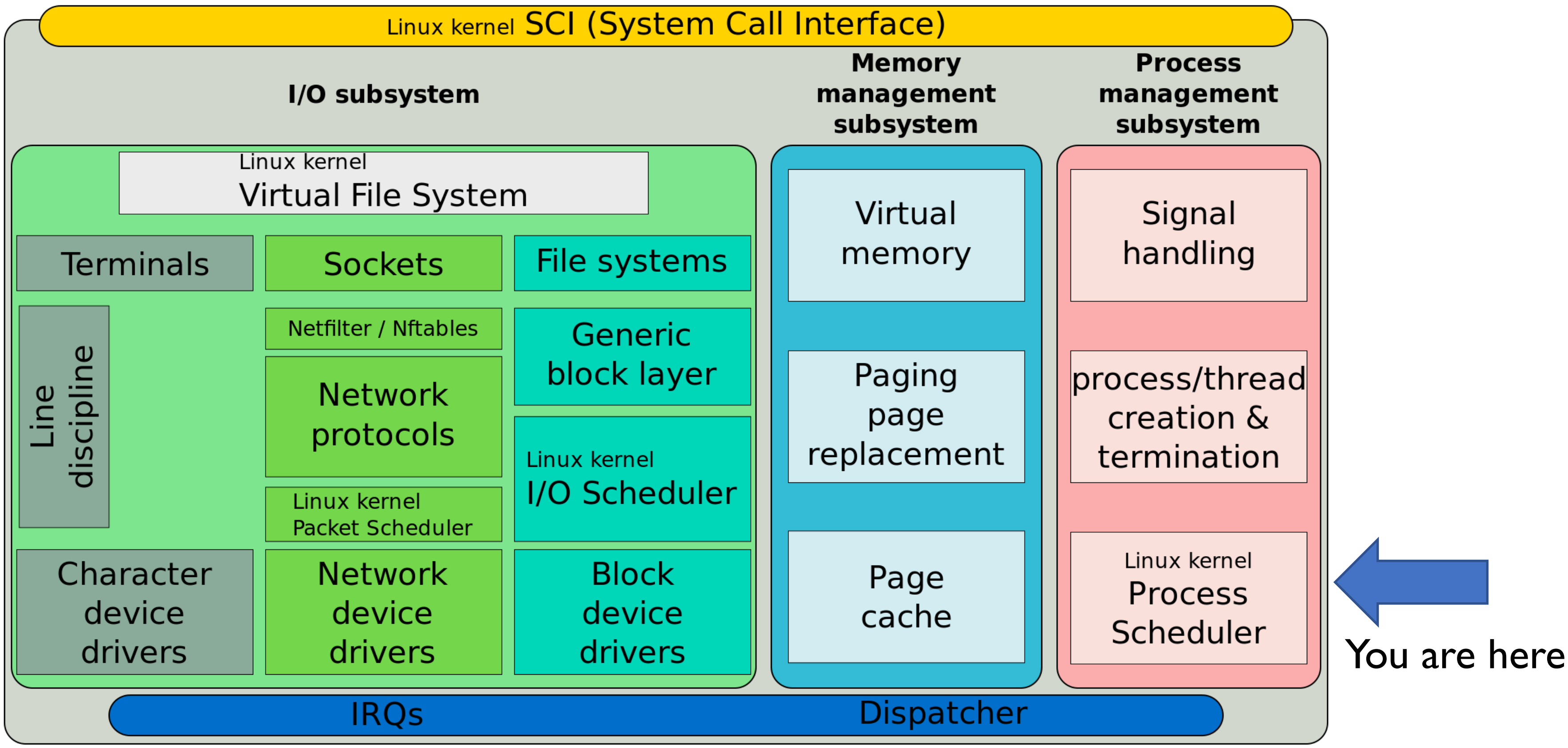
CS 162: Operating Systems and Systems Programming

Lecture 6: Scheduling - Basics

September 17, 2019
Instructor: David Culler
https://cs162.eecs.berkeley.edu

Read: 3 Easy Pieces: ch 7
A&D 7.1
HW 1 Dues Wed 9/18
Proj 1 Design Doc

Where are we?



Linux kernel SCI (System Call Interface)
I/O subsystem
Memory management subsystem
Process management subsystem
Linux kernel Virtual File System
Terminals
Sockets
File systems
Line discipline
Netfilter / Nftables
Generic block layer
Network protocols
Linux kernel I/O Scheduler
Linux kernel Packet Scheduler
Character device drivers
Network device drivers
Block device drivers
Virtual memory
Paging page replacement
Page cache
Signal handling
process/thread creation & termination
Linux kernel Process Scheduler
IRQs
Dispatcher
You are here

Recall: What's below the surface ??

File descriptor number
- an int

Application / Service

High Level I/O — streams

Low Level I/O — handles

Syscall — registers

File System — descriptors

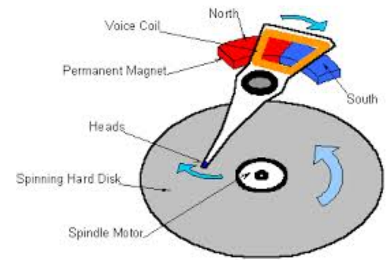
I/O Driver — Commands and Data Transfers

Disks, Flash, Controllers, DMA

File Descriptors
- a struct with all the info about the files







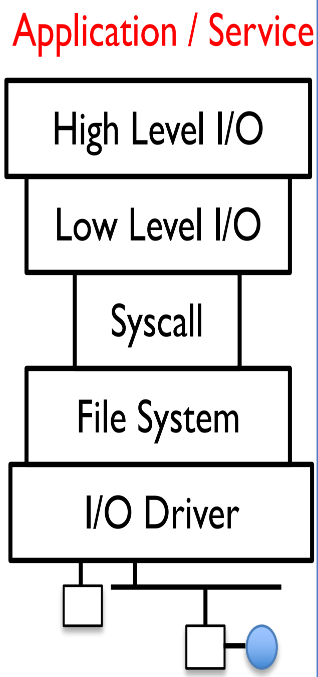
Recall: Layer by layer

User App

```
length = read(input_fd, buffer, BUFFER_SIZE);
```

User library

```
ssize_t read(int, void *, size_t){
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
};
```



Exception U→K, interrupt processing

```
Void syscall_handler (struct intr_frame *f) {
    unmarshall call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
}
```

```
ssize_t vfs_read(struct file *file, char
__user *buf, size_t count, loff_t *pos)
{
      UserProcess/File System relationship
      call device driver to do the work
}
```

Device Driver

Recall: C Low Level I/O Operations

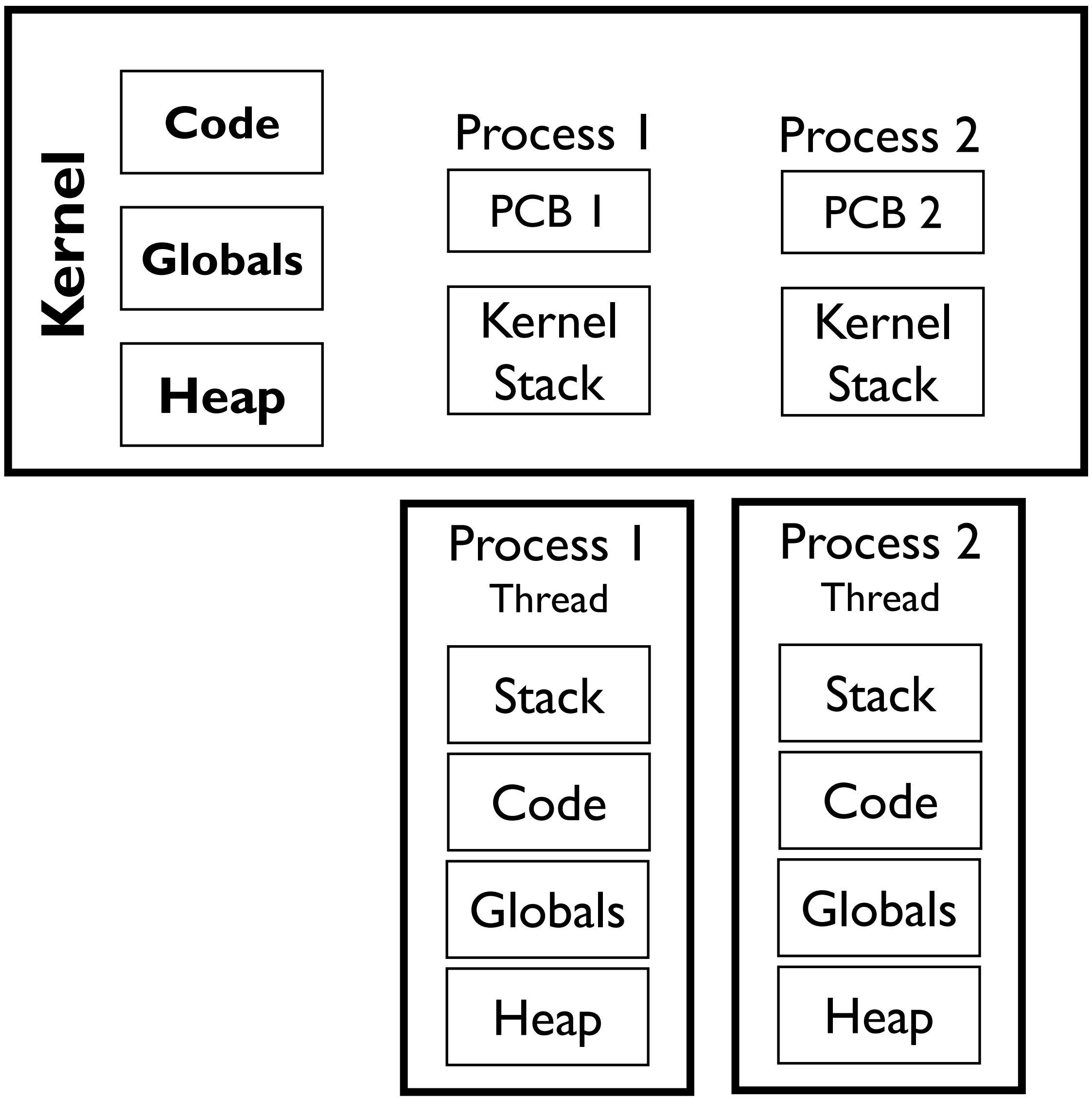
```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish
```

- When write returns, data is on its way to disk and can be read, but it may not actually be permanent!
- Low I/O has int "descriptors" – kernel holds real descriptor in PCB, just a handle
- Buffering reflects a kernel/user "compromise" protocol
  - Kernel will read/write what it can buffer and tell the user what it did
  - User needs to loop on read/write to complete the entire transfer
- What type of object is written/read?

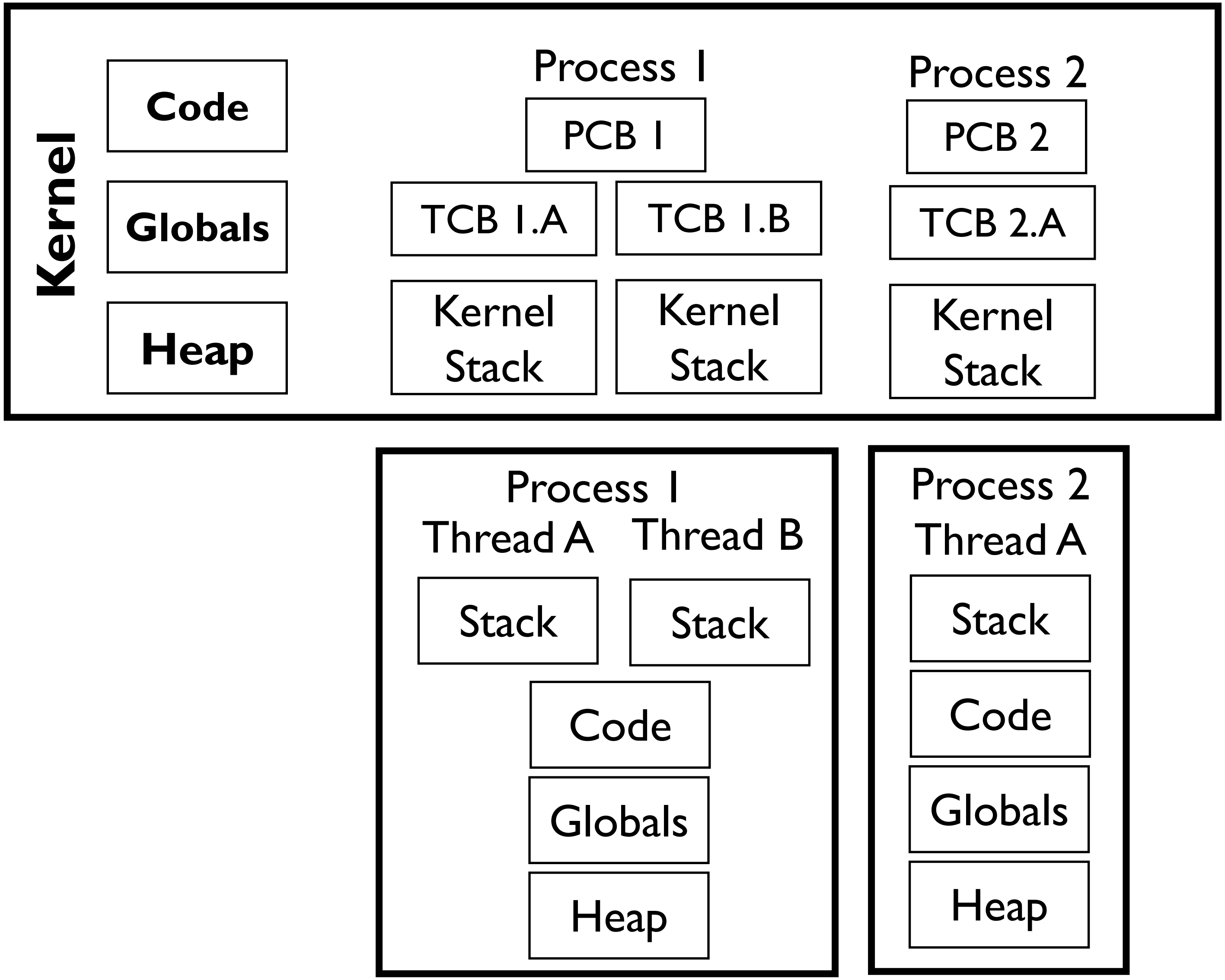
Recall: Kernel-Supported Threads

- Kernel-Supported Thread: OS stores thread control block, schedules thread directly
  - Block independently on I/O, processes may wait on multiple events

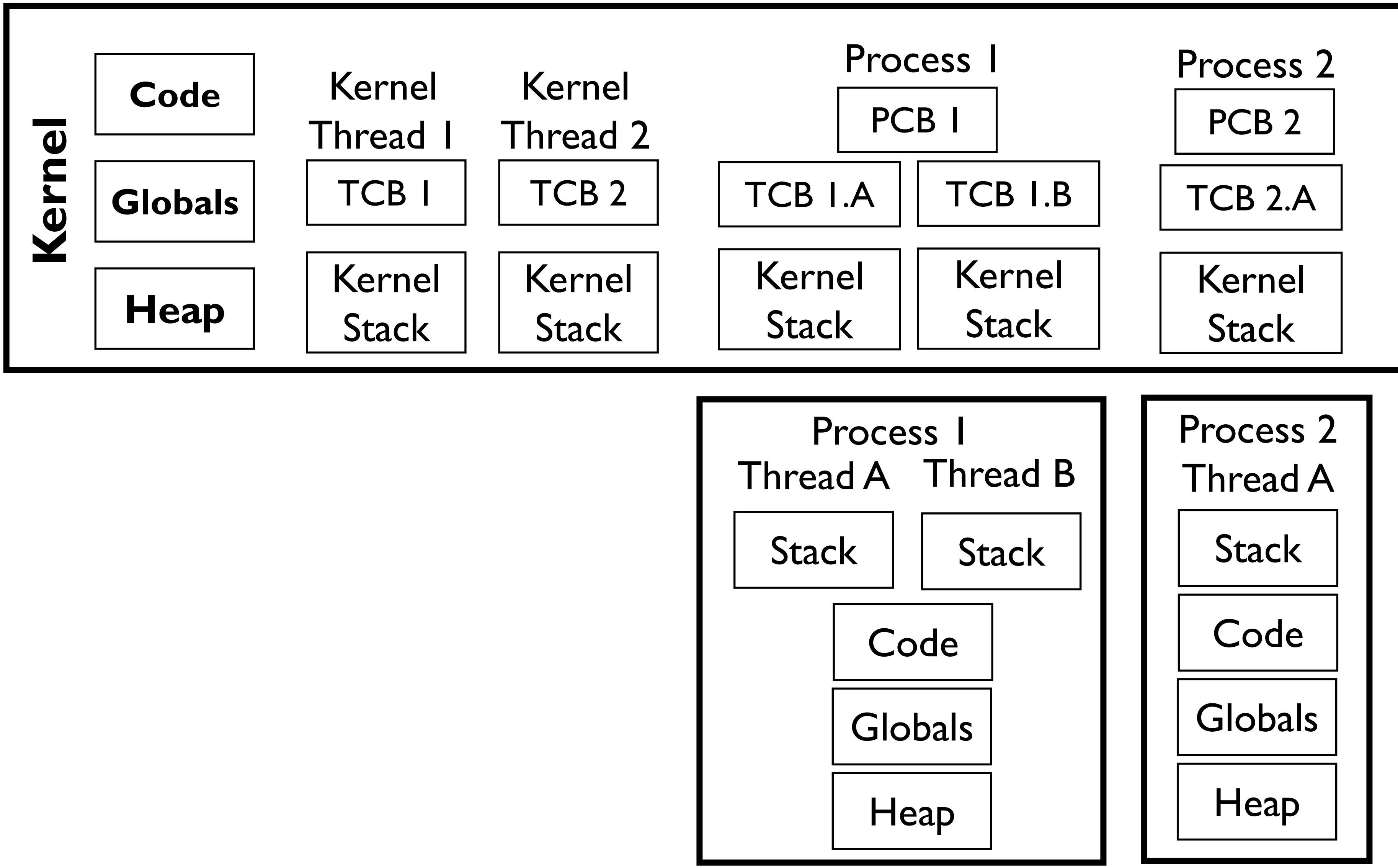
Recap So Far: Kernel Structure



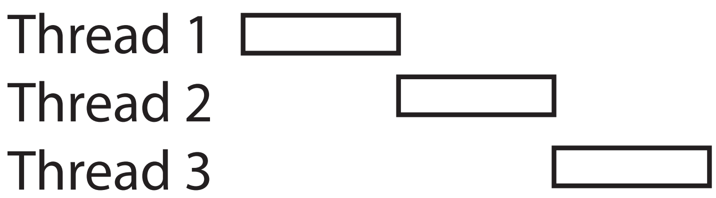
Recap So Far: Kernel Structure



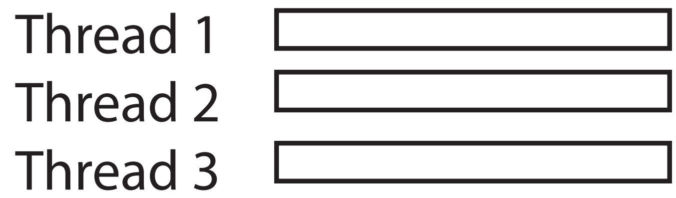
Recap So Far: Kernel Structure



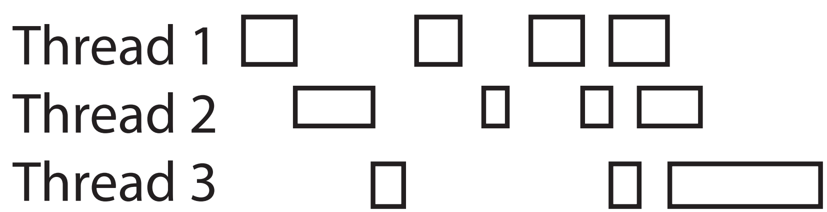
Recall: Possible Executions



a) One execution

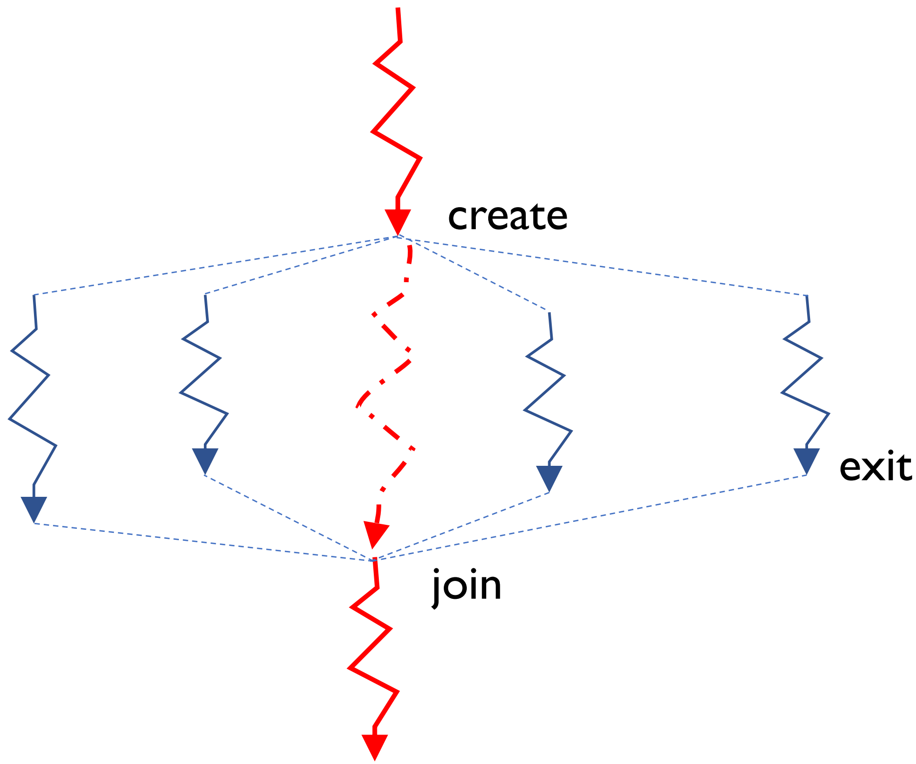


b) Another execution



c) Another execution

Fork-Join Pattern



- Main thread *creates* (forks) collection of sub-threads passing them args to work on, *joins* with them, collecting results.

Recall: Synchronization

- **Mutual Exclusion:** Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)

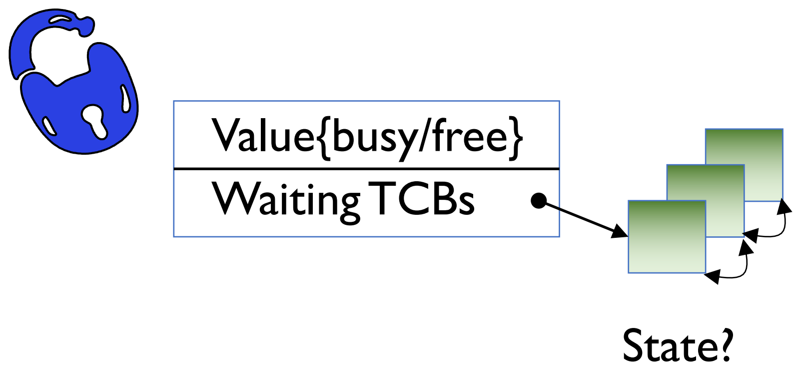
- **Critical Section:** Code exactly one thread can execute at once
  - Result of mutual exclusion

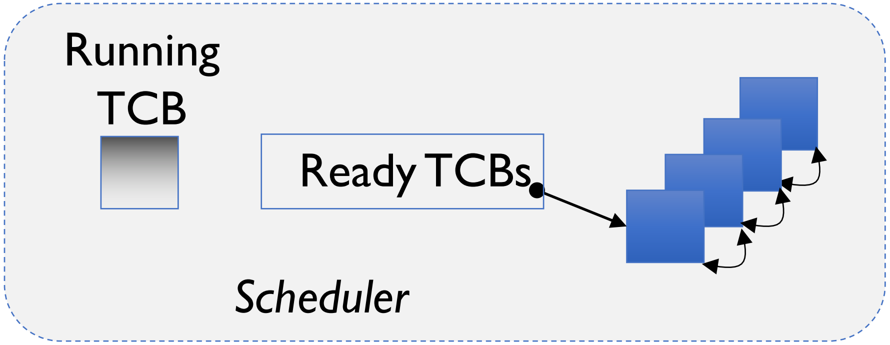
Recall: Locks

- **Lock:** An object only one thread can hold at a time
  - **Provides** mutual exclusion

- Offers two **atomic** operations:
  - `Lock.Acquire()` – wait until lock is free; then grab
  - `Lock.Release()` – Unlock, wake up waiters

Recall: Basic Lock Implementation






```
Acquire(*lock) {
   disable interrupts;
   if (lock->value == BUSY) {
      put thread on lock's wait_Q
      "i.e, Go to sleep"
       allow a ready thread to run
   } else {
      lock->value = BUSY;
   }
   enable interrupts;
}
```

```
Release(*lock) {
   disable interrupts;
   if (any TCB on lock wait_Q) {
      "i.e., lock busy";
      take thread off wait queue
      Place on ready queue;
   } else {
      lock->value = FREE;
   }
   enable interrupts;
}
```

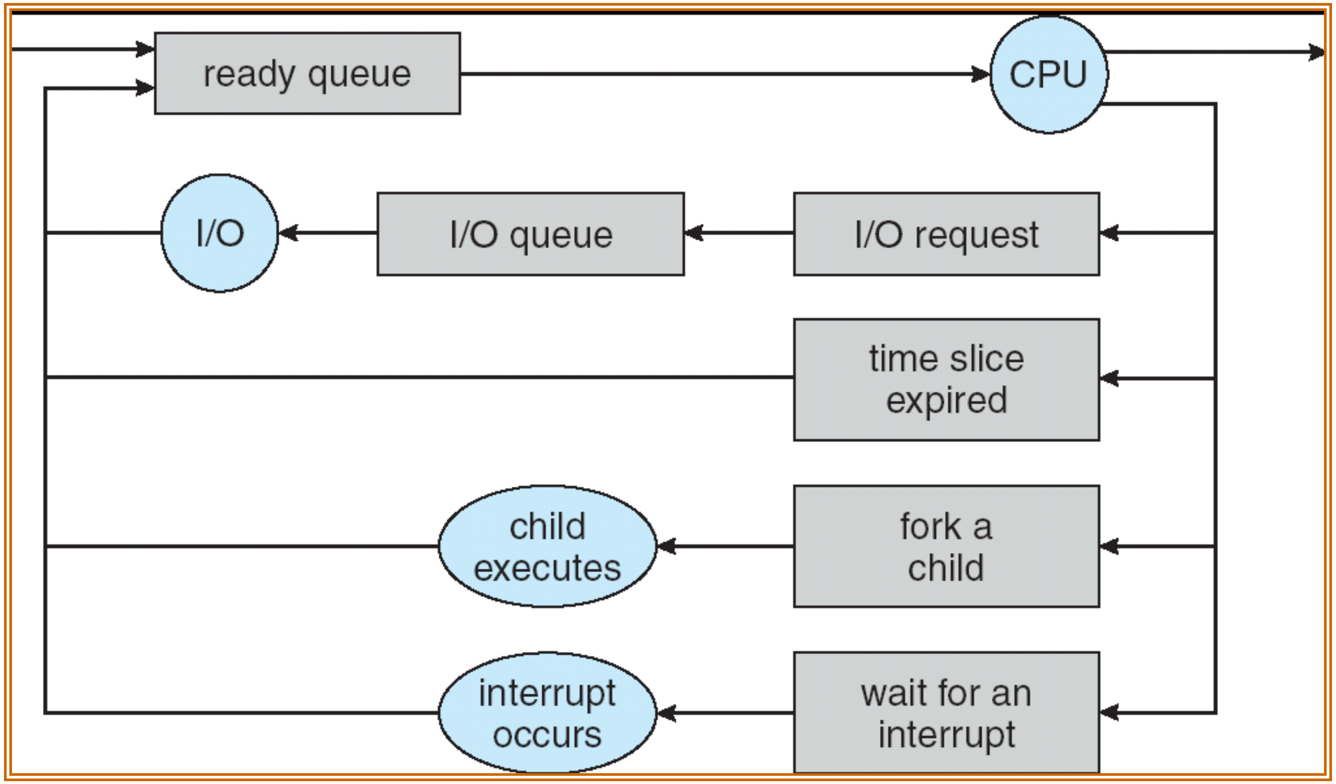
Today: Scheduler

```
if ( readyThreads(TCBs) ) {
    nextTCB = selectThread(TCBs);
    run( nextTCB );
} else {
    run_idle_thread();
}
```

- Scheduler: **Which thread should run on the CPU next?**

Scheduling: All About Queues

Scheduling: All About Queues

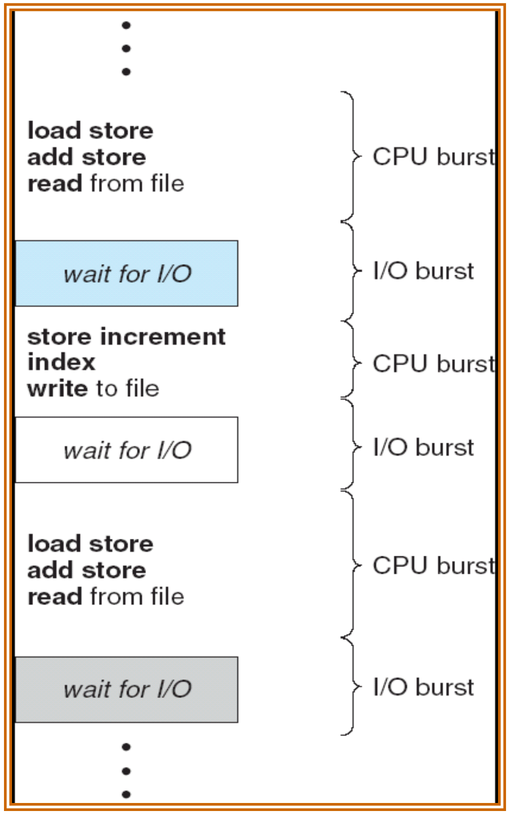


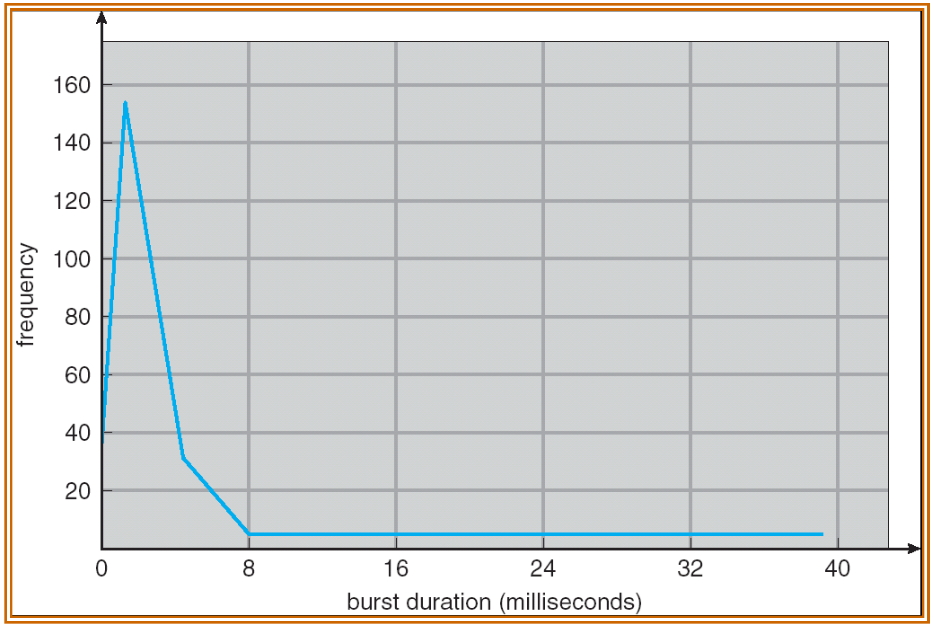
Processor Scheduling: Which thread to remove from ready queue?

Scheduling: All about trade-offs

- Individuals care about getting their task done quickly
- System cares about overall efficiency
  - Utilize multiple HW resources well, low overhead, …
- Huge variation in job characteristics
- Fairness ???
- Utility function
  - deadlines?, interactivity?
- …

CPU & I/O Bursts





- Programs alternate between bursts of CPU, I/O activity
- Scheduler: Which thread (CPU burst) to run next?
- **Interactive programs vs Compute Bound vs Streaming**

Evaluating Schedulers

- **Response Time** (ideally *low*)
  - What user sees: from keypress to character on screen
  - Or completion time for non-interactive
- **Throughput** (ideally *high*)
  - Total operations (jobs) per second
  - Overhead (e.g. context switching), artificial blocks
- **Fairness**
  - Fraction of resources provided to each
  - May conflics with best avg. throughput, resp. time

Scheduling Assumptions

- Equal or variable job length ?
- Run to completion vs preemption ?
- Arrival time (at once vs varied) ?
- Resources: CPU(s), I/O, Network, … ?
- Advanced Knowledge of Job characteristics or need
  - Off-line scheduling is given the entire collection of tasks and computes a schedule
  - On-line scheduling makes decisions as tasks arrive

First-Come First-Served (FCFS)

- Also: "First In First Out"
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Arrival *Order*: P_1, P_2, then P_3 *(essentially at time 0)*

Asmpt: V, RTC, AT=0, CPU, none

First-Come First-Served (FCFS)

- Response Times: P1 = 24, P2 = 27, P3 = 30
- Average Response Time = (24+27+30)/3 = 27

- Waiting times: P1 = 0, P2 = 24, P3 = 27
- Average Wait Time = (0 + 24 + 27)/3 = 17
- **Convoy Effect: Short processes stuck behind long processes**
  - P2, P3 arrive any time <24 wait

Slightly different arrival order?

- P2 < P3 < P1
- Response Time: P1 = 30, P2 = 3, P3 = 6
- Average Response Time = (30 + 3 + 6)/3 = 13
  - versus 27 with P1 < P2 < P3
- Waiting Time: P1 = 6, P2 = 0, P3 = 3
- Average Waiting Time = (6+0+3)/3 = 3

Asmpt: V, RTC, AT=0, CPU, none

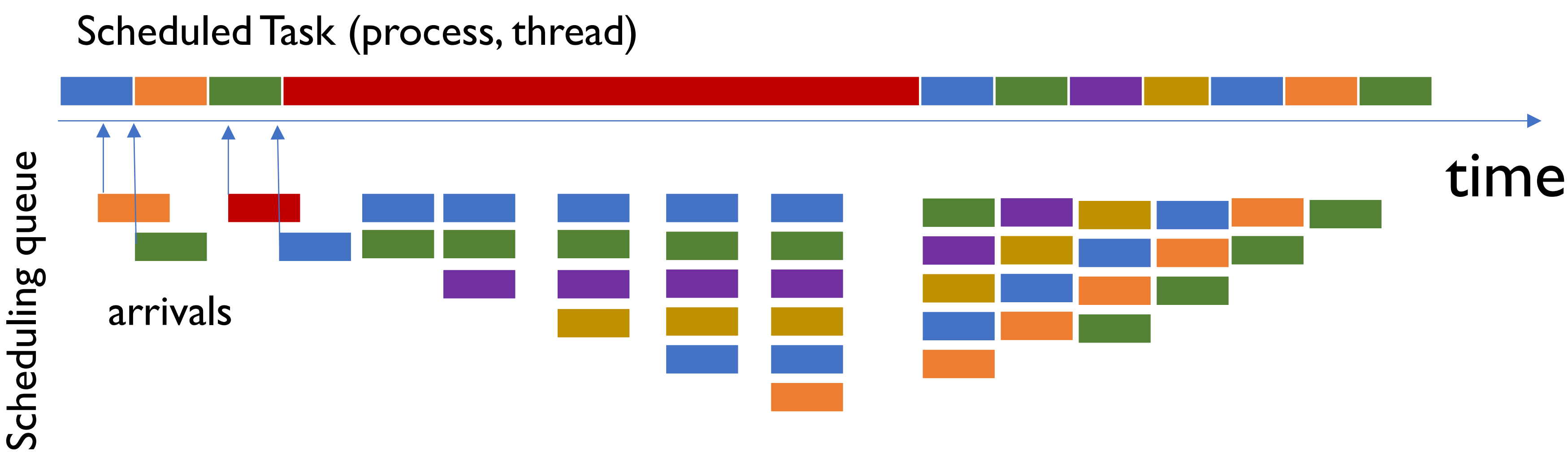
How does kernel implement FCFS ?

- Comes down to scheduling queue data structure
  - FIFO
  - eg., `push_front, pop_back`

Peer discussion

- If we have `1` long task (say `100` units) and `n` little tasks (say `1` unit), that all arrive in random order
- How long should a little task expect to wait?
- What's the chance of getting stuck behind the large one?
- What might we do to avoid this situation?

Convoy effect

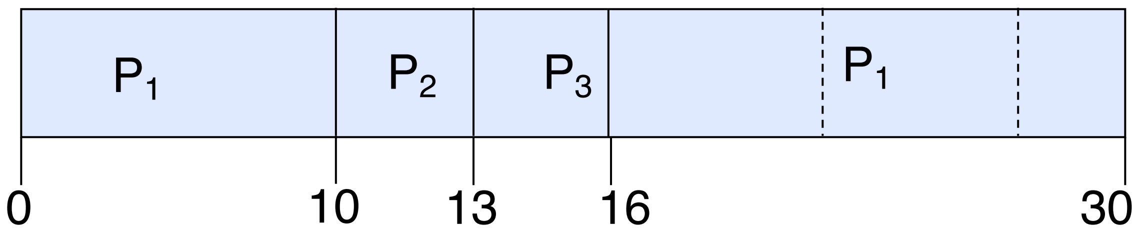


- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.

Preemption

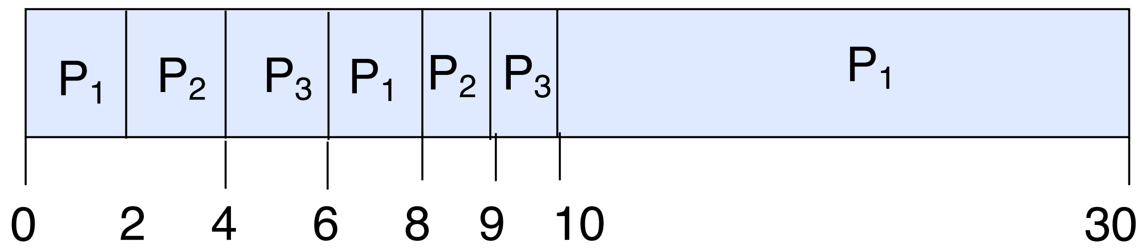
- Give out *small* units of CPU time ("time quantum")
  - Typically 10 – 100 milliseconds
- When quantum expires, **preempt,** and schedule
  - Round Robin: add to end of the queue
- Each of *N* processes gets ~1/*N* of CPU (in window)
  - With quantum length *Q* ms, process waits at most *(N-1)*Q* ms to run again
- Downside: More context switches
  - What should *Q* be?
  - Too high: back to FCFS, Too Low: context switch overhead

Our example (Q=10)



- Regardless of arrival order, short jobs gets a chance early
- Much less sensitive to arrival order
- How much context switch overhead?

Our example (Q=2)



- Smaller Q, more interactive and fair
- More overhead

Round Robin Example (Q = 20)

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

- Average response time = (125+28+153+112)/4 = 104.5
- Waiting time for P_1=(68-20)+(112-88)=72
  P_2=(20-0)=20
  P_3=(28-0)+(88-48)+(125-108)=85
  P_4=(48-0)+(108-68)=88
- Average waiting time = (72+20+85+88)/4 = 66.25
- And don't forget context switch overhead!

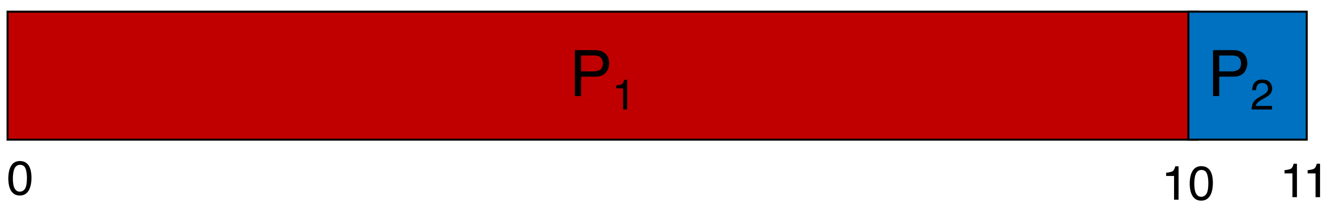
Round Robin Quantum

- Assume there is no context switching overhead
- What happens when we *decrease Q*?

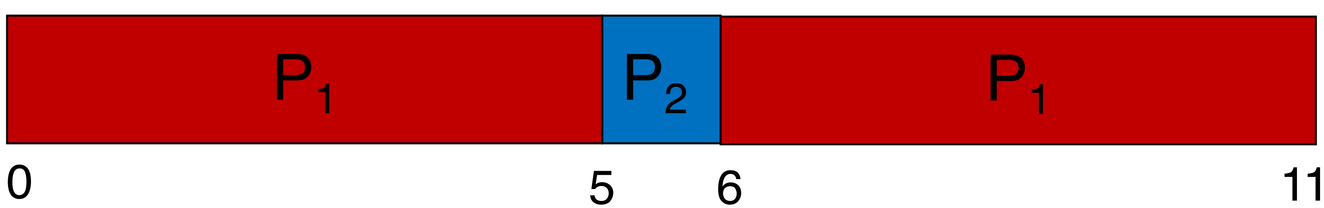
1. Avg. response time always **decreases** or **stays the same**
2. Avg. response time always **increases** or **stays the same**
3. Avg. response time can **increase**, **decrease**, or **stays the same**

Decrease Response Time

- P_1: Burst Length 10
- P_2: Burst Length 1
- Q = 10



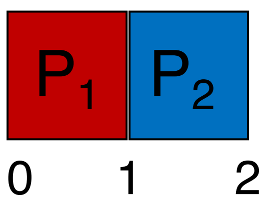
- Average Response Time = (10 + 11)/2 = 10.5
- Q = 5



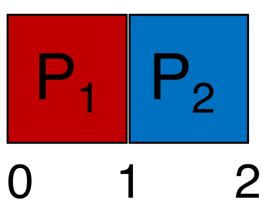
- Average Response Time = (6 + 11)/2 = 8.5

Same Response Time

- P_1: Burst Length 1
- P_2: Burst Length 1
- Q = 10



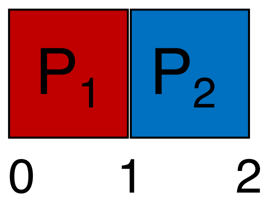
- Average Response Time = (1 + 2)/2 = 1.5
- Q = 1



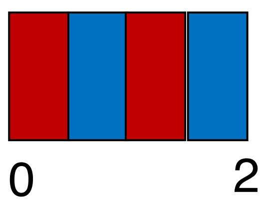
- Average Response Time = (1 + 2)/2 = 1.5

Increase Response Time

- P_1: Burst Length 1
- P_2: Burst Length 1
- $Q = 1$



- Average Response Time = (1 + 2)/2 = 1.5
- $Q = 0.5$

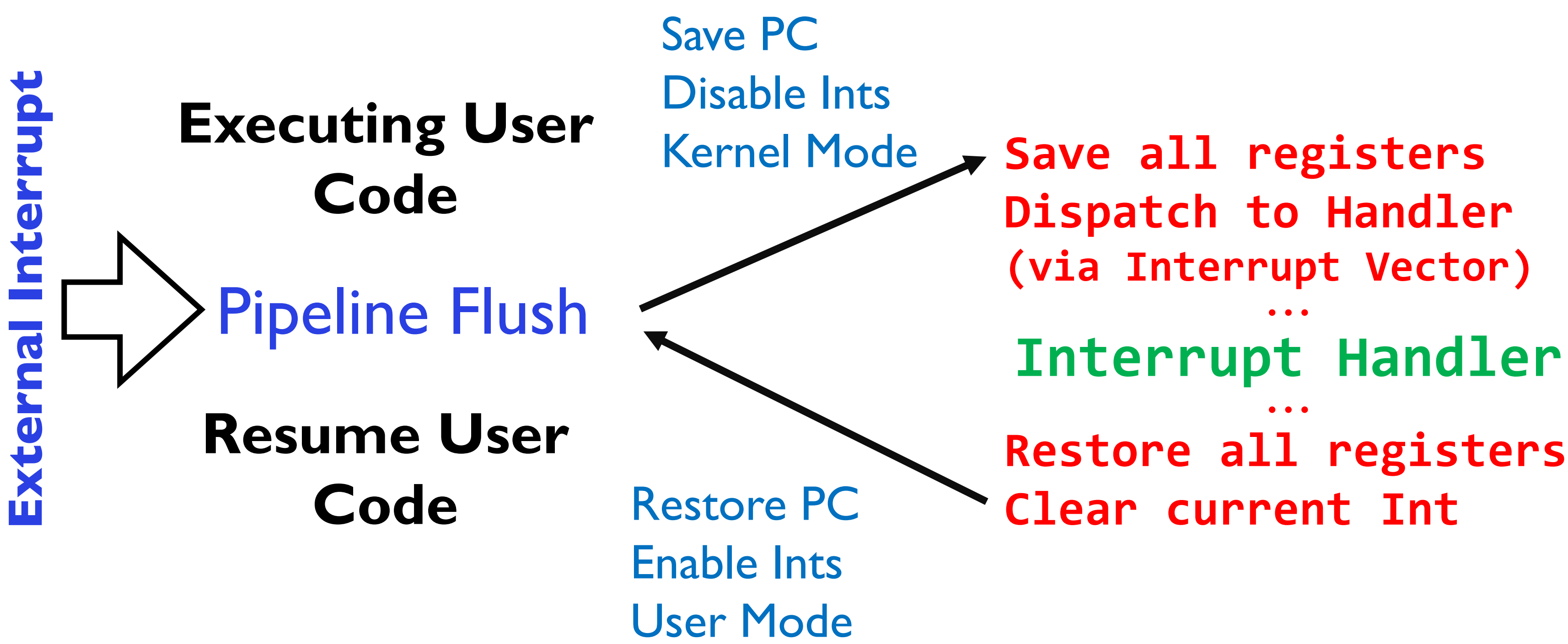


- Average Response Time = (1.5 + 2)/2 = 1.75

How does kernel realize round-robin scheduling?

Project 2.1

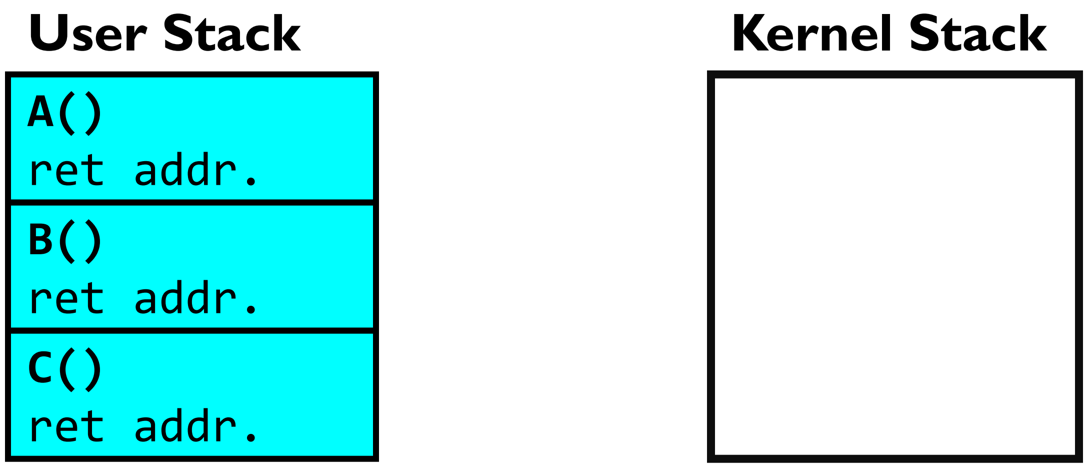
Recall: Interrupt Handling



- An interrupt is a hardware-invoked trap to the kernel
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

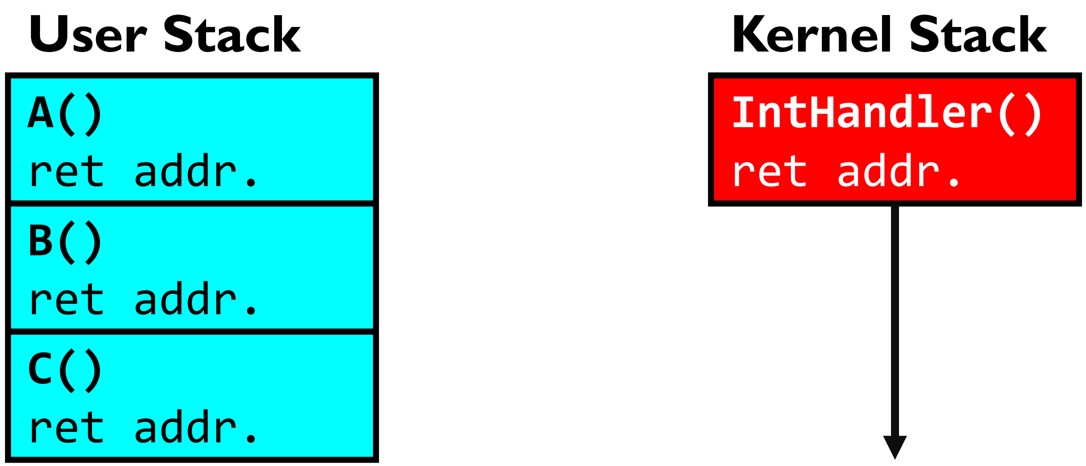
Recall: Kernel Stack

- Use protected region of memory for stack when running in Kernel Mode
- While thread is running user's code:



Recall: Kernel Stack

- When an Interrupt Occurs:



Recall: Kernel Stack

- When a Thread is Ready, but not Running
  - Waiting for return to `switch` from another thread

User Stack	Kernel Stack
`A()` `ret addr.`	**`IntHandler()`** `ret addr.`
`B()` `ret addr.`	**`run_new_thread`** `ret. addr.`
`C()` `ret addr.`	**`Switch()`**

Scheduling Opportunities

- Every “yield”
- Every syscall
- Every timer tick (interrupt)
- Every interrupt

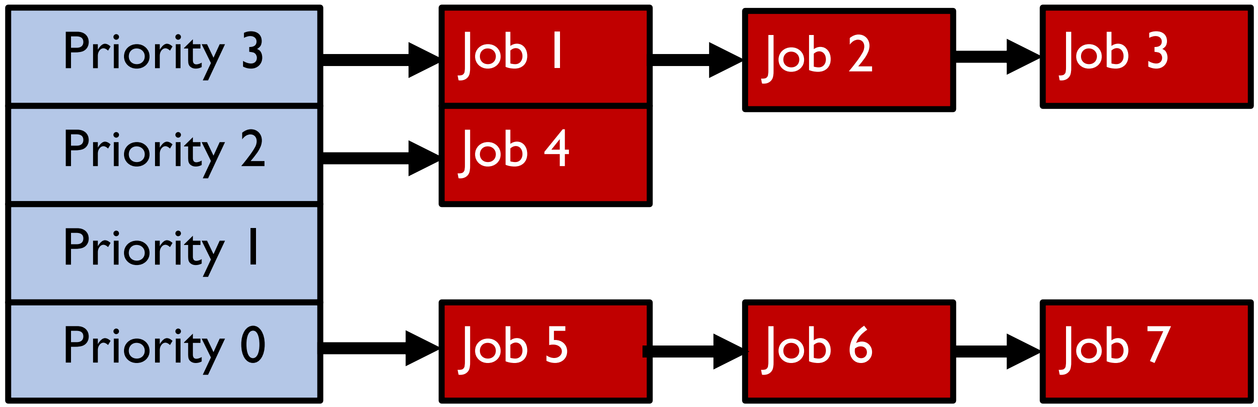
Strawman: What would LCFS scheduling do?

- Stack (LIFO) as a scheduling data structure
- Late arrivals get fast service
- Early ones wait – extremely unfair
- In the worst case – *starvation*
- When would this occur?
  - When arrival rate (offered load) exceeds service rate (delivered load)
  - Queue builds up faster than it drains
- Queue can build in FIFO too, but ”serviced in the order received”…

Priority

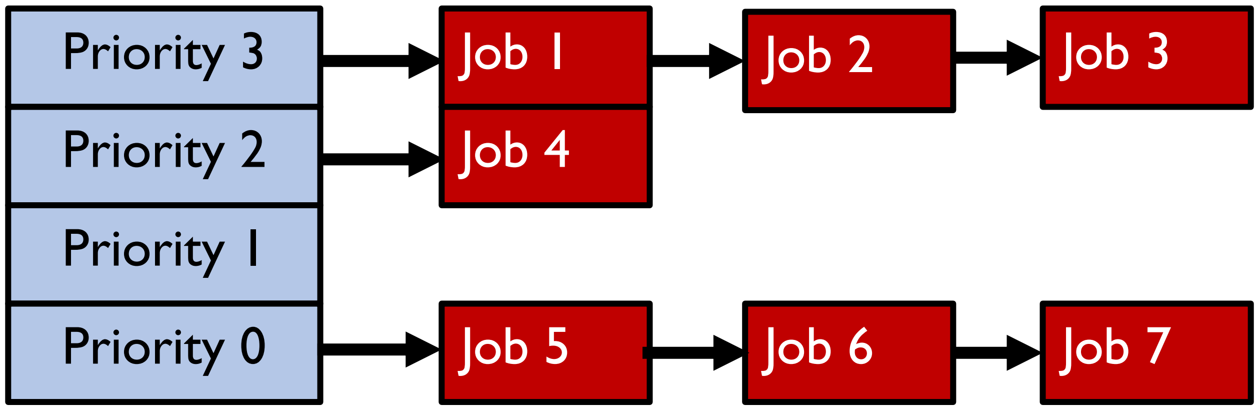
- Lots of basic tasks, versus payroll processing on Friday…
- Interactive vs compute bound

Priority Scheduling



- Something gives jobs (processes) priority
  - Usually the user sets it explicitly, perhaps based on $ rate
- Always run the **ready** thread with *highest priority*
  - Low priority thread might never run!
  - **Starvation**
- Part of Project 2

Policy based on Priority Scheduling Mechanism

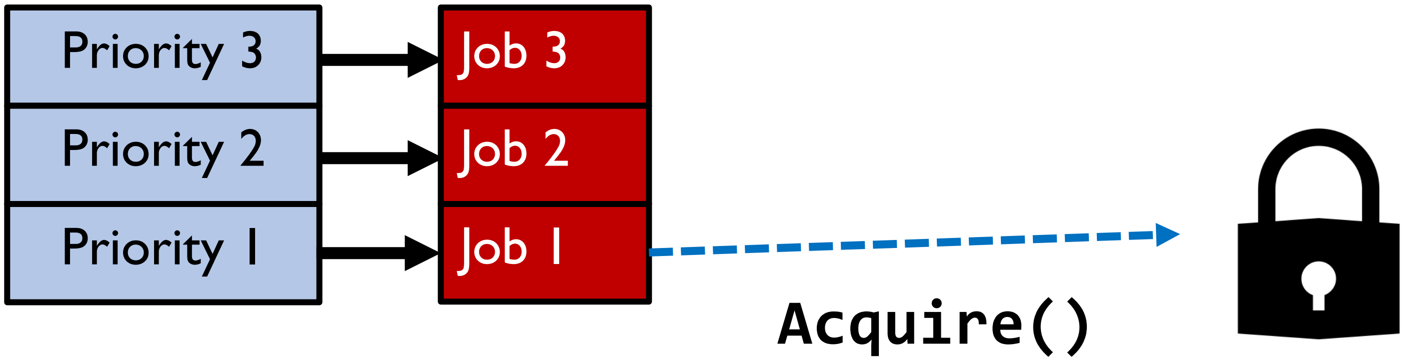


- Systems may try to set priorities according to some **policy goal**
- Example: Give interactive higher priority than long calculation
  - Prefer jobs waiting on I/O to those consuming lots of CPU
- Try to achieve fairness: elevate priority of threads that don't get CPU time (ad-hoc, bad if system overload)

How does kernel do Priority Scheduling?

- Scheduling queue data structure determines next thread **of those in the ready queue**.
- Why might a thread not be in the ready queue?
- Waiting on I/O
- Locks?

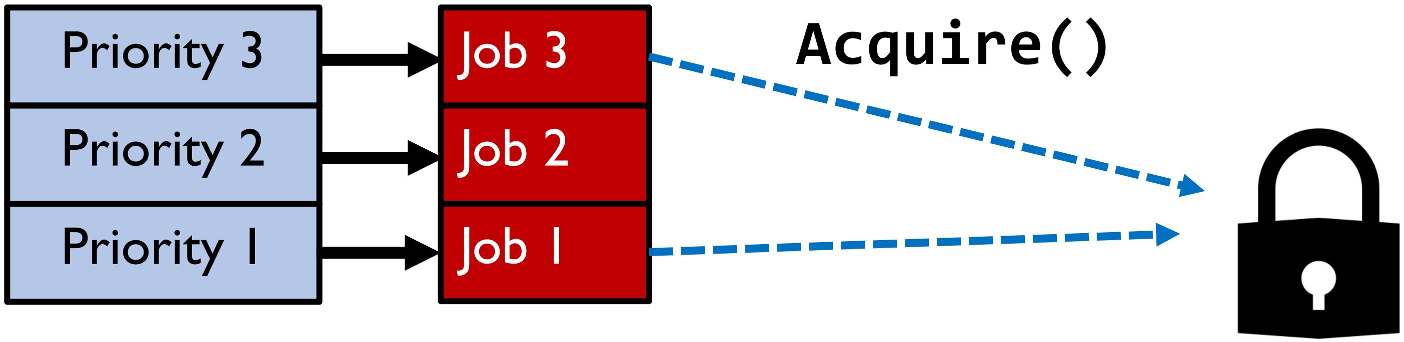
Priority and Locks



At this point, which job does the scheduler choose?

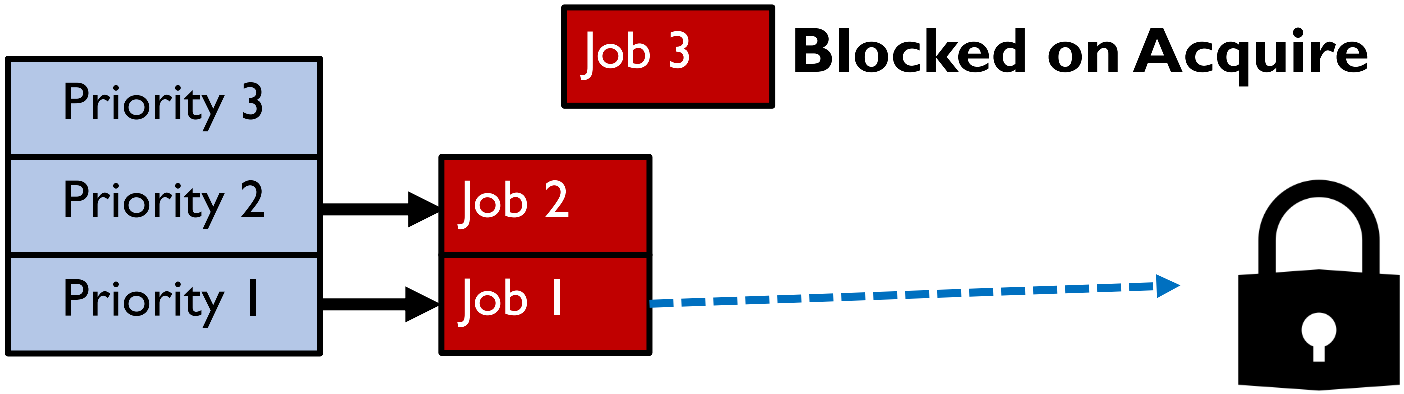
Job 3 (Highest Priority)

Priority and Locks



Job 3 attempts to acquire lock held by Job 1

Priority and Locks



At this point, which job does the scheduler choose?

Job 2 (Medium Priority)

Priority Inversion

Priority Inversion

- Where high priority task is blocked waiting on low priority task
- Low priority one ***must*** run for high priority to make progress
- When might priority lead to starvation or “live lock” ?

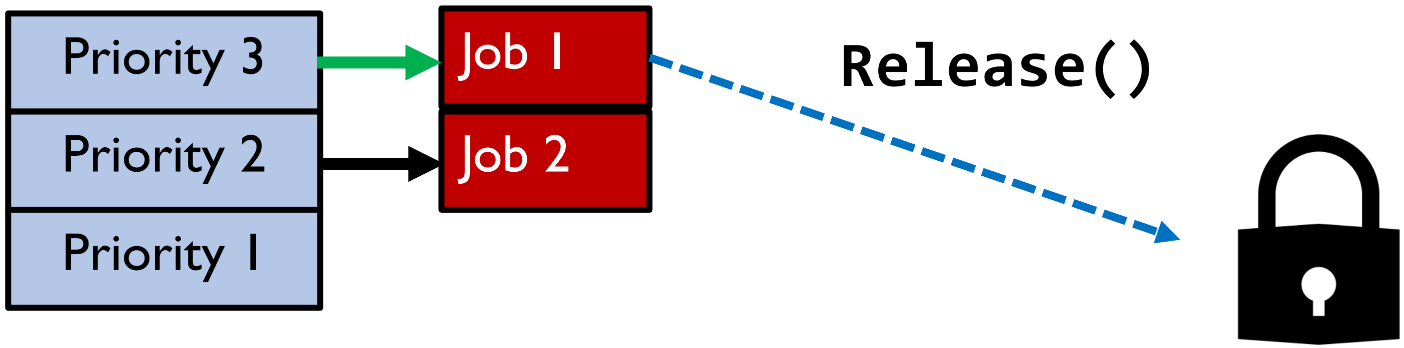
High Priority

```
while (try_lock) {
…
}
```

Low Priority

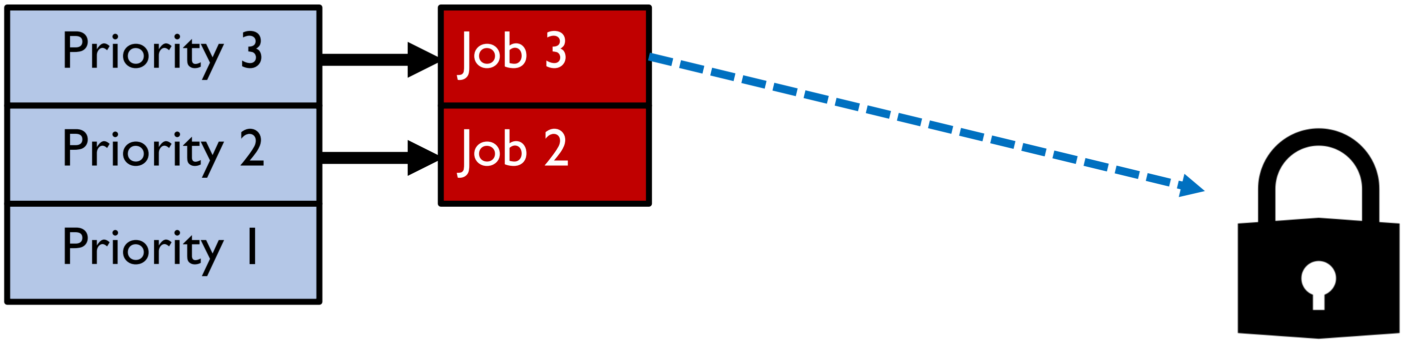
```
lock.acquire(…)
…
lock.release(…)
```

One Solution: Priority Donation



`Job 3` temporarily grants `Job 1` its "highest priority" to run on its behalf

Priority and Locks



Job 1 completes critical section and releases lock
Job 3 acquires lock, runs again
How does scheduler know?

Break

Logistics

- Reminder: submit anonymous feedback on the class at http://bit.ly/cs162fa19anon

Scheduling Wisdom

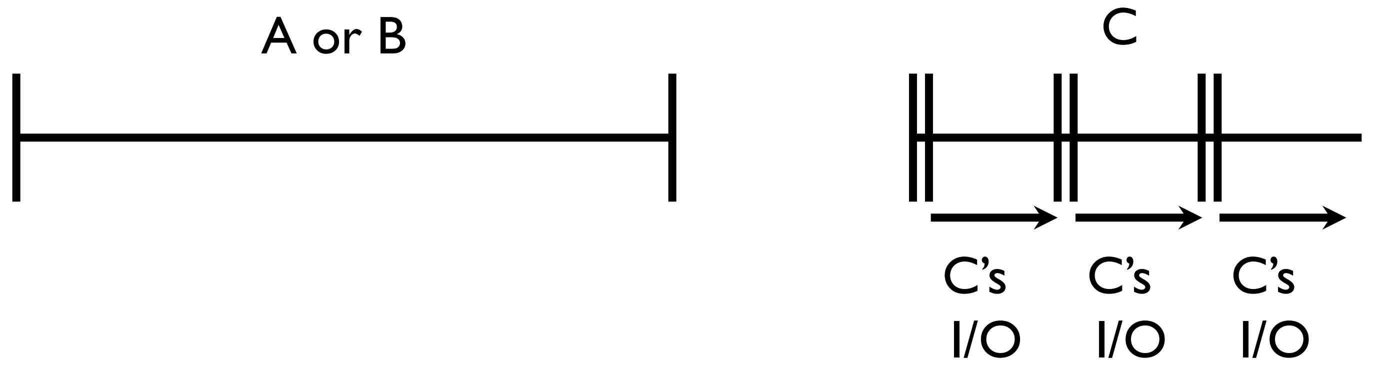
- Modern schedulers use knowledge about program to make better scheduling decisions
- Provided by the user (servers vs background)
- Estimate future based on the past

What if we know how much time each process needs in advance?

- Key Idea: remove convoy effect
  - Short jobs always stay ahead of long ones
- Non-preemptive: **Shortest Job First**
  - Like FCFS if we always chose the best possible ordering
- Preemptive Version: **Shortest Remaining Time First**
  - A newly ready process (e.g., just finished an I/O operation) with shorter time replaces the current one

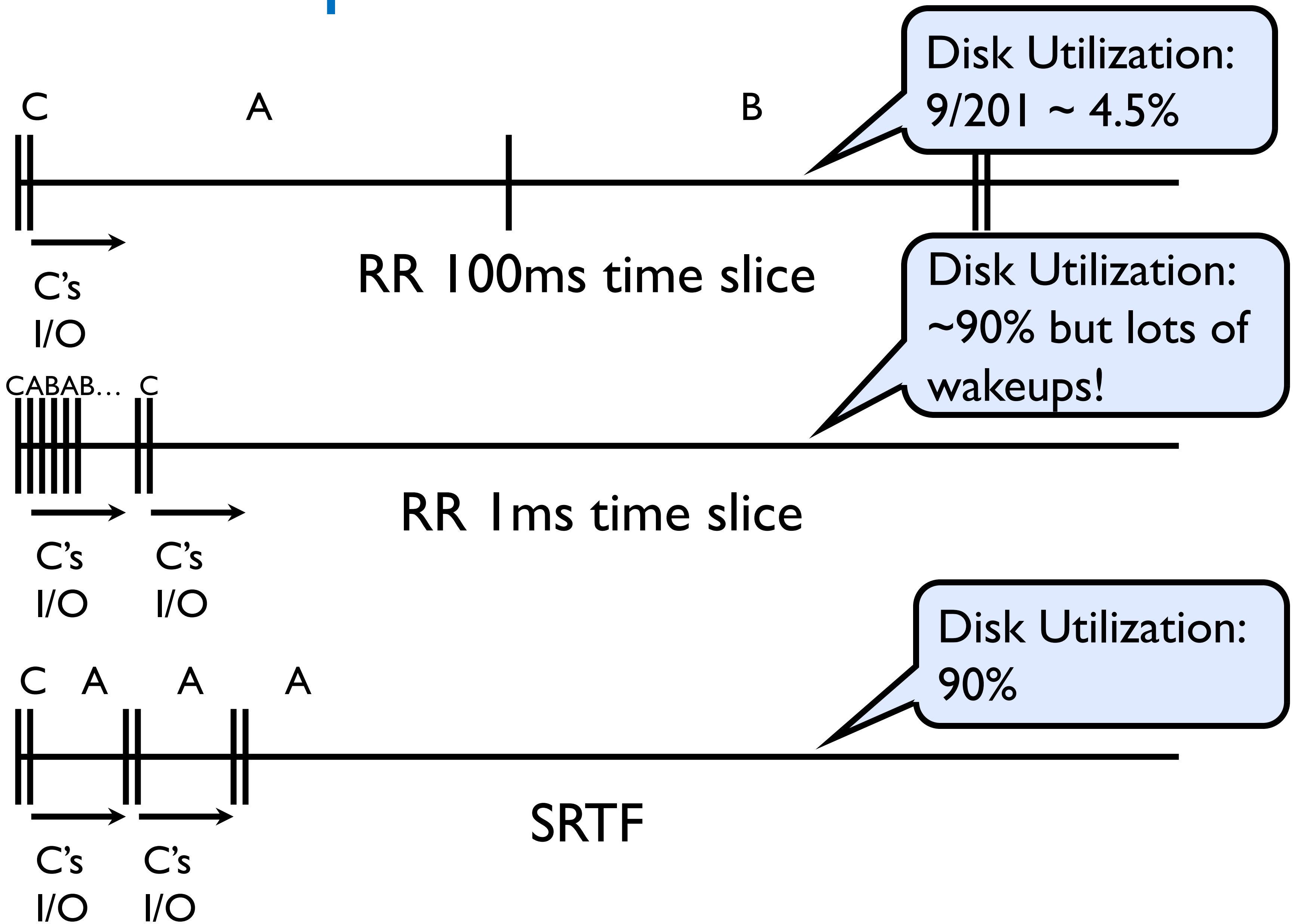
SRTF Example

- Three jobs in system
  - *A* and *B* are CPU calculations that take a week to run
  - *C*: Continuous loop of 1ms CPU time, 9ms of I/O time



- FCFS? *A* or *B* starve *C*
  - I/O throughput problem: lose opportunity to do work for *C* while CPU runs *A* or *B*

SRTF Example



Shortest Job/Remaining Time First

- **Provably Optimal** with respect to *Response Time*

- But Starvation is possible
  - What if new short jobs keep arriving?

Shortest Job/Remaining Time First

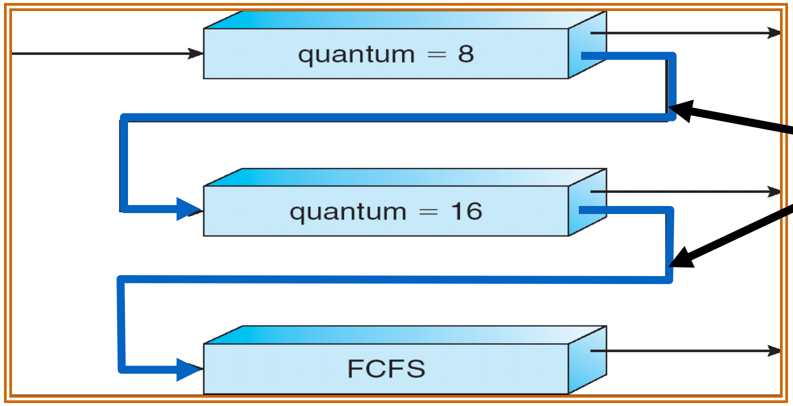
- **How do we know the time a job/process will take?**
  - Usually, we don't

- Ask the users?
  - They can try to estimate…
  - Or guess, or game the system, …

Observing Process Behavior

- Consider jobs scheduled by Round Robin
- Process exhausts quantum, has to be preempted
  - Consuming all of the CPU time it can, "CPU-Bound"
- Process blocks on I/O before quantum exhausted
  - "I/O-Bound"
- Past behavior is good indicator of future behavior
  - I/O-bound now, likely to be I/O-bound later

Multi-Level Feedback Scheduling

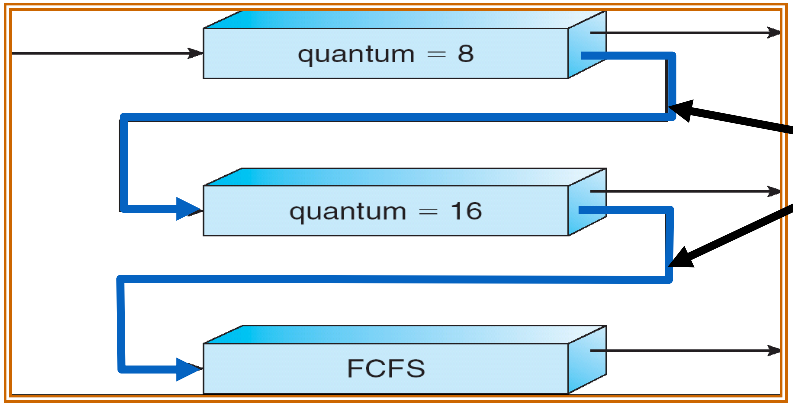


Long-Running Compute Tasks Demoted to Low Priority

- Multiple queues, each of different priority
  - Round Robin within each queue
  - Different quantum length for each queue
- Favor I/O-bound jobs for interactivity
  - Get click or kick off I/O transfer
- Low overhead for CPU bound

Project 2.3

Multi-Level Feedback Scheduling



Long-Running Compute Tasks Demoted to Low Priority

- Intuition: Priority Level proportional to burst length
- Job Exceeds Quantum: Drop to lower queue
- Job Doesn't Exceed Quantum: Raise to higher queue

Multi-Level Feedback Scheduling

- Approximates Shortest Remaining Time First
  - CPU-bound have lowest priority (run last)
  - I/O-bound (short CPU bursts) have highest priority (run first)
- Low overhead
  - Easy to update priority of a job
  - Easy to find next ready task to run
- Can a process cheat?
  - Yes, add meaningless I/O operations (but has a cost)

Multi-Level Feedback Scheduling

- What about starvation?
- Long-running jobs fall into low priority, may never get the CPU
  - Time-slice among the queues (e.g., 70% to highest priority, 20% to middle, 10% to low)
  - Artificially boost priority of a starved job

- These solutions **improve fairness** but **hurt response time**

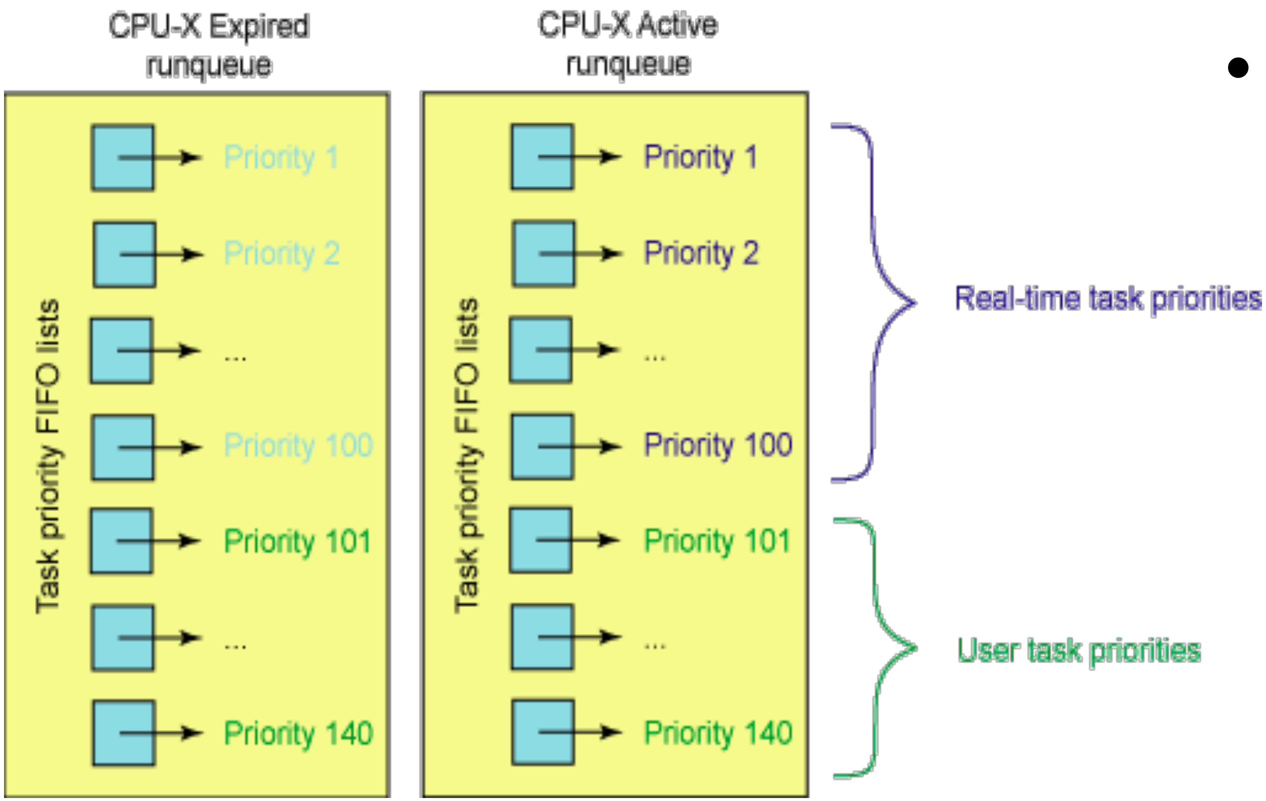
Do you need arrays of queues?

- How might you leverage priority-based scheduling mechanism (and its data structure)?

Linux O(1) Scheduler

- MLFQ-Like Scheduler with 140 Priority Levels
  - 40 for user tasks, 100 "realtime" tasks
  - All algorithms O(1) complexity – low overhead
- *Active* and *expired* queues at each priority
  - Once active is empty, swap them (pointers)
  - **Round Robin** within each queue (varying quanta)

Linux O(1) Scheduler



- Lots of ad-hoc heuristics
  - Try to boost priority of I/O-bound tasks
  - Try to boost priority of starved tasks

Classification

# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, HP-UX, Win NT to 8, Solaris, OS X, Android, iOS

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space

So does the OS schedule processes or threads?

- We've been talking about processes assuming the "old model" -> one thread per process
  - And many textbooks say this as well
- Usually it's really: **threads** (e.g., in Linux)
- More on some of these issues later

- One point to notice: switching threads vs. switching processes incurs different costs:
  - Switch threads: Save/restore registers
  - Switch processes: Change active address space too!
    - Expensive
    - Disrupts caching

User-level threads?

- Can multiple threads be implemented entirely at user level?
- Most other aspects of system virtualize.

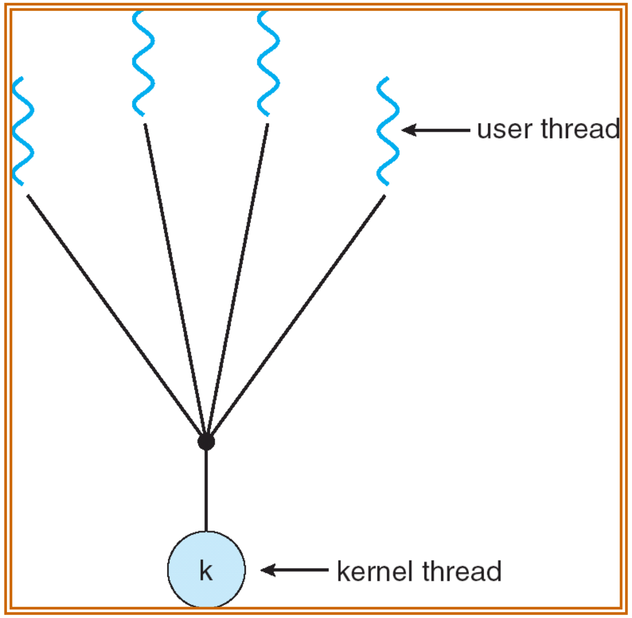
Kernel-Supported Threads

- Threads run and block (e.g., on I/O) independently
- One process may have multiple threads waiting on different things
- Two mode switches for every context switch (expensive)
- Create threads with syscalls

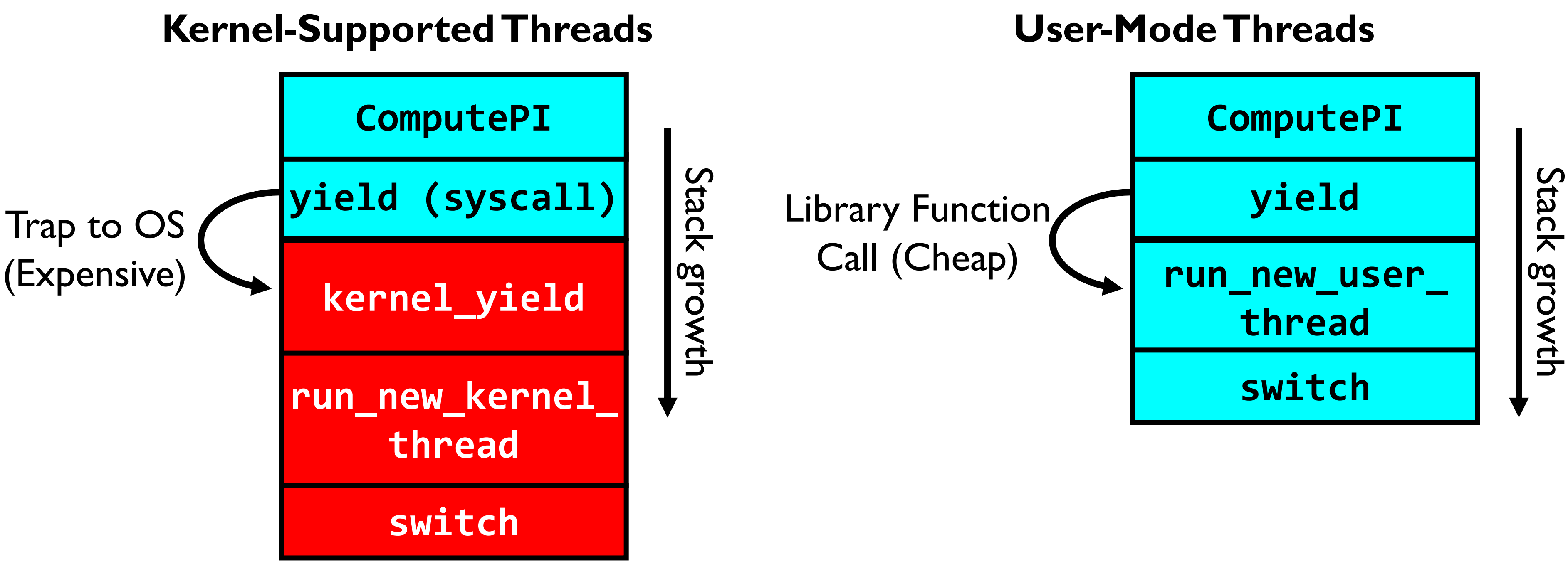
- Alternative: multiplex several streams of execution (at user level) on top of a single OS thread
  - E.g., Java, Go, … (and many many user-level threads libraries before it)

User-Mode Threads

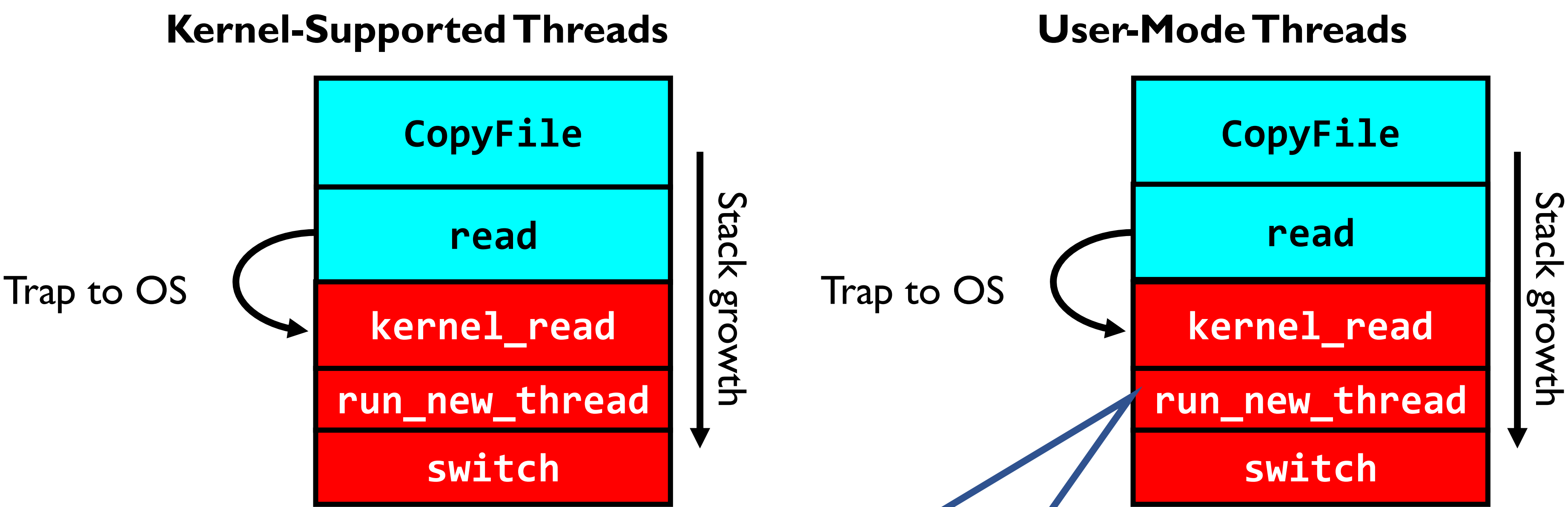
- User program contains its own scheduler
- Several user threads per kernel thd.
- User threads may be scheduled non-preemptively
  - Only switch on `yield`
- Context switches cheaper
  - Copy registers and jump (`switch` in userspace)



Thread Yield



Thread I/O



- Selects a new *kernel thread* to run
- Bypassing user-level scheduler

User-Mode Threads: Problems

- One user-level thread blocks on I/O: they all do
  - Kernel cannot adjust scheduling among threads it doesn't know about
- Multiple Cores?
- Can't completely avoid blocking (syscalls, page fault)
- One Solution: *Scheduler Activations*
  - Have kernel inform user-level scheduler when a thread blocks
- Evolving the contract between OS and application.

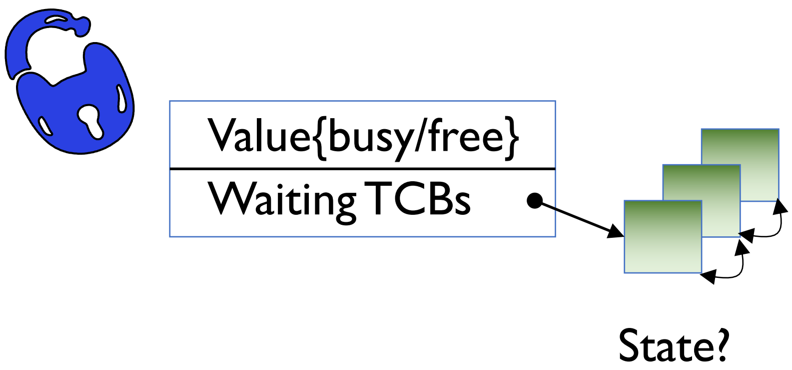
Going back …

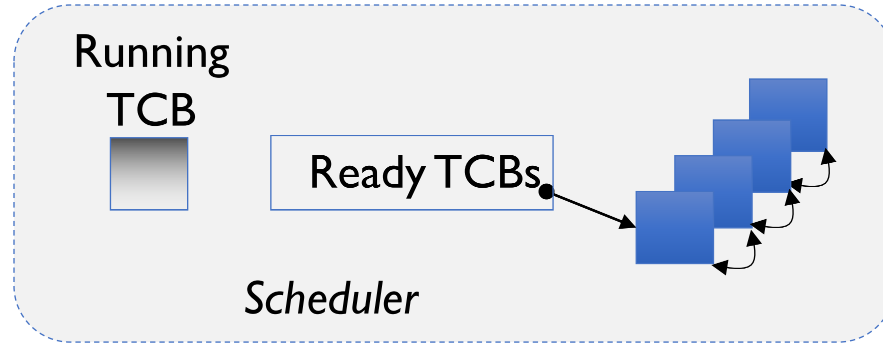
- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

Recall: Basic Lock Implementation






```
Acquire(*lock) {
   disable interrupts;
   if (lock->value == BUSY) {
      put thread on lock’s wait_Q
      “i.e, Go to sleep”
       allow a ready thread to run
   } else {
      lock->value = BUSY;
   }
   enable interrupts;
}
```

```
Release(*lock) {
   disable interrupts;
   if (any TCB on lock wait_Q) {
      “i.e., lock busy”;
      take thread off wait queue
      Place on ready queue;
   } else {
      lock->value = FREE;
   }
   enable interrupts;
}
```

Reenabling Interrupts When Waiting

enable interrupts

enable interrupts

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
} else {
    value = BUSY;
  }
  enable interrupts;
}
```

- Before on the queue?
  - Release might not wake up this thread!
- After putting the thread on the queue?
  - Gets woken up, but immediately switches away

Reenabling Interrupts When Waiting

enable interrupts →

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
} else {
    value = BUSY;
  }
  enable interrupts;
}
```

- Best solution: after the current thread suspends
- How?
  - `run_new_thread()` should do it!
  - Part of returning from `switch()`

How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

<u>Thread A</u>		<u>Thread B</u>

`.`
`.`
`disable ints`
`sleep`		

	context switch →	`sleep return`
		`enable ints`
		`.`
		`.`
		`.`
		`disable int`
		`sleep`

`sleep return`
`enable ints`
`.`
`.`
	← context switch	

Impacts of Scheduling on …

- Lot’s of attention to algorithmic complexity of operations on the scheduling data structure
  - These queues don’t get that long. Otw, buy more hardware
- Interactions of scheduling with memory hierarchy
  - Locality is fundamentally at odds with fairness
  - “Cache / VM / File buffer *affinity*”
- Interactions of scheduling with multiple processors
  - Processor / Core affinity is really about caches
- Memory performance (locality) is critical

Summary

- **First-Come First-Served**: Simple, vulnerable to convoy effect
- **Round-Robin**: Fixed CPU time quantum, cycle between ready threads
- **Priority:** Respect differences in importance
- **Shortest Job/Remaining Time First:** Optimal for average response time, but unrealistic
- **Multi-Level Feedback Queue:** Use past behavior to approximate SRTF and mitigate overhead

System Design …

- Sophisticated policies (often with deep theoretical basis) boil down into simple manipulation of data structures.
- And understanding multi-dimensional interactions

- We'll return to advanced scheduling (with randomness) later in the term