

CS111, Lecture 19

Preemption and Implementing Locks



masks recommended

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?

CS111 Topic 3: Multithreading, Part 2

Scheduling and
Dispatching

Lecture 17



Scheduling and
Preemption,
Continued

Lecture 18



Preemption and
Implementing
Locks

This Lecture

assign5: implement your own version of **thread**, **mutex** and **condition_variable**!

Learning Goals

- Compare tradeoffs between various approaches to scheduling
- Learn about the assign5 infrastructure and how to implement a dispatcher with *preemption*
- See how our understanding of thread dispatching/scheduling allows us to implement locks

Plan For Today

- **Recap and continuing:** Scheduling
- Preemption and Interrupts
- Implementing Locks

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

Plan For Today

- **Recap and continuing: Scheduling**
- Preemption and Interrupts
- Implementing Locks

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

Scheduling

Key Question: How does the operating system decide which thread to run next? (e.g. many **ready** threads). Assume just 1 core.

We discussed 2 main designs so far:

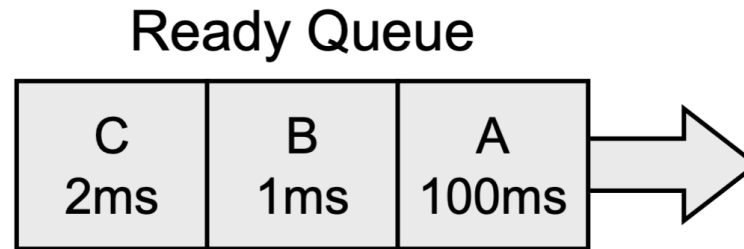
- 1. First-come-first-serve (FIFO / FCFS):** keep threads in ready queue, add threads to the back, run thread from front until completion or blocking.
- 2. Round Robin:** run thread for one time slice, then add to back of queue if wants more time

Scheduling Algorithms

How do we decide whether a scheduling algorithm is good?

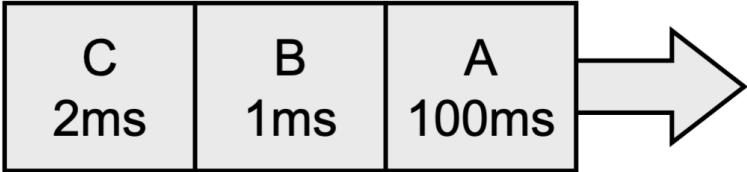
- Minimize response time (time to useful result)
 - e.g. keystroke -> key appearing, or “make” -> program compiled
 - Assume useful result is when the thread blocks or completes
- Use resources efficiently
 - keep cores + disks busy
 - low overhead (minimize context switches)
- Fairness (e.g. with many users, or even many jobs for one user)

Comparing FCFS/RR: Scenario 1



Comparing FCFS/RR: Scenario 1

Ready Queue



FIFO



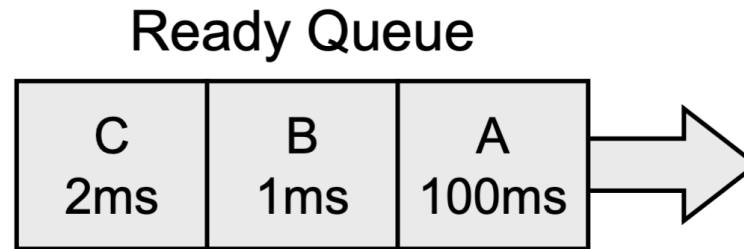
time

100 101 103

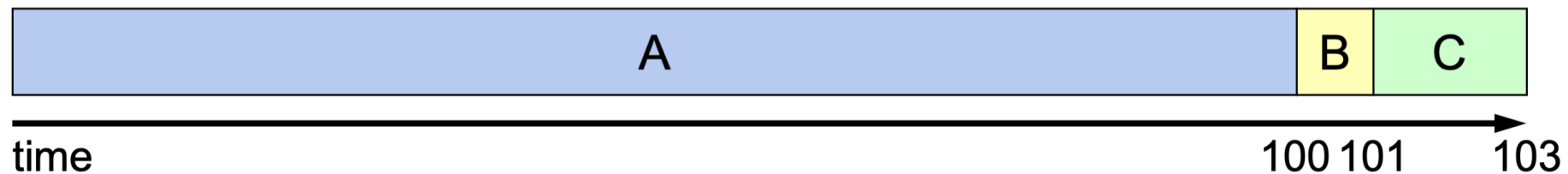
Avg:
101.3

Comparing FCFS/RR: Scenario 1

Is RR *always* better than FCFS?

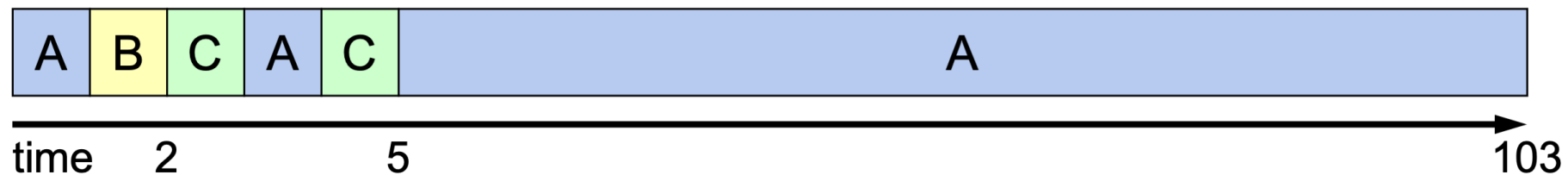


FIFO



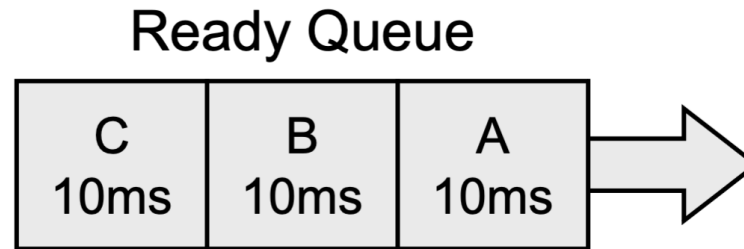
Avg:
101.3

Round Robin



Avg:
36.7

Comparing FCFS/RR: Scenario 2

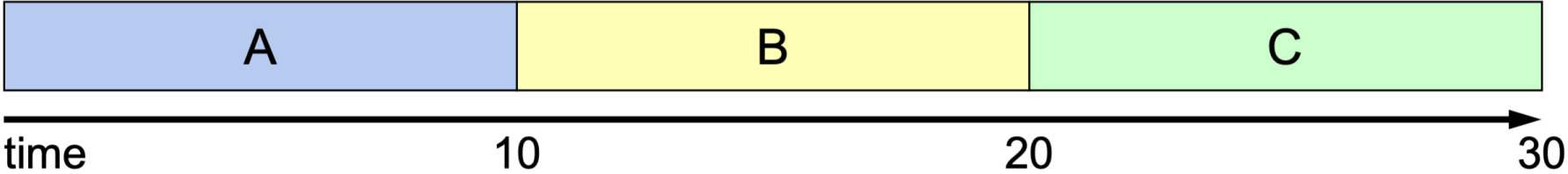


Comparing FCFS/RR: Scenario 2

Ready Queue

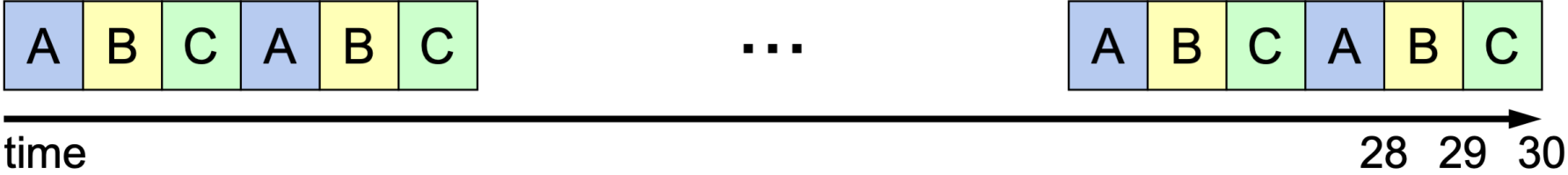


FIFO



Avg:
20

Round Robin



Avg:
29

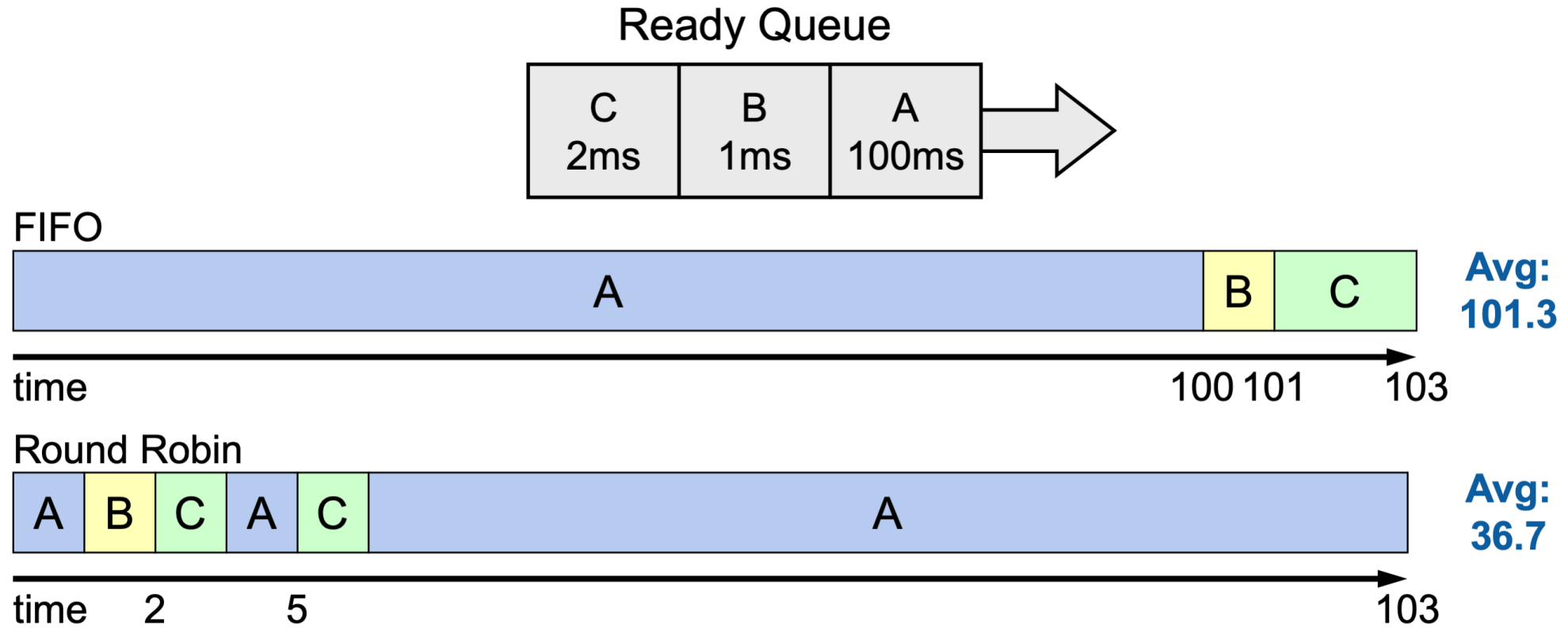
What's the optimal approach if we want to minimize average response time?

Shortest Remaining Processing Time

What would it look like if we optimized for completion time? (time to finish, or time to block).

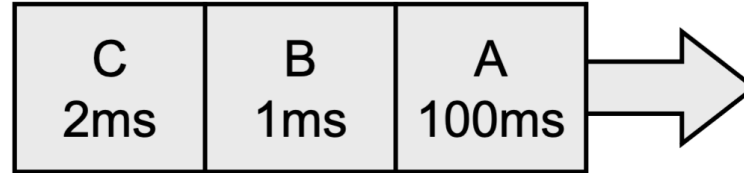
Idea - SRPT: pick the thread that will finish the most quickly and run it to completion. This is the optimal solution for minimizing average response time.

Evaluating SRPT

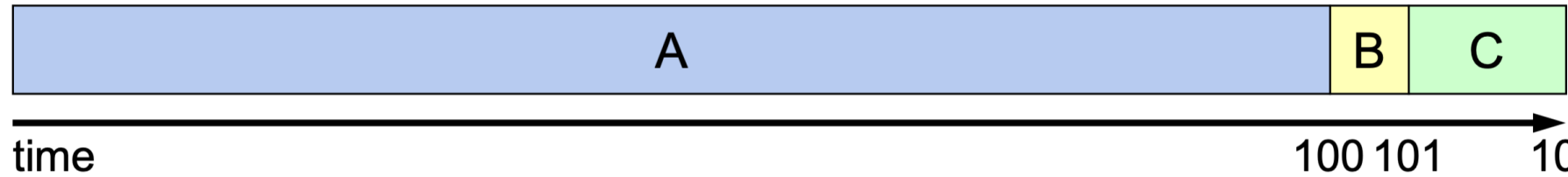


Evaluating SRPT

Ready Queue

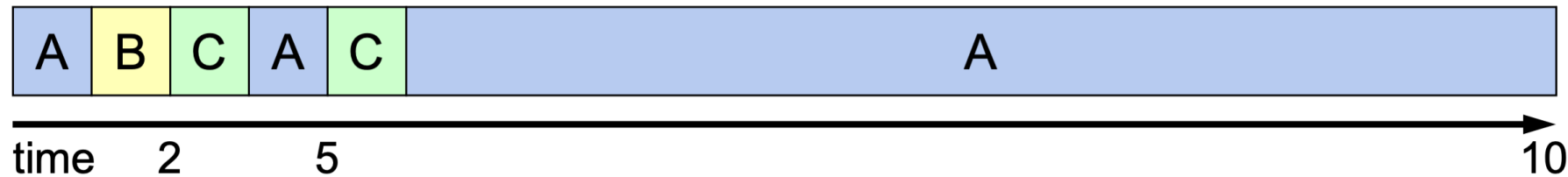


FIFO



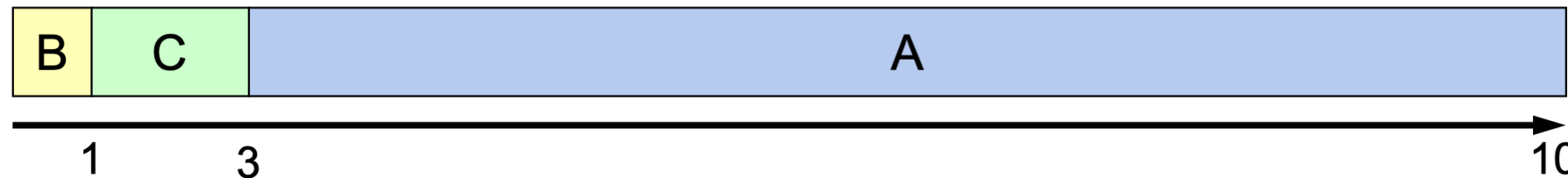
Avg:
101.3

Round Robin



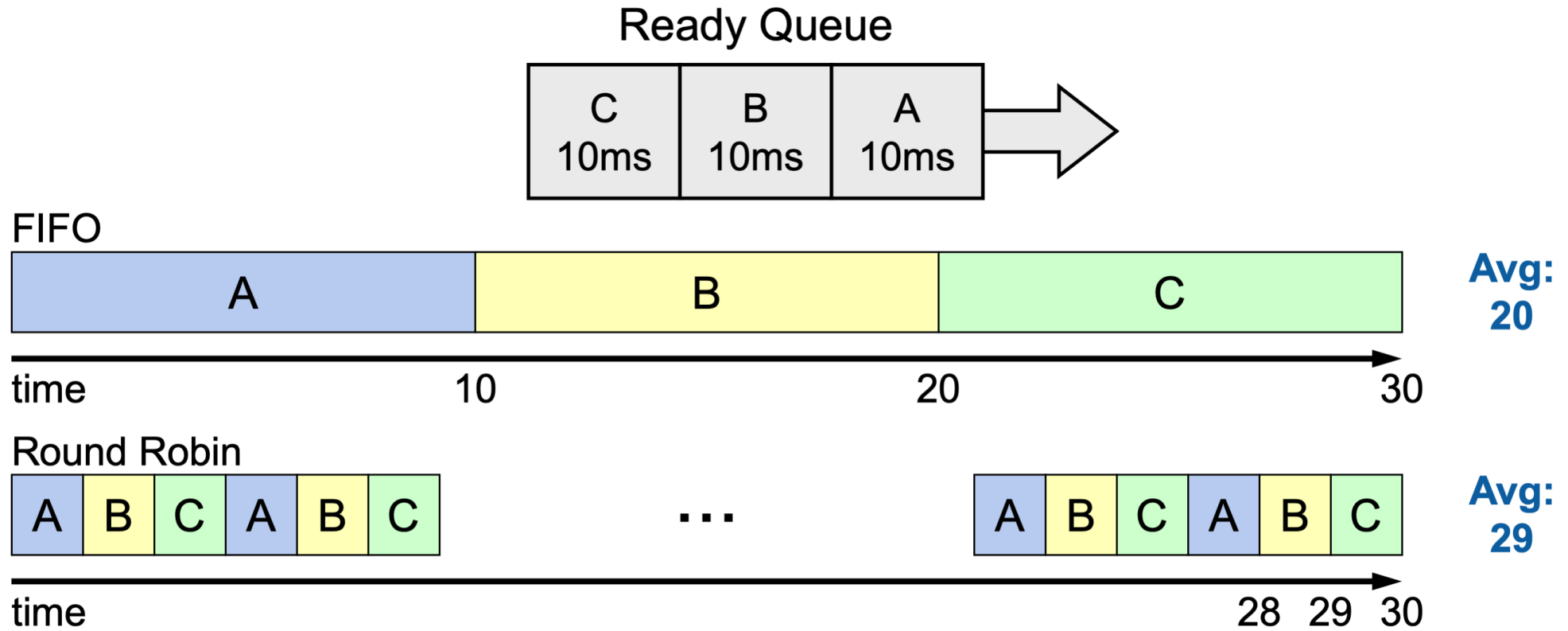
Avg:
36.7

SRPT



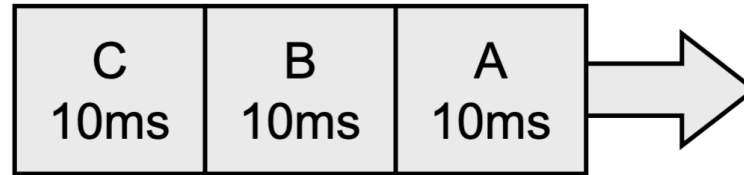
Avg:
35.7

Evaluating SRPT

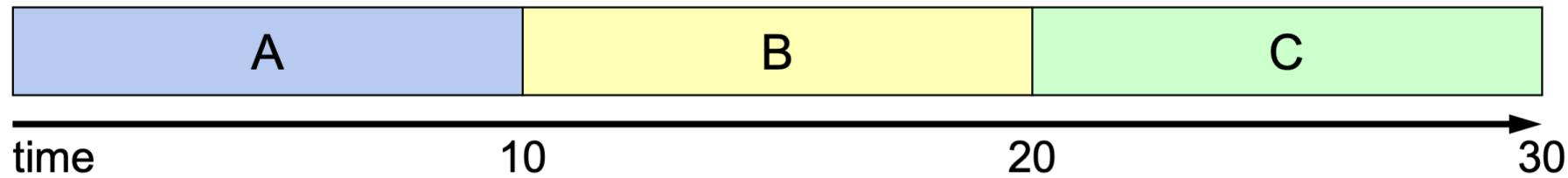


Evaluating SRPT

Ready Queue

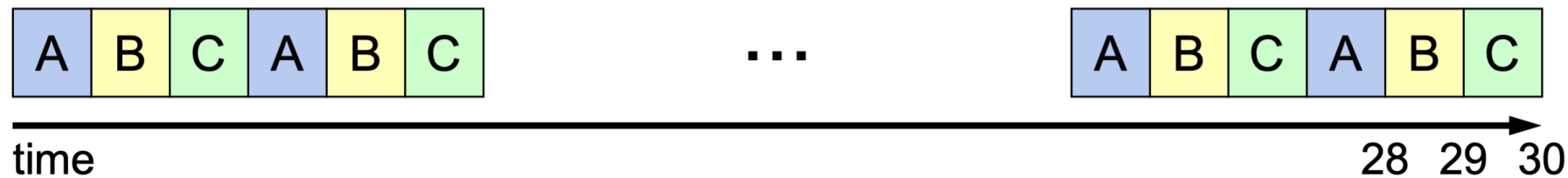


FIFO



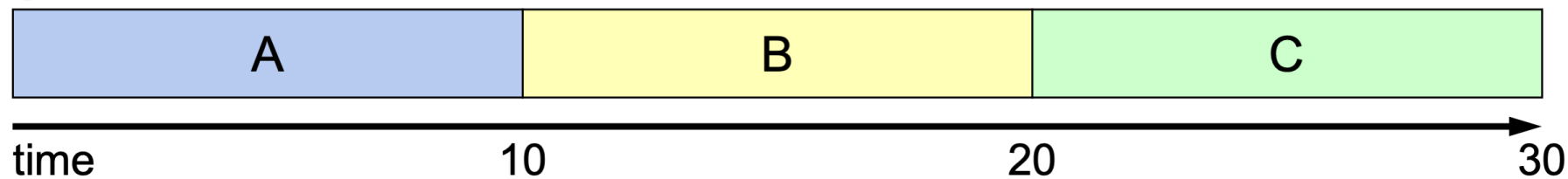
Avg:
20

Round Robin



Avg:
29

SRPT



Avg:
20

Shortest Remaining Processing Time

SRPT: pick the thread that will finish the most quickly and run it to completion. This is the optimal solution for minimizing average response time.

What are some problems/challenges with the SRPT approach?

Problem #1: how do we know which one will finish most quickly? (we must be able to predict the future...)

Problem #2: if we have many short-running threads and one long-running one, the long one will not get to run (“starvation”)

How can we get close to SRPT but without having to predict the future or neglect certain threads?

Priority-Based Scheduling

Goal: we want to get close to SRPT, but without having to predict the future, and without neglecting certain threads.

Key Idea: can use past performance to predict future performance.

- Behavior tends to be consistent
- If a thread runs for a long time without blocking, it's likely to continue running

Priority-Based Scheduling

Goal: we want to get close to SRPT, but without having to predict the future, and without neglecting certain threads.

Idea: let's make threads have priorities that adjust over time as they run. We'll have 1 ready queue for each priority, and always run highest-priority threads.

- Overall idea: threads that aren't using much CPU time stay in the higher-priority queues, threads that are migrate to lower-priority queues.
- After blocking, thread starts in highest priority queue
- If a thread reaches the end of its time slice without blocking it moves to the next lower queue.

Problem: could still neglect long-running threads!

Priority-Based Scheduling

Idea: let's make threads have priorities that adjust over time as they run. We'll have 1 ready queue for each priority, and always run highest-priority threads.

Problem: could still neglect long-running threads!

Let's keep track of *recent CPU usage per thread*. If a thread hasn't run in a long time, its priority goes up. And if it has run a lot recently, priority goes down.
(4.4 BSD Unix used this, ideas carried forward)

- No more neglecting threads: a thread that hasn't run in a long time will get its priority increased
- If there are many equally-long threads that want to run, the priorities even out over time, at a kind of "equilibrium"

Scheduling

Key Question: How does the operating system decide which thread to run next? (e.g. many **ready** threads). Assume just 1 core.

We discussed 4 main designs:

- 1. First-come-first-serve (FIFO / FCFS):** keep threads in ready queue, add threads to the back, run thread from front until completion or blocking.
- 2. Round Robin:** run thread for one time slice, then add to back of queue if wants more time
- 3. Shortest Remaining Processing Time (SRPT):** pick the thread that will complete or block the soonest and run it to completion.
- 4. Priority-Based Scheduling:** threads have priorities, and we have one ready queue per priority. Threads adjust priorities based on time slice usage, or based on recent CPU usage (4.4 BSD Unix)

Plan For Today

- Recap and continuing: Scheduling
- **Preemption and Interrupts**
- Implementing Locks

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

Preemption and Interrupts

On assign5, you'll implement a **dispatcher and scheduler** using the Round Robin approach.

- *Preemptive*: threads can be kicked off in favor of others (after time slice)

To implement this, we've provided a **timer** implementation that lets you run code every X microseconds.

- Fires a timer interrupt at specified interval

Idea: we can use the timer handler to trigger a context switch!

(For simplicity, on assign5 we'll always do a context switch when the timer fires)

Timer Demo

```
// this program runs timer_interrupt_handler every 0.5 seconds
```

```
void timer_interrupt_handler() {  
    cout << "Timer interrupt occurred!" << endl;  
}
```

```
int main(int argc, char *argv[]) {  
    // specify microsecond interval and function to call  
    timer_init(500000, timer_interrupt_handler);  
    while (true) {}  
}
```



interrupt.cc

**Demo: context-switch-
preemption-buggy.cc**

Interrupts

When the timer handler is called, it's called with (all) interrupts **disabled**. Why?
To avoid a timer handler interrupting a timer handler.

When the timer handler finishes, interrupts are **re-enabled**.

```
// within timer code
```

```
// (omitted) timer disables interrupts here
```

```
your_timer_handler();
```

```
// (omitted) timer re-enables interrupts here
```

Interrupt state is shared (not per-thread).

Interrupts

When the timer handler is called, it's called with (all) interrupts **disabled**. Why?
To avoid a timer handler interrupting a timer handler.

When the timer handler finishes, interrupts are **re-enabled**.

```
// within timer code
```

```
// (omitted) timer disables interrupts here
```

```
your_timer_handler();
```

```
// (omitted) timer re-enables interrupts here
```

Problem: because we context switch in the middle of the timer handler, when we start executing another thread **for the first time**, we will have interrupts **disabled** and the timer won't be heard anymore!

Enabling Interrupts

Solution: manually enable interrupts when a thread is first run.

```
void other_func() {  
    intr_enable(true);  
    while (true) {  
        cout << "Other thread here! Hello." << endl;  
    }  
}
```

On assign5: when a program creates a thread and gives you the function that thread should run, you will run that thread initially by **enabling interrupts first** and *then* running their specified function.

Disabling/Enabling Interrupts

The assignment starter code provides the following to enable or disable interrupts:

```
void intr_enable(bool on);
```


Interrupts

What about when we switch to a thread that we've already run before? Do we need to enable interrupts there too?

No – if a thread is paused that means when it was previously running, the timer handler was called and it context-switched to another thread. Therefore, when that thread resumes, **it will resume at the end of the timer handler**, where interrupts are re-enabled.

Interrupts

- On assign5, there are other places where interrupts can cause complications.
- E.g. we could be in the middle of adding to the ready queue, but then the timer fires and we go to remove something from the ready queue!
 - This sounds like a race condition problem we can solve with **mutexes**!....right?
 - **Not in this case** – because we are the OS, and we implement mutexes! And they rely on the thread dispatching code in this assignment.
 - Therefore, the mechanism for avoiding race conditions is to enable/disable interrupts when we don't want to be interrupted (e.g. by timer).
 - Interrupts are a shared state – not per-thread.
 - We're assuming a single-core machine, where disabling interrupts is sufficient to guarantee no other thread will run.

Plan For Today

- Recap and continuing: Scheduling
- Preemption and Interrupts
- **Implementing Locks**

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

Implementing Locks

Now that we understand how thread dispatching/scheduling works, we can write our own **mutex** implementation! Mutexes need to block threads (functionality the dispatcher / scheduler provides).

What does the design of a lock look like? What state does it need?

- Track whether it is locked / unlocked
- The lock “owner” (if any) – perhaps combine with first bullet
- A list of threads waiting to get this lock

Implementing Locks

Now that we understand how thread dispatching/scheduling works, we can write our own **mutex** implementation! Mutexes need to block threads (functionality the dispatcher / scheduler provides).

What does the design of a lock look like? What state does it need?

- Track whether it is locked / unlocked
- The lock “owner” (if any) – perhaps combine with first bullet
- A list of threads waiting to get this lock

We can keep a queue of threads (for fairness). (Hint: C++ has a built-in **queue** data structure)

Lock

1. If this lock is unlocked, mark it as locked by the current thread
2. Otherwise, add the current thread to the back of the waiting queue

```
// Instance variables
```

```
int locked = 0;
```

```
ThreadQueue q;
```

```
void Lock::lock() {
```

```
    if (!locked) {
```

```
        locked = 1;
```

```
    } else {
```

```
        q.add(currentThread);
```

```
        blockThread(); // block/switch to next ready thread
```

```
    }
```

```
}
```

Unlock

1. If no-one is waiting for this lock, mark it as unlocked
2. Otherwise, keep it locked, but unblock the next waiting thread

```
// Instance variables
```

```
int locked = 0;
```

```
ThreadQueue q;
```

```
void Lock::unlock() {
```

```
    if (q.empty()) {
```

```
        locked = 0;
```

```
    } else {
```

```
        unblockThread(q.remove()); // add to ready queue
```

```
    }
```

```
}
```

Mutex

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);

        // block/switch to next
        // ready thread
        blockThread();
    }
}
```

```
void Lock::unlock() {
    if (q.empty()) {
        locked = 0;
    } else {
        // add to ready queue
        unblockThread(q.remove());
    }
}
```

Can you think of an example race condition that could occur if we do not disable interrupts here and two threads lock a single mutex at the same time?

Mutex

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);

        // block/switch to next
        // ready thread
        blockThread();
    }
}
```

```
void Lock::unlock() {
    if (q.empty()) {
        locked = 0;
    } else {
        // add to ready queue
        unblockThread(q.remove());
    }
}
```

Can you think of an example race condition that could occur if we do not disable interrupts here and two threads lock a single mutex at the same time?

Example: thread 1 is in the middle of getting ownership, but then the timer fires, we switch to thread 2, and it locks the mutex. Then thread 1 resumes and *also* gets the mutex.

Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        intr_enable(true); // ??
        blockThread(); // block/swit
    }
}
```

Possible scenario (2 threads):

1. Thread #1 locks mutex
2. Thread #2 locks mutex, adds itself to the queue, enables interrupts
3. *Right before thread #2 blocks, thread #1 unlocks the mutex and unblocks thread #2*
4. Thread #2 then proceeds to block.
5. Nobody unblocks thread #2 😞

Lock

```
// Instance variables
```

```
int locked = 0;
```

```
ThreadQueue q;
```

```
void Lock::lock() {
```

```
    IntrGuard guard;
```

```
    if (!locked) {
```

```
        locked = 1;
```

```
    } else {
```

```
        q.add(currentThread);
```

```
        blockThread(); // block/switch to next ready thread
```

```
    }
```

```
}
```

Instead, we must re-enable interrupts at the end of **lock()**. This means that once a thread *unblocks* to acquire the lock, it wakes up after **blockThread()** and re-enables interrupts.

Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread(); // block/swit
    }
}
```

IntrGuard is like unique_lock but for interrupts. It saves the current interrupt state (enabled/disabled) when it's created and turns interrupts off. When it is deleted, it restores interrupts to the saved state.

Key idea: if interrupts are already disabled when an IntrGuard is created, it keeps them disabled.

Unlock

1. If no-one is waiting for this lock, mark it as unlocked
2. Otherwise, keep it locked, but unblock the next waiting thread

```
// Instance variables
```

```
int locked = 0;
```

```
ThreadQueue q;
```

```
void Lock::unlock() {
```

```
    IntrGuard guard;
```

```
    if (q.empty()) {
```

```
        locked = 0;
```

```
    } else {
```

```
        unblockThread(q.remove()); // add to ready queue
```

```
    }
```

```
}
```

Lock

```
// Instance variables
```

```
int locked = 0;
```

```
ThreadQueue q;
```

```
void Lock::lock() {
```

```
    IntrGuard guard;
```

```
    if (!locked) {
```

```
        locked = 1;
```

```
    } else {
```

```
        q.add(currentThread);
```

```
        blockThread(); // block/switch to next ready thread
```

```
    }
```

```
}
```

Problem: what happens when we switch to the next ready thread? Interrupts will be disabled!

Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread(); // block/switch
    }
}
```

Problem: what happens when we switch to the next ready thread? Interrupts will be disabled!

Key Idea: we know that every possible way a thread resumes (e.g. timer), it will re-enable interrupts. Therefore, this isn't a problem.



Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```

Thread #2

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```




Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::lock() {  
    → IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```

Thread #2

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::lock() {  
    IntrGuard guard;  
    → if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```

Thread #2

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        → locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    → IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    → if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```




Enabling/Disabling Interrupts


Interrupts
OFF

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```





Enabling/Disabling Interrupts


Interrupts
OFF

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```





Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

Thread #2 (blocked)

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

➔ (assume thread 1 reenables interrupts when resumed and disables them when paused)



Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::unlock() {  
    IntrGuard guard;  
    if (q.empty()) {  
        locked = 0;  
    } else {  
        unblockThread(q.remove());  
    }  
}
```

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2 (blocked)

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::unlock() {  
    → IntrGuard guard;  
    if (q.empty()) {  
        locked = 0;  
    } else {  
        unblockThread(q.remove());  
    }  
}
```

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2 (blocked)

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```



(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```



Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

➔ (assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```




Enabling/Disabling Interrupts

Interrupts
OFF

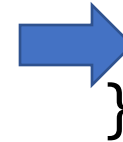
Thread #1

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

```
void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```





Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
void Lock::unlock() {  
    IntrGuard guard;  
    if (q.empty()) {  
        locked = 0;  
    } else {  
        unblockThread(q.remove());  
    }  
}
```

(assume thread 1 reenables
interrupts when resumed and
disables them when paused)

Thread #2

```
void Lock::lock() {  
    IntrGuard guard;  
    if (!locked) {  
        locked = 1;  
    } else {  
        q.add(currentThread);  
        blockThread();  
    }  
}
```



Plan For Today

- Recap and continuing: Scheduling
- Preemption and Interrupts
- Implementing Locks

Next time: Virtual Memory

Lecture 19 takeaway: To implement preemption and locks, we must make sure to correctly enable and disable interrupts. Locks consist of a waiting queue and redispaching to make threads sleep.