popping and pushing operations.

## Deadlock

**Deadlock** is a situation where there is a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action. Deadlock is a special form of starvation but has a stronger condition since starvation can end but deadlock cannot.

There are four necessary *but not sufficient* conditions for a deadlock to occur. Eliminating any one of these will eliminate the possibility of a deadlock.

### Mutual Exclusion and Bounded Resources

Finite number of threads (usually one) can simultaneously use a resource.

### Hold and Wait

Thread holds one resource while waiting to acquire additional resources held by other threads.

### No Preemption

Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.

### Circular Waiting

There exists a set of waiting threads such that each thread is waiting for a resource held by another.

## Detection

The following is a deadlock detection algorithm.

### Require:

available, array of how much of each resource is available

alloc, 2D array where the  $i$ -th element is how many of each resource thread  $i$  currently holds.

request, 2D array where the  $i$ -th element is the how many of each of resource thread  $i$  is requesting.

unfinished  $\leftarrow$  all threads

done  $\leftarrow$  false

**while** done is false **do**

    done  $\leftarrow$  true

**for** thread in unfinished **do**

**if** request[thread]  $\leq$  available **then**

            remove thread from unfinished

            available  $\leftarrow$  available + alloc[thread]

            done  $\leftarrow$  false

**end if**

**end for**

**end while**

When the algorithm finishes, the system is said to be deadlocked if there are threads left in unfinished.

## Handling

There are four main ways a system can handle a deadlock.

### Denial

Pretend deadlock is not a problem (i.e. ostrich algorithm). This is usually effective when deadlocks are uncommon. In the rare times when deadlocks do occur, the system is usually restarted.

### Prevention

Write systems that don't result in deadlock. This typically involves eliminating one of the four necessary conditions. Infinite resources could eliminate the bounded resources problem. While not possible for all types of resources, an illusion of infinite resources typically suffices (e.g. virtual memory). No sharing of resources can eliminate mutual exclusion, but totally independent threads are not very realistic and defeats the purpose of using threads. Forcing all threads to request resources in a particular order can also be useful and typically done in many scenarios, but it does require careful coding practices.

Threads could also be forced to not wait and instead keep trying until a successful acquisition of a resource, which is quite inefficient.

### Recovery

Let deadlock happen, and recover from it afterwards. A simple and intuitive approach may be to terminate a thread, forcing it to give up its resources. However, this isn't always possible since killing a thread holding a mutex leaves the system in an inconsistent state. The system could also try to take away resources from the thread temporarily, but that may be difficult to fit the semantics of computation. A more general technique is to roll back actions of deadlocked threads. However, this is also prone to deadlock in the same way again if threads are executed in the same order.

### Avoidance

Dynamically delay resource requests, so deadlock doesn't happen. The first idea that comes to mind is to check if a resource request would result in a deadlock when a thread requests a resource. However, this may be too late as the system may already be in a **unsafe state** where there isn't a deadlock yet but there is potential for a pattern of resource requests that unavoidably leads to deadlock. The system must always be kept in a **safe state** where the system can delay resource acquisition requests to prevent deadlock. As a result, the system needs to check if a resource request would result in an *unsafe* state not a deadlocked state.

**Banker's algorithm** is a deadlock avoidance technique. A thread states the max amount of each resource it needs in advance. The conservative approach would be to allow a particular thread to proceed if the available resources - number requested is at least the max number needed by any other thread. Banker's algorithm takes a less conservative route and pretends each request is granted. Then, it runs the deadlock detection algorithm with an updated condition as seen below.

#### Require:

available, array of how much of each resource is available

alloc, 2D array where the  $i$ -th element is how many of each resource thread  $i$  currently holds.

max, 2D array where the  $i$ -th element is the max number of resources thread  $i$  will request.

unfinished  $\leftarrow$  all threads

done  $\leftarrow$  false

**while** done is false **do**

    done  $\leftarrow$  true

**for** thread in unfinished **do**

**if** max[thread] - alloc[thread]  $\leq$  available **then**

            remove thread from unfinished

            available  $\leftarrow$  available + alloc[thread]

            done  $\leftarrow$  false

**end if**

**end for**

**end while**

The key idea is that banker's algorithm determines if threads are able to be scheduled in a way to avoid deadlock. However, not every execution order has to avoid deadlock.

## 2.1 Concept Check

1. In what sense is Linux CFS completely fair?

2. How can you easily implement lottery scheduling?

3. Is stride scheduling prone to starvation?

4. When using stride scheduling, if a task is more urgent, should it be assigned a larger stride or a smaller stride?

## 2.2 Simple Priority Scheduler

Let's implement a new scheduler in Pintos called the simple priority scheduler (SPS). We will just split threads into two priorities: high (1) and low (0). High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

```

1 struct thread {
2     ...
3     int priority;
4     struct list_elem elem;
5     ...
6 }
7
8 struct list ready_list;
9
10 void thread_unblock (struct thread *t) {
11     ASSERT(is_thread(t));
12
13     enum intr_level old_level;
14     old_level = intr_disable();
15     ASSERT(t->status == THREAD_BLOCKED);
16
17     if (_____) {
18         _____;
19     } else {
20         _____;
21     }
22
23     t->status = THREAD_READY;
24     intr_set_level(old_level);
25 }
```

pintos\_sps.c

1. Complete the blanks of `thread_unblock` to complete implement SPS. Assume that SPS will treat the ready queue as FIFO. You may ignore any possibilities of preemptions.

2. In order for this scheduler to be "fair" briefly describe when you would make a thread high priority and when you would make a thread low priority.

3. If we let the user set the priorities of this scheduler with `set_priority`, why might this scheduler be preferable to the normal pintos priority scheduler?

4. How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? Assume we still want a fast insert.

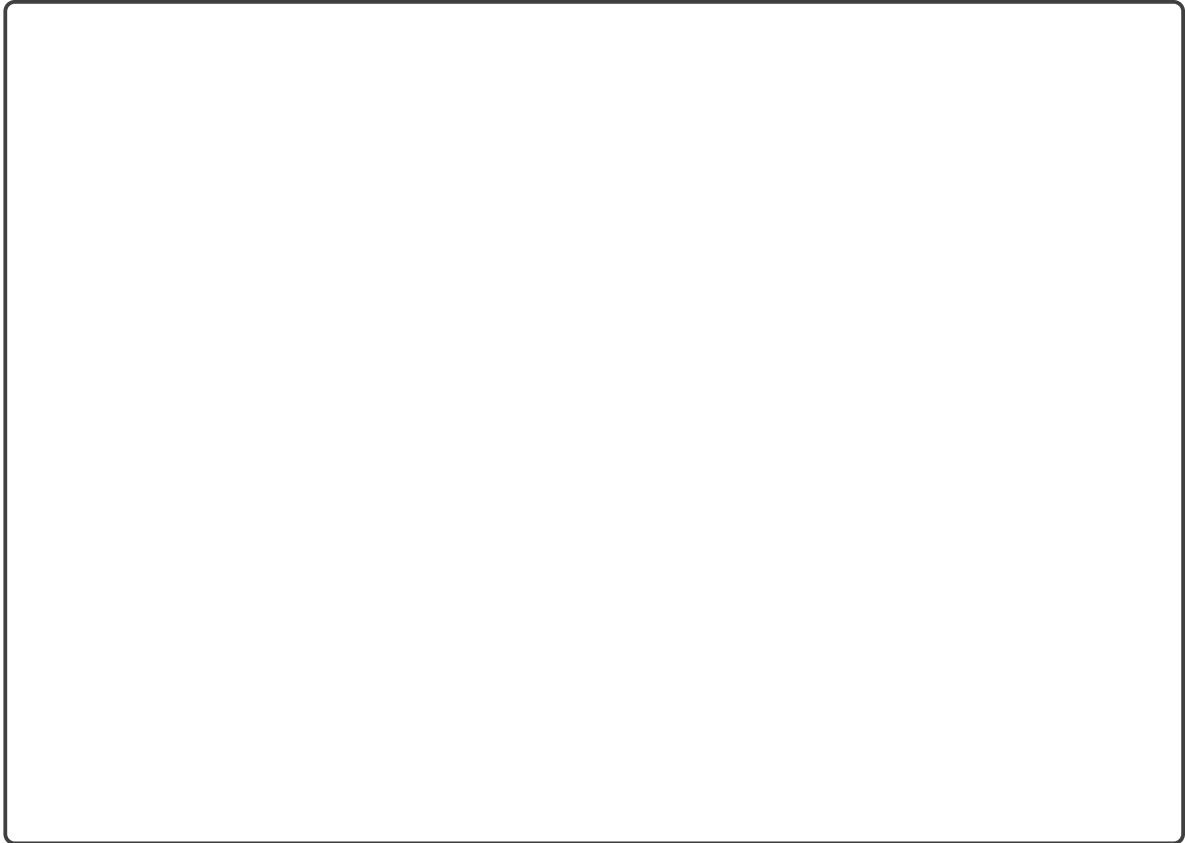
### 2.3 Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total		
A	B	C
7	8	9

T/R	Current			Max		
	A	B	C	A	B	C
T1	0	2	2	4	3	3
T2	2	2	1	3	6	9
T3	3	0	4	3	1	5
T4	1	3	1	3	3	4

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.



2. If the total number of C instances is 8 instead of 9, is the system still in a safe state?



### 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)
<pre>TSET(&amp;addr) {   int result = M[addr];   M[addr] = 1;   return (result); }</pre>	<pre>SWAP(&amp;addr, expr) {   int result = M[addr];   M[addr] = expr;   return (result); }</pre>	<pre>CAS(&amp;addr, expr1, expr2) {   if (M[addr] == expr1) {     M[addr] = expr2;     return true;   } else {     return false;   } }</pre>

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.*

**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, _____, _____ ));
}
```

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, _____, _____);
    }
}
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->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 enqueueing 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 ourselves 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:**

**Point 2:**

**Point 3:**

**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:   _____
C:   _____
D:   _____

```

**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:

```
// Thread-safe version of original lines 9-13
```

E: \_\_\_\_\_

F: \_\_\_\_\_

G: \_\_\_\_\_

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:** \_\_\_\_\_

What should you replace it with:

**I:** \_\_\_\_\_

```

// 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:** \_\_\_\_\_

**K:** \_\_\_\_\_

**L:** \_\_\_\_\_

**M:** \_\_\_\_\_

**N:** \_\_\_\_\_

```

19:   } else {
20:     myLock->lockvar = UNLOCKED;
21:   }
}
```

