Kernel Locks in a Multiprocessor Universe

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

If we have a **single processor**, providing access to a critical section is “easy”:

Acquire:

```
Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        sch_move_to_blocked(myTCB);

        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

Release:

```
Lock::release() {
    disableInterrupts();
    if (!waiting.empty()) {
        nextTCB = waiting.remove();
        sch_move_to_ready(nextTCB);


    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

If we have a **single processor**, providing access to a critical section is “easy”:

Acquire:

```
Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        sch_move_to_blocked(myTCB);

        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

Release:

```
Lock::release() {
    disableInterrupts();
    if (!waiting.empty()) {
        nextTCB = waiting.remove();
        sch_move_to_ready(nextTCB);


    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

If we have a **mutli-processor**, disabling interrupts globally is impossible:

Acquire:

```
Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        sch_move_to_blocked(myTCB);

        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

★ Interrupts are handled on a **per-core basis**:
- Advanced Programmable Interrupt Controller (APIC) delegates interrupts to various cores.

★ **Performance would be terrible if we block all 64 cores** on a 64-core CPU for every lock operation!

Goals

- ★ **To ensure a globally shared variable** (ex: scheduler data) is accessed only by one core, we need some shared state.

- ★ **Q:** What is the absolute fastest way to achieve this?
  - **Need:** No impact on other cores (we might be one of 64 cores!)
  - **Need:** Safety to ensure no two cores can advance into the critical section at the same time.

test-and-set

★ **Idea:** Atomicly perform two operations with preemption:

1. **test** if a specific bit of shared memory is 0,
2. If and only if it was 0, **set** the bit to 1,

LOCK (x86): Assert LOCK# Signal Prefix

LOCK

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.
[...]
The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

BTS (x86): Bit Test and Set

BTS BitBase, BiPosition

CF = BitBase[BitOffset];

BitBase[BitOffset] ← 1;

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

Test-and-set (x86)

- ★ The BTS operation always does two things:
  1. Writes the shared memory into our core's Carry Flag (CF) register
  2. Modifies the shared memory to contain a 1.

Test-and-set (x86)

★ The BTS operation always does two things:

1. Writes the shared memory into our core's Carry Flag (CF) register
2. Modifies the shared memory to contain a 1.

⇒ After the LOCK-BTS operation:

- If CF==0, the value was 0 and we have acquired the lock!
- If CF==1, the value was 1 and someone else has the lock. We cannot advance.

Test-and-set (x86)

★ The BTS operation always does two things:

1. Writes the shared memory into our core's Carry Flag (CF) register
2. Modifies the shared memory to contain a 1.

⇒ After the LOCK-BTS operation:

- If CF==0, the value was 0 and we have acquired the lock!
- If CF==1, the value was 1 and someone else has the lock. We cannot advance.

We will abstract this away to: **`testAndSet(&sharedMemory)`**.

SpinLock:

```
spinLock(&lock) {

  while (testAndSet(&lock)) {
    yield();
  }

}
```

Acquire:

```
Lock::acquire() {
    disableInterrupts();
    spinLock(&lock);
    if (value == BUSY) {
        waiting.add(myTCB);
        sch_move_to_blocked(myTCB);
        /* only returns when RUNNING */
    } else {
        value = BUSY;
    }
    spinLockRelease(&lock);
    enableInterrupts();
}
```

SpinLock:

```
spinLock(&lock) {

  while (testAndSet(&lock)) {
    yield();
  }

}
```

Acquire:

```
Lock::acquire() {
    disableInterrupts();
    spinLock(&lock);
    if (value == BUSY) {
        waiting.add(myTCB);
        spinLockRelease(&lock);
        sch_move_to_blocked(myTCB);
        /* only returns when RUNNING */
        spinLock(&lock);
    } else {
        value = BUSY;
    }
    spinLockRelease(&lock);
    enableInterrupts();
}
```

SpinLock Solution

- ★ We minimize the need for multi-core coordination down to a single LOCK-BTS operation.

- ★ The busy-waiting occurs on only a single core/thread:
  - Busy-waiting only occurs when the lock is locked (should be rare)
  - We control the scheduler, further optimization possible with scheduler coordinator

Semaphores

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

What are semaphores?

In modern systems, semaphores (“counting semaphores”) are a synchronization mechanism where:

- **sem_wait()**: decrements (locks) the semaphore.
  - *If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks.*

- **sem_post()**: increments (unlocks) the semaphore.

Semaphores and State

★ This means that **semaphores are like "blocking integers"**, where we can only interact tem through sem_wait/sem_post.

★ This means semaphores have state!

- Much more complex to reason about. *(What does val==3 mean?)*
- Same functionality can be accomplished **explicitly** with conditional variables -- **CVs are much easier to reason about**.

Lock:

```
Lock::acquire() {
    disableInterrupts();
    spinLock(&lock);
    if (value == BUSY) {
        waiting.add(myTCB);
        spinLockRelease(&lock);
        sch_move_to_blocked(myTCB);
        /* only returns when RUNNING */
        spinLock(&lock);
    } else {
        value = BUSY;
    }
    spinLockRelease(&lock);
    enableInterrupts();
}
```

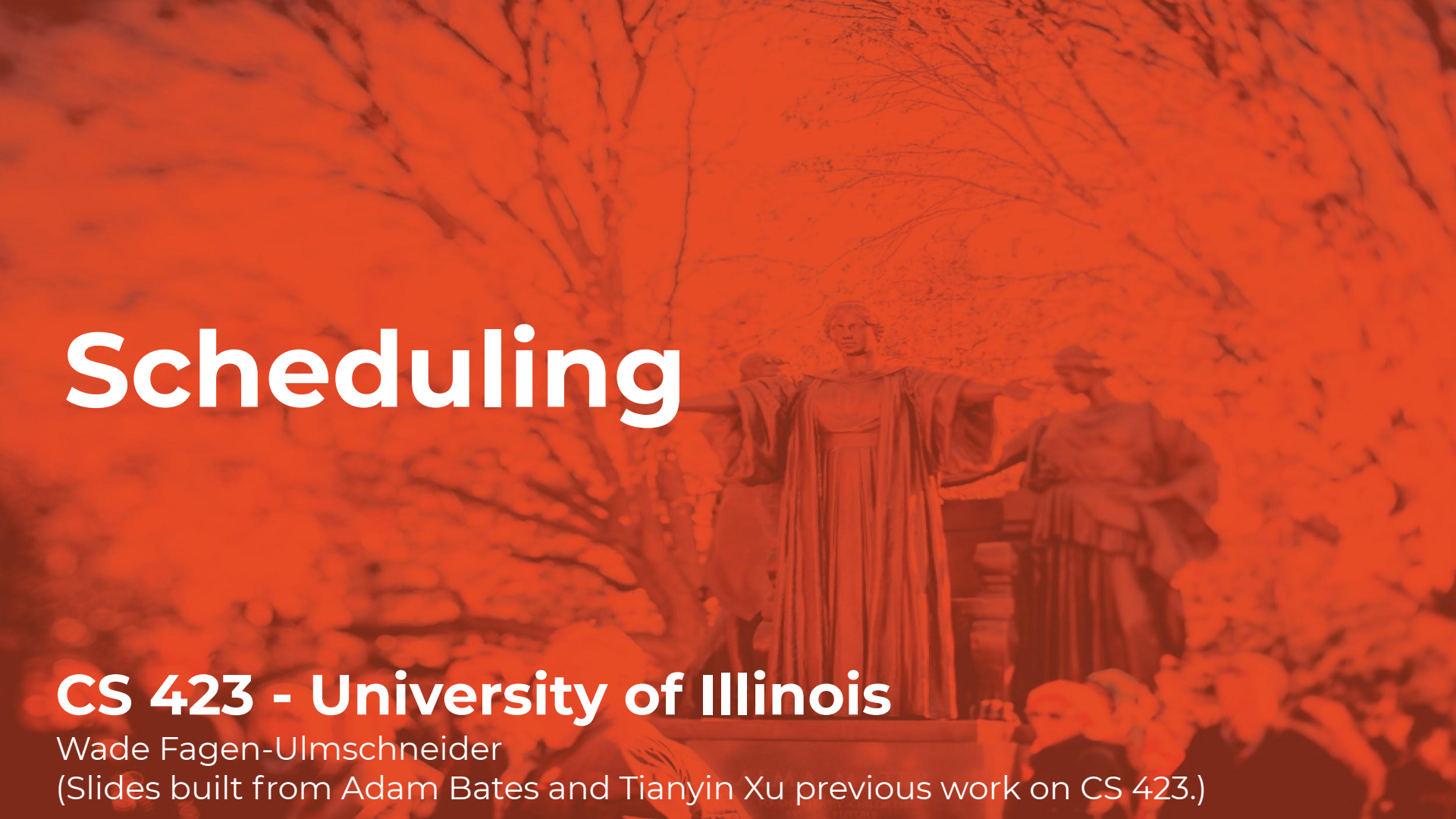
Semaphore:

```
Lock::acquire() {
    disableInterrupts();
    spinLock(&lock);
    if (value == 0) {
        waiting.add(myTCB);
        spinLockRelease(&lock);
        sch_move_to_blocked(myTCB);
        /* only returns when RUNNING */
        spinLock(&lock);
    } else {
        value--;
    }
    spinLockRelease(&lock);
    enableInterrupts();
}
```

Semaphores Exist before CVs

- ★ Semaphores conflate the roles of locks and condition variables (mutual exclusion, shared data).

- ★ Previously, semaphores were used to implement CVs -- but modern systems provide direct support for CVs.
  - ○ *Limited need for semaphores in modern codebases.*



Scheduling

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Scheduling:

Multiprocessor: 400 threads in the ready queues of four cores – which one to run next on which core?

Cluster: 1000 MapReduce jobs – which one to run on which machine and on which core?

Datacenters: 10000 user request – which one to run on which datacenter on which cluster on which machine?

Scheduling Complexity

Jobs/requests are **not created equal**

- Some are more important than the others

Jobs/requests could have **deadlines**

- Finishing late means nothing but wasting resources.

Jobs/requests have **constraints**

- Affinity is important – same node and same PCIe switch for GPUs

Workloads could be very different.

Scheduling

- **Always** an active research topic:
  - *Everyone wants run more jobs with less resources.*

Scheduling: Goals

Generate illusion of concurrency

Maximize resource utilization (e.g., mix CPU and I/O bound processes appropriately)

Meet needs of both I/O-bound and CPU-bound processes

- Give I/O-bound processes better interactive response
- Do not starve CPU-bound processes

Support Real-Time (RT) applications

Scheduling: Terms

Task/Job: Something that needs CPU time (thread, softirq, etc),

First Response Time: How long does a task take to start?

Latency/Turnaround Time: How long does a task take to complete?

Throughput: How many tasks can be done per unit of time?

Workload: Set of tasks for system to perform

Scheduling: Terms

Overhead: How much extra work is done by the scheduler?

Fairness: How equal is the performance received by different users?

Predictability: How consistent is the performance over time?

Starvation: A task never receives the resources it needs to complete. *(Not very fair!)*

Work-conserving: Resource is used whenever there is a task to run

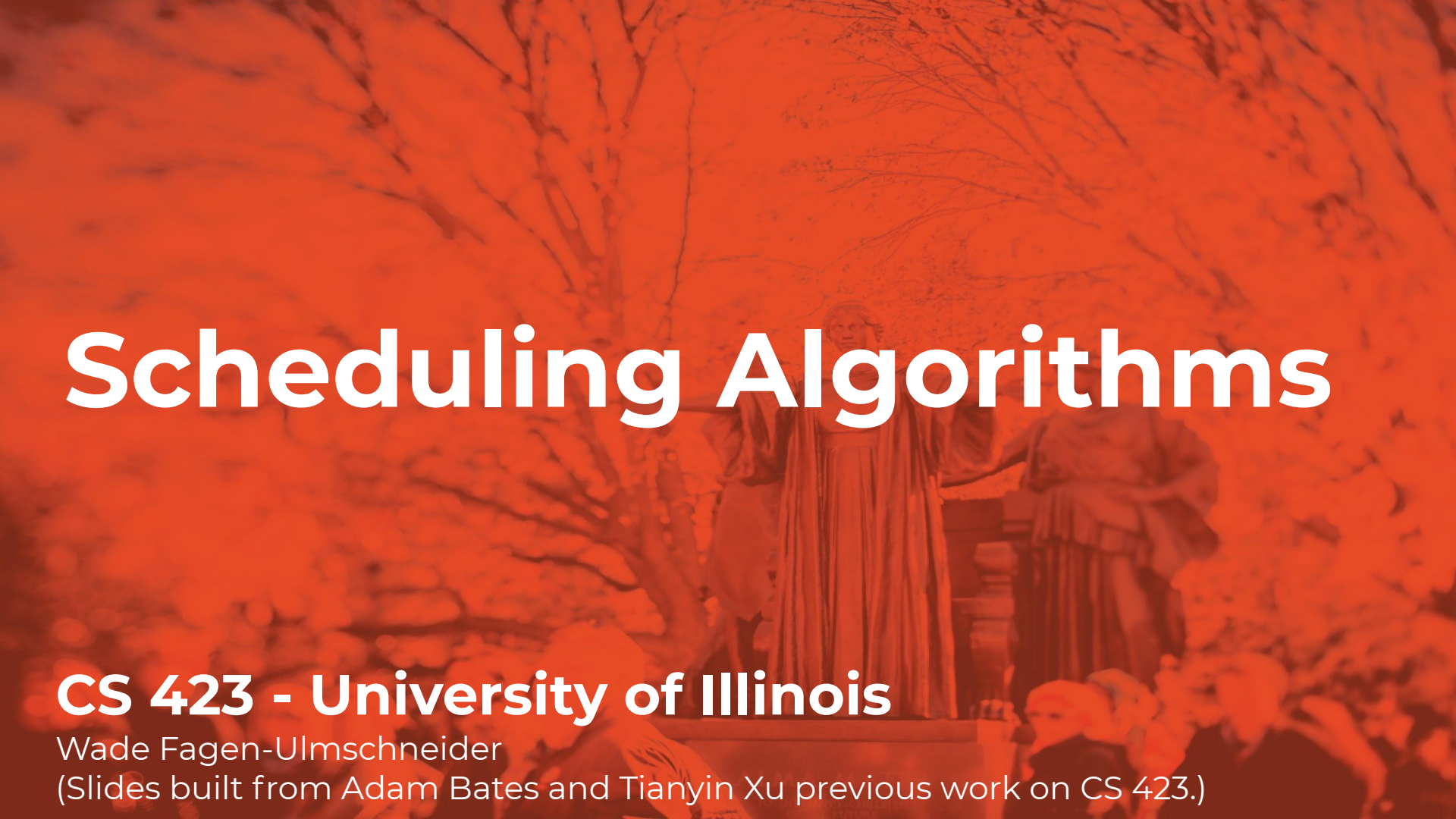
Scheduling: Terms

Non-preemptive scheduling: The running process keeps the CPU until it voluntarily gives up the CPU

- Ex: Interrupt handling in modern systems; real-time systems; many types of embedded systems.

Preemptive scheduling: The running process can be interrupted and must release the CPU (can be forced to give up CPU)

Scheduling algorithm: takes a workload as input and decides on a job to be run next



Scheduling Algorithms

CS 423 - University of Illinois

Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

First In, First Out (FIFO)

Algorithm: Run the earliest job to have arrived next.

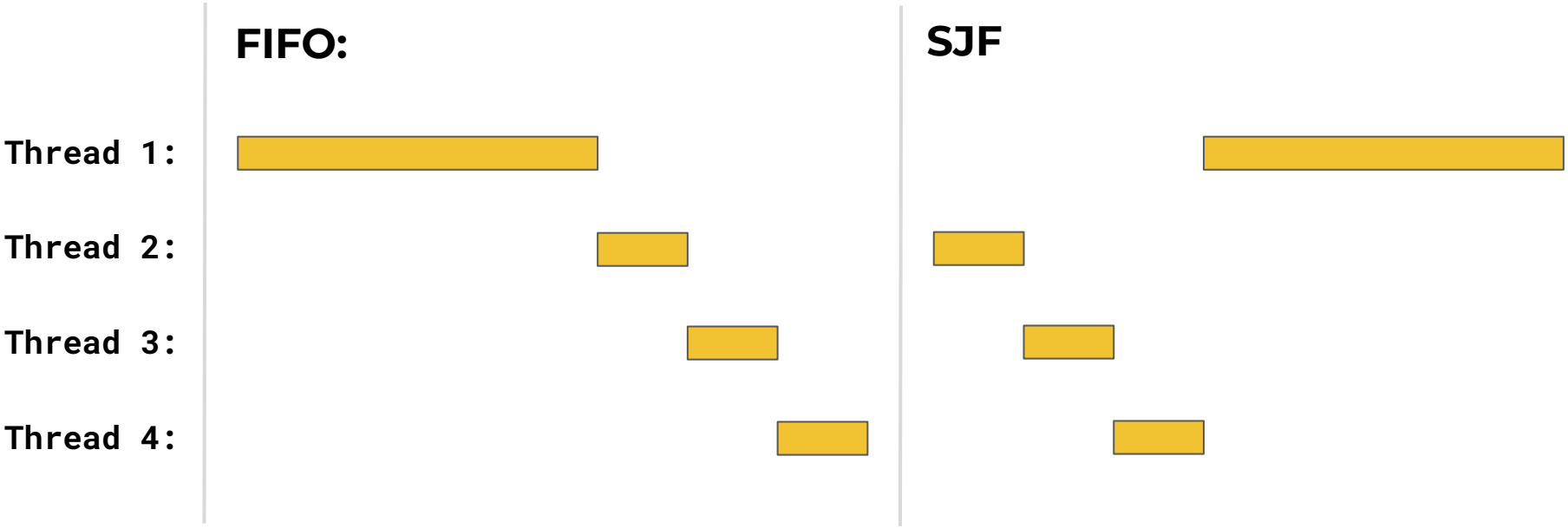
- AKA: First Come, First Served (FCFS)

Shortest Job First (SJF)

Algorithm: Run the job with the shortest remaining work to do.

- AKA: Shortest Remaining Time First (SRTF)

FIFO vs SJF



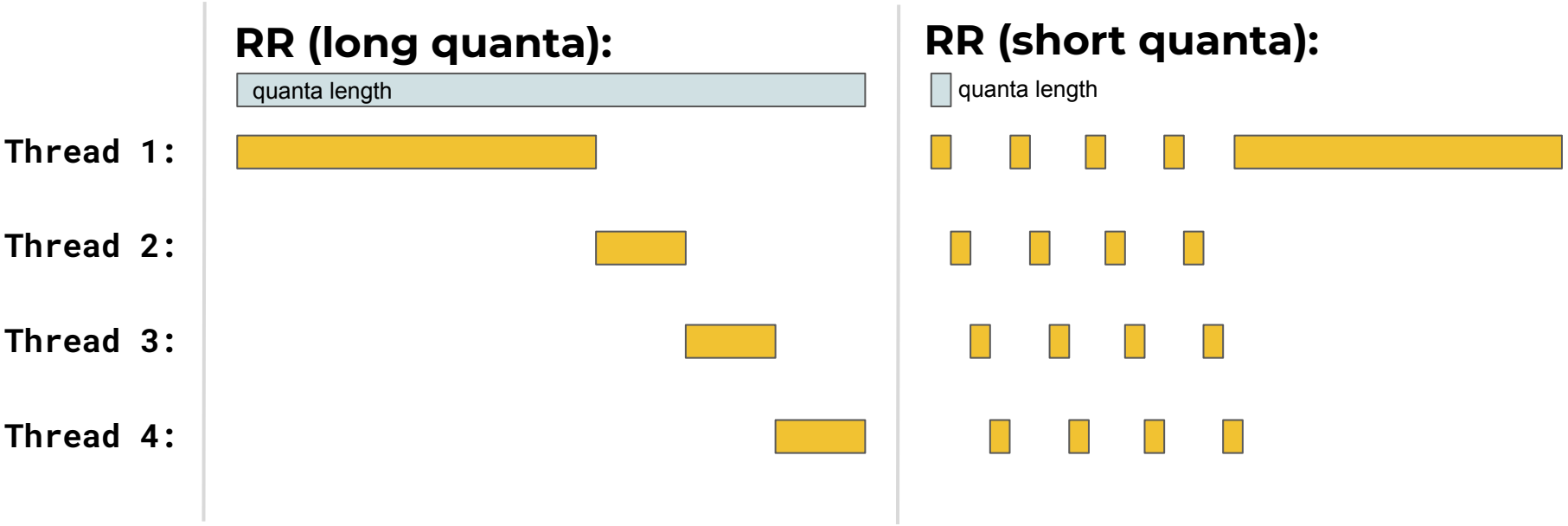
★ All four threads arrive at the exact same time.

Round Robin (RR)

- **Algorithm:** Each task gets resource for a fixed period of time (time quantum). If task doesn't complete, it goes back in line.

- Characteristics varies based on the time quantum:
  - Extremely Short Quantum: Similar to SJF
  - Extremely Long Quantum: Identical to FIFO

FIFO vs SJF

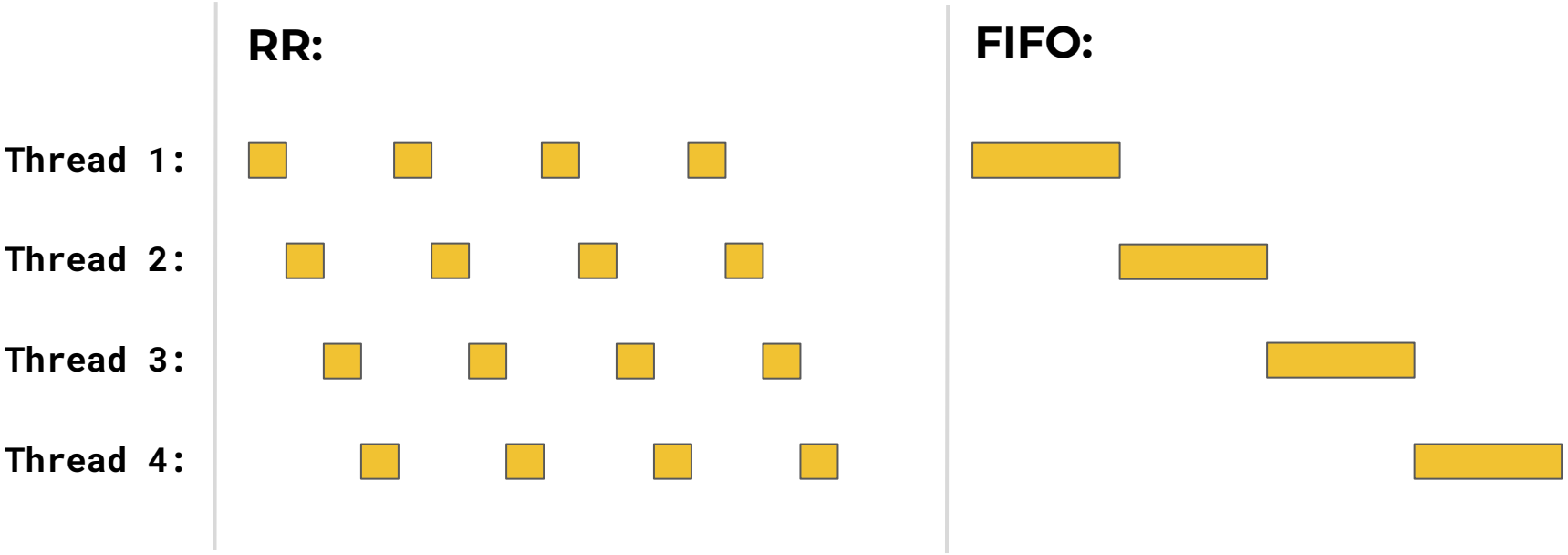


★ All four threads arrive at the exact same time.

FIFO vs RR?

- **Q:** Assuming no overhead, is RR always better than FIFO?

RR vs FIFO



★ All four threads arrive at the exact same time.

Measuring Scheduling Algorithms Is Hard

★ Measurement must be done over a **long** period of time, short-period sampling can introduce bias.

★ Start and stop the measurement during idle time when there's no work to be done by the system.

Scheduling: Min-Max Fairness

CS 423 - University of Illinois

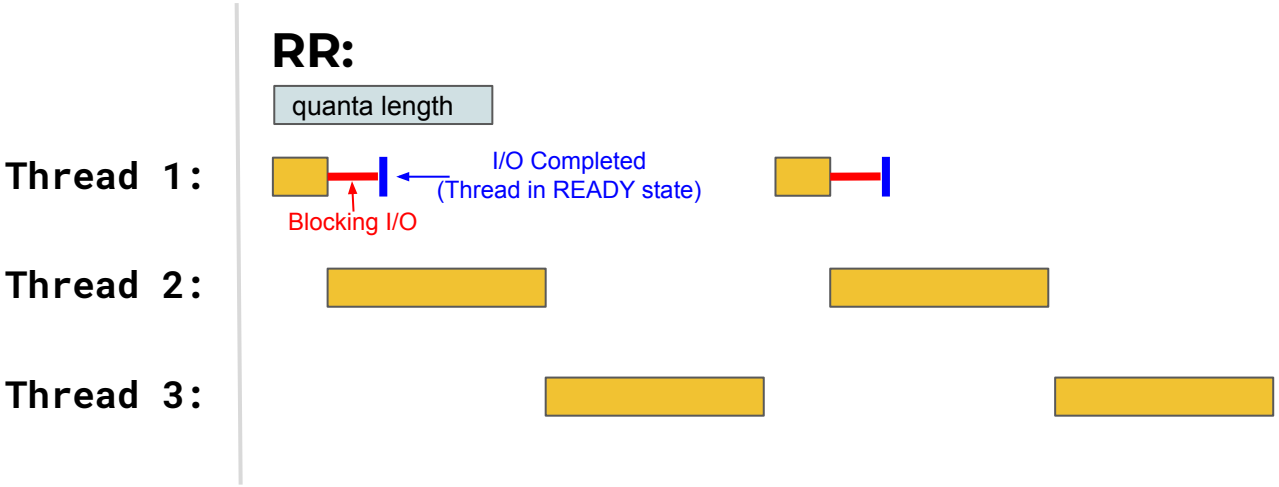
Wade Fagen-Ulmschneider
(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

What is fairness?

Lots of different ideas of “fairness”:

- FIFO?
- Equal share of the CPU?
- What if some tasks don’t need their full share?
- Minimize worst case divergence?
- Time task would take if no one else was running?
- Time task takes under scheduling algorithm?

RR vs FIFO



Using RR, “Thread 1” is disadvantaged for giving up the CPU!

…that does not seem fair…

Idea: Max-Min Fairness

Theoretical Idea:

- The least demanding task will get its share first,
- Then the next least demanding,

...etc...

Idea: Max-Min Fairness

Theoretical Idea:

- The least demanding task will get its share first,
- Then the next least demanding,

...etc...

Implementation: We want to maximize the minimum allocation given to each task:

- If any task needs less than it's equal share, schedule the smallest of these first. *(May result in left over share to split among others.)*
- Split the remaining time using max-min
- If all remaining tasks need at least equal share, split evenly.

Algorithm: Multi-level Feedback Queue (MFQ)

- ★ Some jobs will require a short CPU time before the next blocking operation (“less demanding”).
  - “I/O Bound Tasks”

- ★ Other jobs will require a large amount of CPU time (“more demanding”).
  - “CPU Bound Tasks”

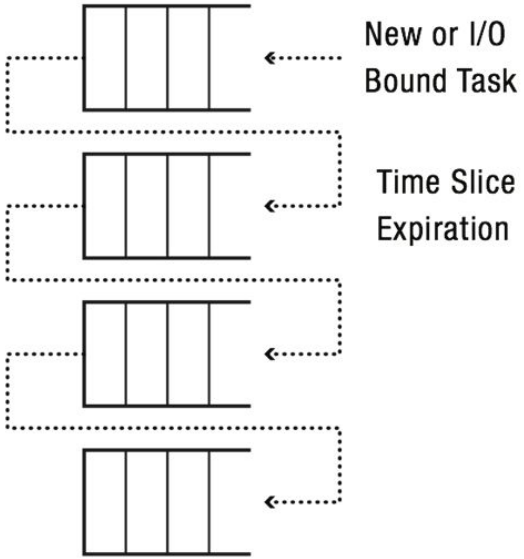
Algorithm: Multi-level Feedback Queue (MFQ)

- ★ **Algorithm:** Given a small, initial amount of CPU time to every task as soon as it needs the CPU (“P1 queue”).
- ★ If the task still needs additional CPU time, move the job to a lower priority queue (ex: “P2 queue”).
- ★ All jobs in the highest priority queue will run first, but CPU time allocated increases in the lower-priority queues.

Algorithm: Multi-level Feedback Queue (MFQ)

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

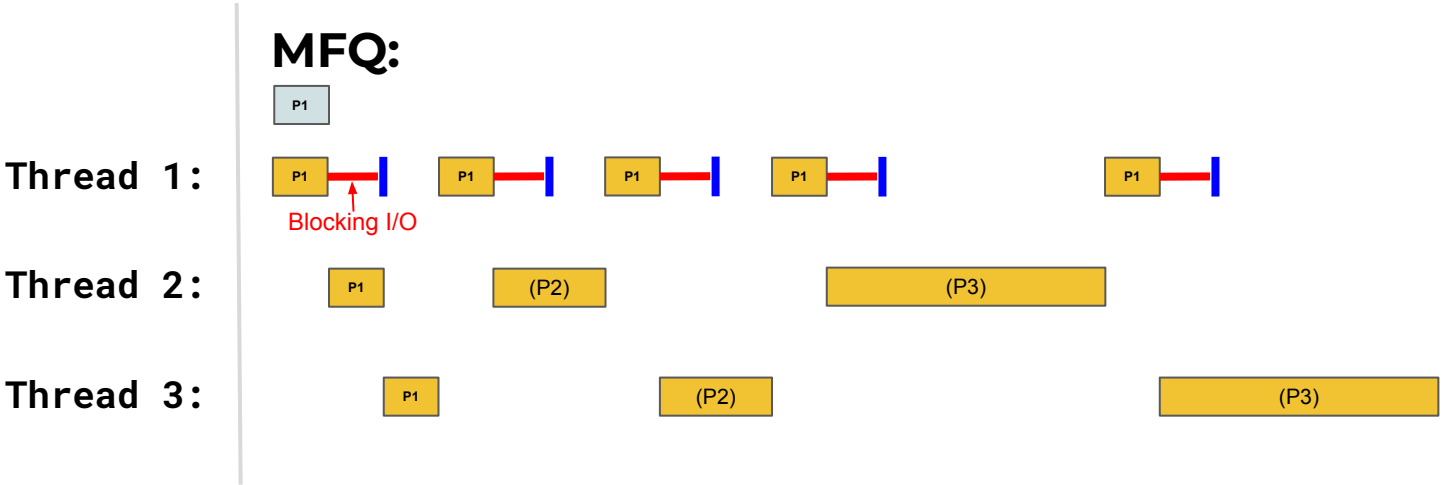
Round Robin Queues



Algorithm: Multi-level Feedback Queue (MFQ)

- ★ **Set of Round Robin queues**
  - Each queue has a separate priority
- ★ **High priority queues have short time slices**
  - Low priority queues have long time slices
- ★ **Scheduler picks first thread in highest priority queue**
  - Tasks start in highest priority queue
- ★ **If time slice expires, task drops one level**

RR vs FIFO


MFQ:
P1
Thread 1:
P1
P1
P1
P1
P1
Blocking I/O
Thread 2:
P1
(P2)
(P3)
Thread 3:
P1
(P2)
(P3)