

CS 162: Operating Systems and System Programming

Lecture 11: Deadlock, Scheduling, & Synchronization

October 3, 2019

Instructor: David E. Culler & Sam Kumar

<https://cs162.eecs.berkeley.edu>

Read: A&D 6.5

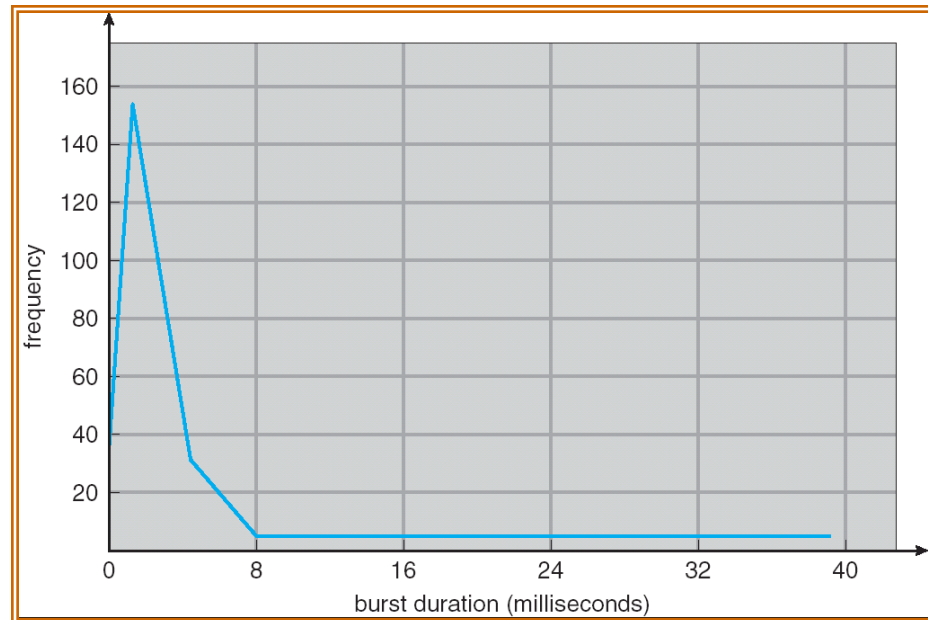
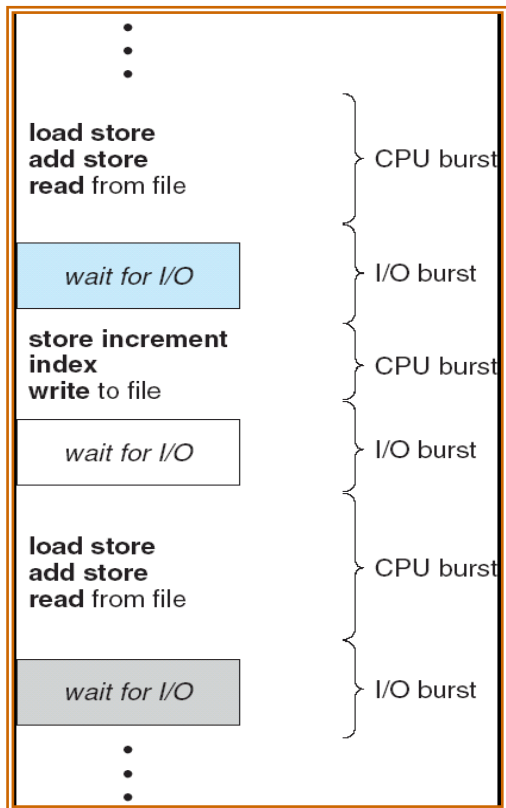
Project 1 Code due TOMORROW!

Midterm Exam next week Thursday

Outline for Today

- (Quickly) Recap and Finish Scheduling
- Deadlock
- Language Support for Concurrency

Recall: 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

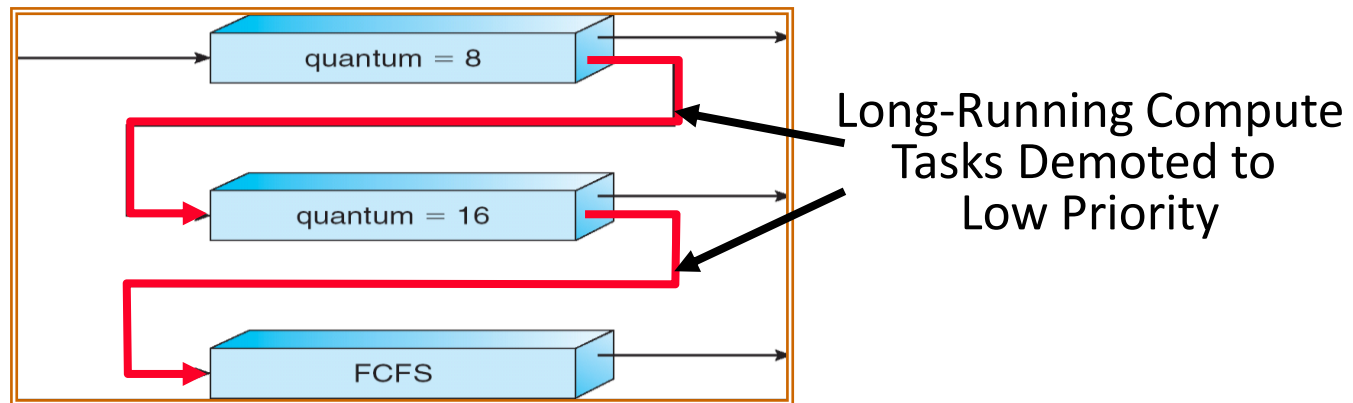
Recall: Evaluating Schedulers

- **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 conflict with best avg. throughput, resp. time

Recall: What if we knew the future?

- Key Idea: remove convoy effect
 - Short jobs always stay ahead of long ones
- Non-preemptive: **Shortest Job First**
 - Like FCFS where 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

Recall: Multi-Level Feedback Scheduling



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

Recall: Linux O(1) Scheduler

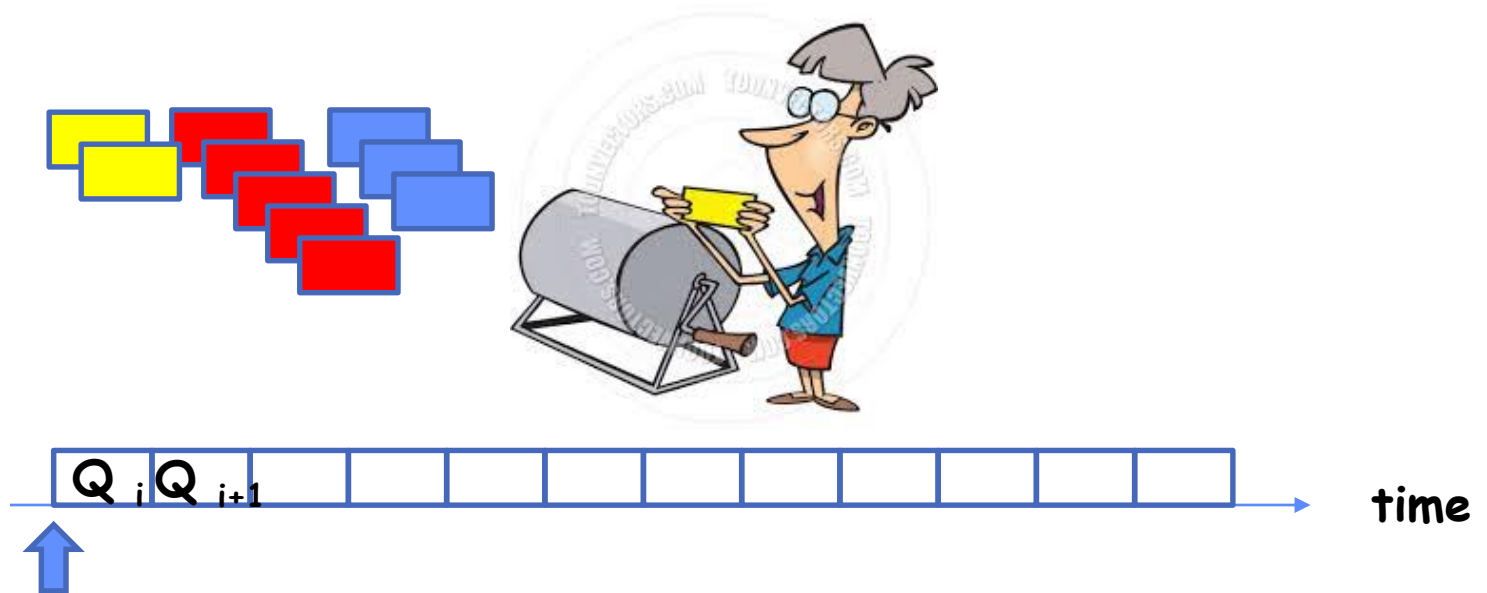


- MLFQ-Like Scheduler with 140 Priority Levels
 - 40 for user tasks, 100 "realtime" tasks
 - All algorithms $O(1)$ complexity – low overhead
- *Active* and *expired* queues at each priority
 - Once active is empty, swap them (pointers)
 - **Round Robin** within each queue (varying quanta)

Recall: Changing landscape of scheduling

- Priority-based scheduling rooted in “time-sharing”
 - Allocating precious, limited resources across a diverse workload
 - » CPU bound, vs interactive, vs I/O bound
- 80’s brought about personal computers, workstations, and servers on networks
 - Different machines of different types for different purposes
 - Shift to fairness and avoiding extremes (starvation)
- 90’s emergence of the web, rise of internet-based services, the data-center-is-the-computer
 - Server consolidation, massive clustered services, huge flashcrowds
 - It’s about predictability, 95th percentile performance guarantees

Recall: Lottery Scheduling



- Given a set of jobs (the mix), provide each with a proportional share of a resource
 - e.g., 50% of the CPU for **Job A**, 30% for **Job B**, and 20% for **Job C**
- Idea: Give out tickets according to the portion each should receive,
- Every quanta (tick) draw one at random, schedule that job (thread) to run

Recall: Stride Scheduling

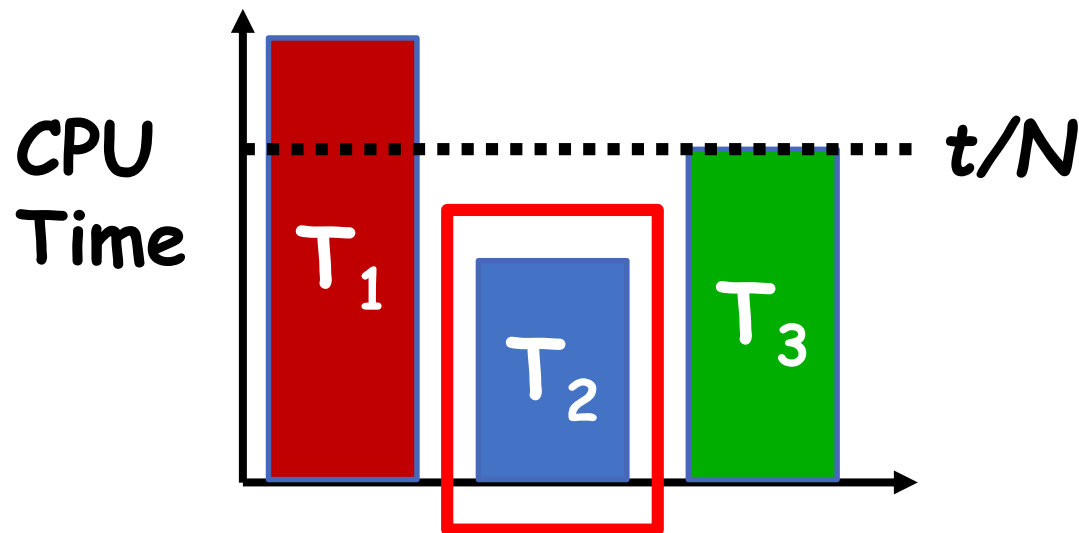
- Achieve proportional share scheduling without resorting to randomness, and overcome the “law of small numbers” problem.
- “Stride” of each job is $\frac{big\#W}{N_i}$
 - The larger your share of tickets, the smaller your stride
 - Ex: $W = 10,000$, $A=100$ tickets, $B=50$, $C=250$
 - A stride: 100, B: 200, C: 40
- Each job as a “pass” counter
- Scheduler: pick job with lowest *pass*, runs it, add its *stride* to its *pass*
- Low-stride jobs (lots of tickets) run more often
 - Job with twice the tickets gets to run twice as often
- Some messiness of counter wrap-around, new jobs, ...

Recall: Linux Completely Fair Scheduler

- Can't do this with real hardware
 - Still need to give out full CPU in time slices
- Instead: track CPU time given to a thread so far

Scheduling Decision:

- "Repair" illusion of complete fairness
- Choose thread with minimum CPU time

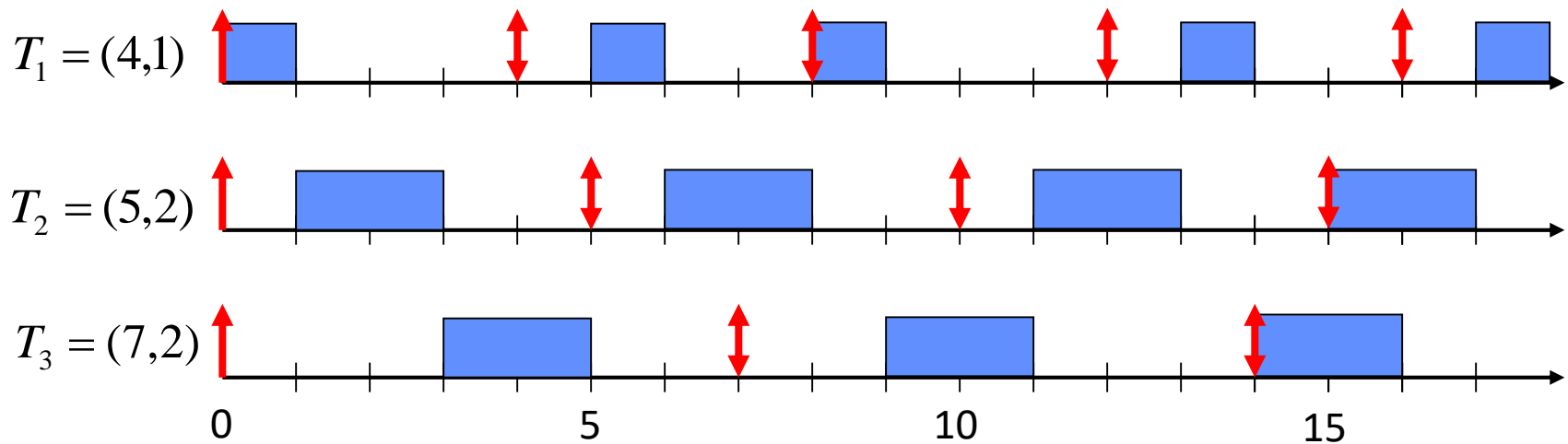


Recall: Real-Time Scheduling

- Goal: **Guaranteed Performance**
 - Meet **deadlines** even if it means being unfair or slow
 - Limit how bad the **worst case** is
- Hard real-time:
 - Meet **all deadlines** (if possible)
 - Ideally: determine in advance if this is possible

Recall: Earliest Deadline First (EDF)

- Priority scheduling with preemption
- Priority proportional to time until deadline
- Example with **periodic tasks**:



Recall: Choosing the Right Scheduler

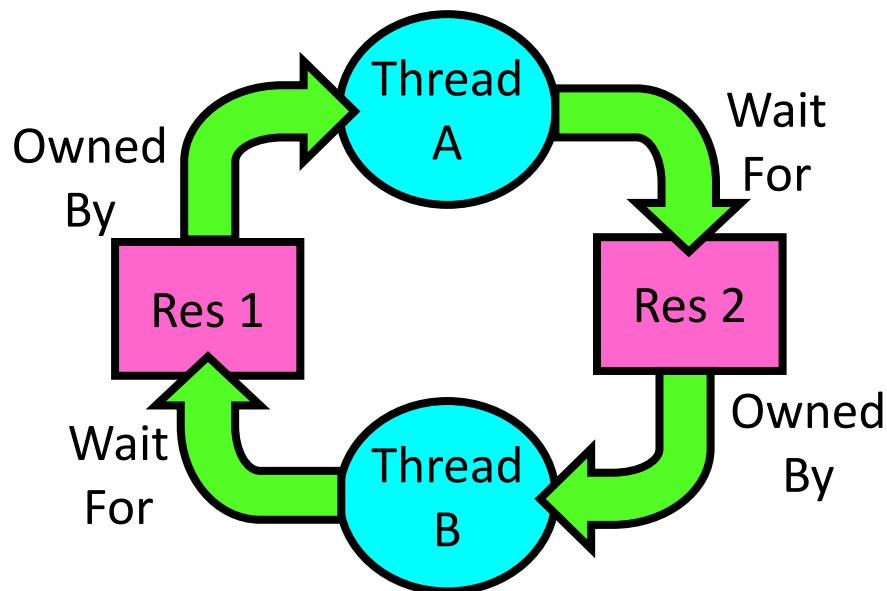
I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness - Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

Ensuring Progress

- Schedulers try to schedule jobs efficiently
- Assume that the threads make progress.
- If they are all just waiting for each other or spinning in loops, the scheduler cannot help much.
- Let's see what sorts of problems we might fall into and how to avoid them

Types of Scheduling Problems

- Starvation – thread fails to make progress for an indefinite period of time
- Deadlock – starvation due to a *cycle of waiting* among a set of threads
 - each thread waits for some other thread in the cycle to take some action

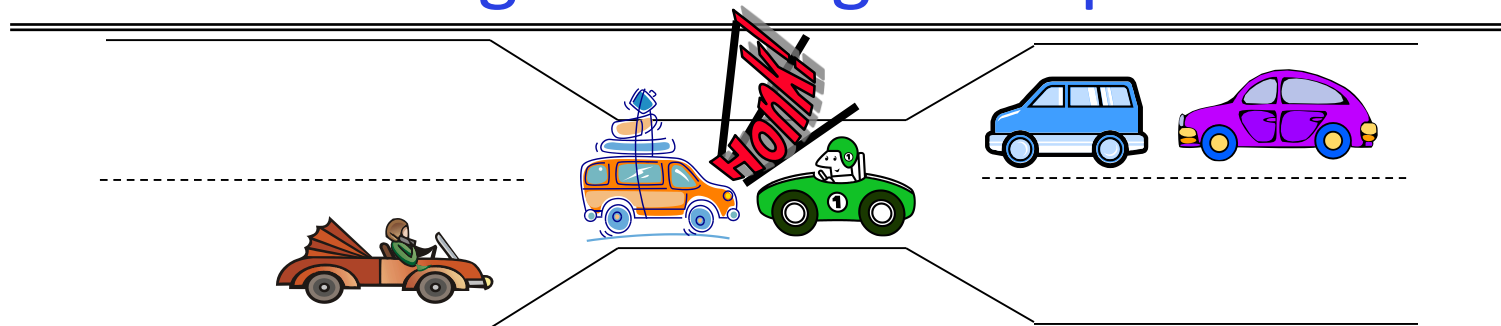


Example: Single-Lane Bridge Crossing



CA 140 to Yosemite National Park

Bridge Crossing Example



- Each segment of road can be viewed as a resource
 - Car must own the segment under them
 - Must acquire segment that they are moving into
- **Deadlock:** Two cars in opposite directions meet in middle
- Resolving deadlock: “Preempt” road segment, force one car (or several cars) to back up
- Prevent deadlock: make sure cars facing opposite directions don’t enter the bridge simultaneously
- **Starvation** (not deadlock): Eastbound traffic doesn’t stop for westbound traffic

Deadlock with Locks

Thread A

`x.Acquire();`

`y.Acquire();`

...

`y.Release();`

`x.Release();`

Thread B

`y.Acquire();`

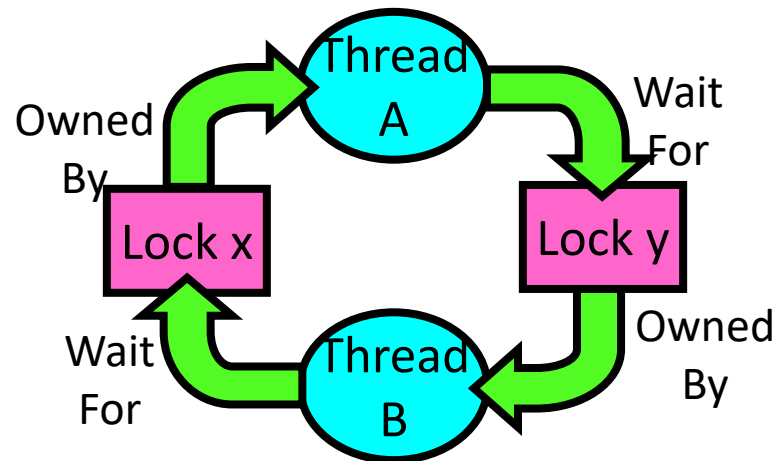
`x.Acquire();`

...

`x.Release();`

`y.Release();`

Nondeterministic Deadlock



Deadlock with Locks: Unlucky Case

Thread A

`x.Acquire();`

`y.Acquire();` *<stalled>*
<unreachable>

...

`y.Release();`

`x.Release();`

Thread B

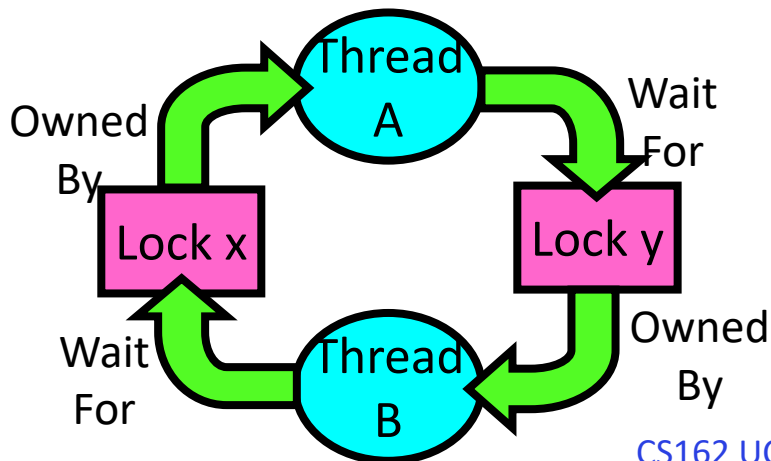
`y.Acquire();`

`x.Acquire();` *<stalled>*
<unreachable>

...

`x.Release();`

`y.Release();`



Deadlock with Locks: “Lucky” Case

Thread A

```
x.Acquire();  
y.Acquire();  
...  
y.Release();  
x.Release();
```

Thread B

```
y.Acquire();  
  
x.Acquire();  
...  
x.Release();  
y.Release();
```

Sometimes schedule won't trigger deadlock

Other Types of Deadlock

- Threads often block waiting for resources
 - Locks
 - Terminals
 - Printers
 - CD drives
 - Memory
- Threads often block waiting for other threads
 - Pipes
 - Sockets
- You can deadlock on any of these!

Deadlock with Space

Thread A

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

Thread B

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

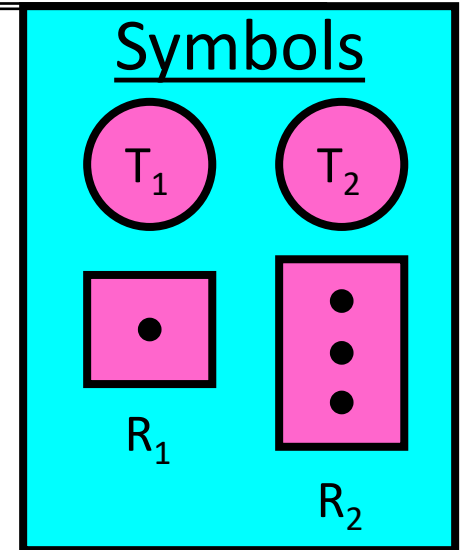
If only 2 MB of space, we get same deadlock situation

HOW TO DETECT DEADLOCK?

Resource-Allocation Graph

- System Model

- A set of Threads T_1, T_2, \dots, T_n
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each thread utilizes a resource as follows:
 - » Request () / Use () / Release ()

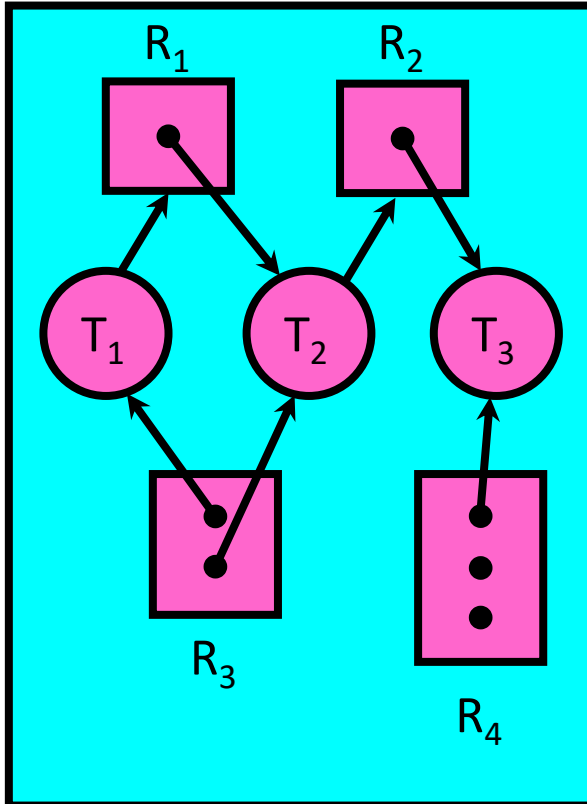


- Resource-Allocation Graph:

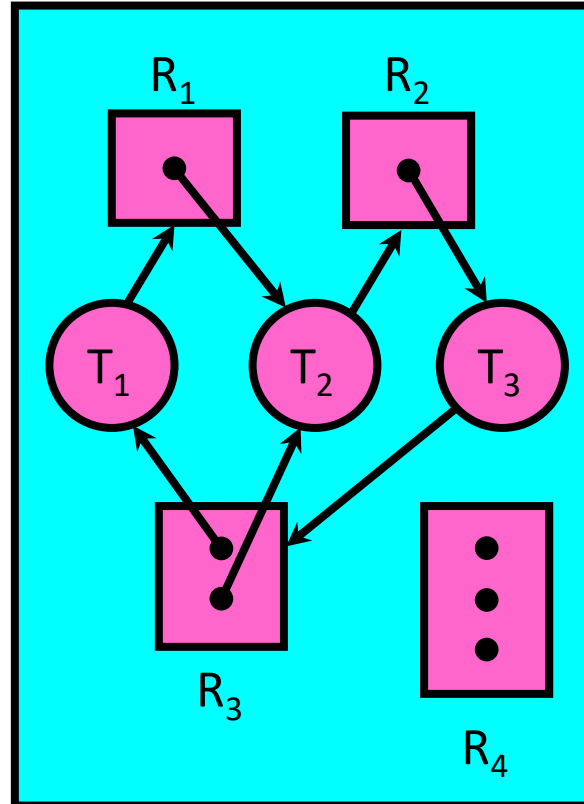
- V is partitioned into two types:
 - » $T = \{T_1, T_2, \dots, T_n\}$, the set threads in the system.
 - » $R = \{R_1, R_2, \dots, R_m\}$, the set of resource types in system
- request edge – directed edge $T_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow T_i$

Resource-Allocation Graph Examples

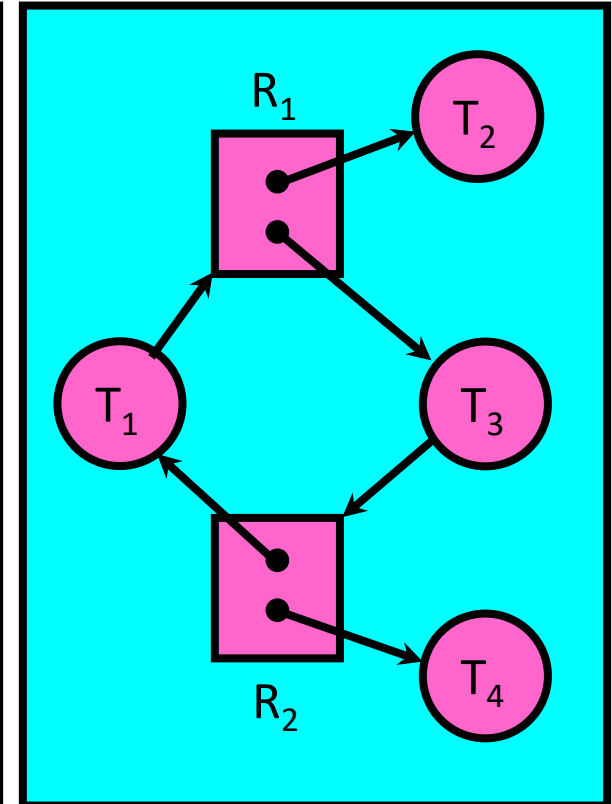
- Model:
 - request edge – directed edge $T_i \rightarrow R_j$
 - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource
Allocation Graph



Allocation Graph
With Deadlock



Allocation Graph
With Cycle, but
No Deadlock

Deadlock Detection Algorithm

- Only one of each type of resource \Rightarrow look for loops
- More General Deadlock Detection Algorithm

- Let $[X]$ represent an m-ary vector of non-negative integers (quantities of resources of each type):

$[FreeResources]$: Current free resources each type

$[Request_x]$: Current requests from thread X

$[Alloc_x]$: Current resources held by thread X

- See if tasks can eventually terminate on their own

$[Avail] = [FreeResources]$

Add all nodes to UNFINISHED

do {

 done = true

 Foreach node in UNFINISHED {

 if ($[Request_{node}] \leq [Avail]$) {

 remove node from UNFINISHED

$[Avail] = [Avail] + [Alloc_{node}]$

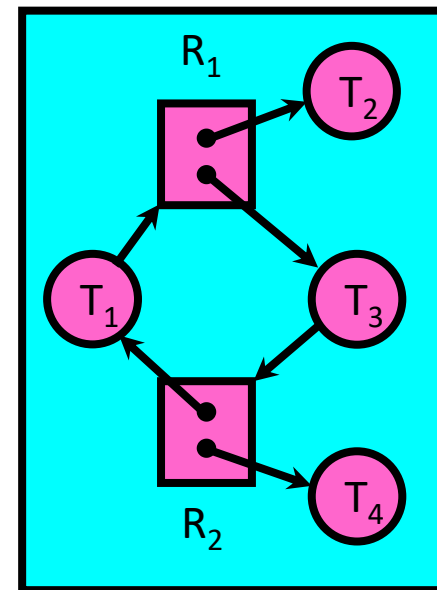
 done = false

 }

 }

 } until(done)

- Nodes left in UNFINISHED \Rightarrow deadlocked



How should a system deal with deadlock?

- Three different approaches:
 1. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
 2. Deadlock prevention: write your code in a way that it isn't prone to deadlock
 3. Deadlock recovery: let deadlock happen, and then figure out how to recover from it
- Modern operating systems:
 - Make sure the *system* isn't involved in any deadlock
 - Ignore deadlock in applications
 - » “Ostrich Algorithm”

Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in deadlock
 - If not, it grants the resource right away
 - If so, it waits for other threads to release resources

THIS DOES NOT WORK!!!!

- Example:

Thread A

`x.Acquire();`

Blocks... `y.Acquire();`

...

`y.Release();`

`x.Release();`

Thread B

`y.Acquire();`

`x.Acquire();`

...

`x.Release();`

`y.Release();`

Wait...

But it's too late...

Deadlock Avoidance: Three States

- Safe state
 - System can delay resource acquisition to prevent deadlock
- Unsafe state
 - No deadlock yet...
 - But threads can request resources in a pattern that ***unavoidably*** leads to deadlock
- Deadlocked state
 - There exists a deadlock in the system
 - Also considered “unsafe”

Deadlock avoidance:
prevent system from
reaching an *unsafe* state

Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in ~~deadlock~~ **an unsafe state**
 - If not, it grants the resource right away
 - If so, it waits for other threads to release resources
- Example:

Thread A

```
x.Acquire();  
y.Acquire();  
...  
y.Release();  
x.Release();
```

Thread B

```
y.Acquire();  
x.Acquire();  
...  
x.Release();  
y.Release();
```

Wait until
Thread A
releases the
mutex

Banker's Algorithm for Preventing Deadlock

- Toward right idea:
 - State maximum (max) resource needs in advance
 - Allow particular thread to proceed if:
 $(\text{available resources} - \text{\#requested}) \geq \text{max remaining that might be needed by any thread}$
- Banker's algorithm (less conservative):
 - Allocate resources dynamically
 - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
 - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
 $([\text{Max}_{\text{node}}] - [\text{Alloc}_{\text{node}}] \leq [\text{Avail}])$ for $([\text{Request}_{\text{node}}] \leq [\text{Avail}])$
Grant request if result is deadlock free (conservative!)



Banker's Algorithm for Preventing Deadlock

- ```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
 done = true
 Foreach node in UNFINISHED {
 if ([Requestnode] <= [Avail]) {
 remove node from UNFINISHED
 [Avail] = [Avail] + [Allocnode]
 done = false
 }
 }
} until(done)
```



- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting  $([Max_{node}] - [Alloc_{node}] \leq [Avail])$  for  $([Request_{node}] \leq [Avail])$   
Grant request if result is deadlock free (conservative!)

# Banker's Algorithm for Preventing Deadlock

- ```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ( $[Max_{node}] - [Alloc_{node}] \leq [Avail]$ ) {
            remove node from UNFINISHED
             $[Avail] = [Avail] + [Alloc_{node}]$ 
            done = false
        }
    }
} until(done)
```



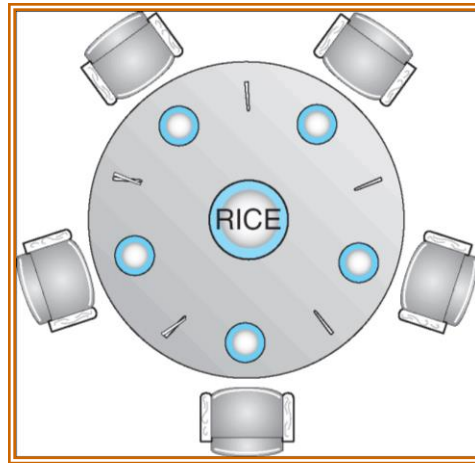
- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting $([Max_{node}] - [Alloc_{node}] \leq [Avail])$ for $([Request_{node}] \leq [Avail])$
Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Preventing Deadlock

- Toward right idea:
 - State maximum resource needs in advance
 - Allow particular thread to proceed if:
(available resources - #requested) \geq max remaining that might be needed by any thread
- Banker's algorithm (less conservative):
 - Allocate resources dynamically
 - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
 - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
 $([Max_{node}] - [Alloc_{node}] \leq [Avail])$ for $([Request_{node}] \leq [Avail])$
Grant request if result is deadlock free (conservative!)
 - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \dots, T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
 - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources



Banker's Algorithm Example



- Banker's algorithm with dining philosophers
 - “Safe” (won't cause deadlock) if when try to grab chopstick either:
 - » Not last chopstick
 - » Is last chopstick but someone will have two afterwards
 - What if k-handed lawyers? Don't allow if:
 - » It's the last one, no one would have k
 - » It's 2nd to last, and no one would have k-1
 - » It's 3rd to last, and no one would have k-2
 - » ...



Deadlock Prevention

- Structure code in a way that it isn't prone to deadlock
- First: What must be true about our code for deadlock to happen?

Four requirements for Deadlock

- **Mutual exclusion**
 - Only one thread at a time can use a resource.
- **Hold and wait**
 - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
 - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
 - There exists a set $\{T_1, \dots, T_n\}$ of waiting threads
 - » T_1 is waiting for a resource that is held by T_2
 - » T_2 is waiting for a resource that is held by T_3
 - » ...
 - » T_n is waiting for a resource that is held by T_1
- **To prevent deadlock, make sure at least one of these conditions does not hold**

Deadlock Prevention (1/2)

- Remove “Mutual Exclusion”
 - Infinite resources
 - » Example: Virtual Memory
 - Restructure program to avoid sharing

(Virtually) Infinite Resources

Thread A

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

Thread B

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

With virtual memory we have “infinite” space so everything will just succeed.

Deadlock Prevention (1/2)

- Remove “Mutual Exclusion”
 - Infinite resources
 - » Example: Virtual Memory
 - Restructure program to avoid sharing
- Remove “Hold-and-Wait”
 - Back off and retry
 - » Removes deadlock but could still lead to starvation
 - Request all resources up front
 - » Reduces concurrency (parallelism?)
 - » Example: Dining philosophers grab both chopsticks atomically

Request Resources Atomically (1)

Thread A

```
x.Acquire();  
y.Acquire();  
...  
y.Release();  
x.Release();
```

Thread B

```
y.Acquire();  
x.Acquire();  
...  
x.Release();  
y.Release();
```

Consider instead:

Thread A

```
Acquire_both(x, y);  
...  
y.Release();  
x.Release();
```

Thread B

```
Acquire_both(y, x);  
...  
x.Release();  
y.Release();
```

Request Resources Atomically (2)

Or consider this:

Thread A

z.Acquire();

x.Acquire();

y.Acquire();

z.Release();

...

y.Release();

x.Release();

Thread B

z.Acquire();

y.Acquire();

x.Acquire();

z.Release();

...

x.Release();

y.Release();

Deadlock Prevention (2/2)

- Remove “No Preemption”
 - Allow OS to revoke resources it has granted
 - » Example: Pre-emptive scheduling
 - Doesn’t always work with resource semantics
 - » Example: Locks

Pre-empting Resources

Thread A

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

Thread B

AllocateOrWait(1 MB)

AllocateOrWait(1 MB)

Free(1 MB)

Free(1 MB)

With virtual memory we have “infinite” space so everything will just succeed.

Alternative view: we are “pre-empting” memory when paging out to disk, and giving it back when paging back in

Deadlock Prevention (2/2)

- Remove “No Preemption”
 - Allow OS to revoke resources it has granted
 - » Example: Pre-emptive scheduling
 - Doesn’t always work with resource semantics
 - » Example: Locks
- Remove “Circular Wait”
 - Acquire resources in a consistent order

Acquire Resources in Consistent Order

Thread A

```
x.Acquire();  
y.Acquire();  
...  
y.Release();  
x.Release();
```

Thread B

```
y.Acquire();  
x.Acquire();  
...  
x.Release();  
y.Release();
```

Consider instead:

Thread A

```
x.Acquire();  
y.Acquire();  
...  
y.Release();  
x.Release();
```

Thread B

```
x.Acquire();  
y.Acquire();  
...  
x.Release();  
y.Release();
```

**Does it matter in
which order the
locks are released?**

Deadlock Recovery

- Let deadlock happen, and then figure out how to deal with it

What to do when detect deadlock?

- Terminate thread, force it to give up resources
 - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
 - Remove a dining lawyer
 - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
 - Take away resources from thread temporarily
 - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
 - Hit the rewind button on TiVo, pretend last few minutes never happened
 - For bridge example, make one car roll backwards (may require others behind him)
 - Common technique in databases (transactions)
 - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options


Recall: Monitors and Condition Variables

- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
 - Use of Monitors is a programming paradigm
 - Some languages like Java provide monitors in the language
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
 - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
 - Contrast to semaphores: Can't wait inside critical section
- Operations:
 - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
 - `Signal()`: Wake up one waiter, if any
 - `Broadcast()`: Wake up all waiters
- Rule: **Must hold lock when doing condition variable ops!**

Recall: (Mesa) Monitor Pattern

- Monitors represent the logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed to recheck their condition
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
    condvar.wait();
}
unlock
```




Check and/or update
state variables
Wait if necessary

```
lock

condvar.signal();

unlock
```



Check and/or update
state variables

Programming Language Support for Concurrency and Synchronization

- Synchronization operations
- Exceptional conditions

Concurrency and Synchronization in C

- Standard approach: use **pthread**s, protect access to shared data structures
- One pitfall: consistently unlocking a mutex

```
int Rtn() {  
    lock.acquire();  
    ...  
    if (error) {  
        lock.release();  
        return errCode;  
    }  
    ...  
    lock.release();  
    return OK;  
}
```

Concurrency and Synchronization in C

- Harder with more locks

```
void Rtn() {
    lock1.acquire();
    ...
    if (error) {
        lock1.release();
        return;
    }
    ...
    lock2.acquire();
    ...
    if (error) {
        lock2.release();
        lock1.release();
        return;
    }
    ...
    lock2.release();
    lock1.release();
    return;
}
```

- Is goto a solution???

```
void Rtn() {
    lock1.acquire();
    ...
    if (error) {
        goto release_lock1_and_exit;
    }
    ...
    lock2.acquire();
    ...
    if (error) {
        goto release_both_and_exit;
    }
    ...
release_both_and_exit:
    lock2.release();
release_lock1_and_exit:
    lock1.release();
    return;
}
```

C++ Lock Guards

```
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
    std::lock_guard<std::mutex> lock(global_mutex);
    ...
    global_i++;
    // Mutex released when 'lock' goes out of scope
}
```

Python **with** Keyword

- More versatile than we'll show here (can be used to close files, database connections, etc.)

```
lock = threading.Lock()
```

```
...
```

```
with lock: # Automatically calls acquire()
```

```
    some_var += 1
```

```
...
```

```
# release() called however we leave block
```


Java Language Support for Synchronization

- Every Java object has an associated lock:
 - Lock is acquired on entry and released on exit from a **synchronized** method
 - Lock is properly released if exception occurs inside a **synchronized** method
 - Mutex execution of synchronized methods (beware deadlock)

```
class Account {
    private int balance;

    // object constructor
    public Account (int initialBalance) {
        balance = initialBalance;
    }
    public synchronized int getBalance() {
        return balance;
    }
    public synchronized void deposit(int amount) {
        balance += amount;
    }
}
```

Java Support for Synchronization

- Along with a lock, every object has a **single** condition variable associated with it
- To wait inside a synchronized method:
 - **void wait();**
 - **void wait(long timeout);**
- To signal while in a synchronized method:
 - **void notify();**
 - **void notifyAll();**

Go Language Support for Concurrency

- Go was designed with concurrent applications in mind
- Some language aspects we'll talk about today
 - defer keyword
 - Goroutines
 - Channels
- Some language aspects we won't talk about
 - select keyword
 - Contexts

The defer keyword in Go

- Solution: use defer

```
func Rtn() {  
    lock.Lock()  
    ...  
    if error {  
        lock.Unlock()  
        return  
    }  
    ...  
    lock.Unlock()  
    return  
}
```

```
func Rtn() {  
    lock.Lock()  
    defer lock.Unlock()  
    ...  
    if error {  
        return  
    }  
    ...  
    return  
}
```

The defer keyword in Go

- The queue of “deferred” calls is maintained dynamically

```
func Rtn() {  
    lock1.Lock()  
    defer lock1.Lock()  
    ...  
    if condition {  
        lock2.Lock()  
        defer lock2.Unlock()  
    }  
    ...  
    return  
}
```

- lock1 is always unlocked here
- lock2 is unlocked here only if the condition was true earlier



Goroutines in Go

- Goroutines are lightweight, user-level threads
 - Scheduling not preemptive (relies on goroutines to yield)
 - Yield statements inserted by compiler
- Advantages relative to regular threads (e.g., pthreads)
 - More lightweight
 - Faster context-switch time
- Disadvantages
 - Less sophisticated scheduling at the user-level
 - OS is not aware of user-level threads

Channels in Go

- A channel is a bounded buffer
 - Writes block if buffer is full
 - Reads block if buffer is empty
- “Do not communicate by sharing memory; instead, share memory by communicating.”
 - From *Effective Go*
- Go prefers using channels to synchronize goroutines
 - not mutexes and condition variables, as in pthreads

Channels: Analogy to Homework 1

- You used a mutex to synchronize pwords
- You used pipes to synchronize fwords
- A channel is like a pipe between goroutines
- Like pipes:
 - Channels are a bounded buffer abstraction
 - Writer closes channels so the reader knows to stop trying to read
- Unlike pipes:
 - Channels exist in userspace, so you don't have to worry about using file descriptors or different address spaces
 - You can send structured data (e.g., objects), not just bytes

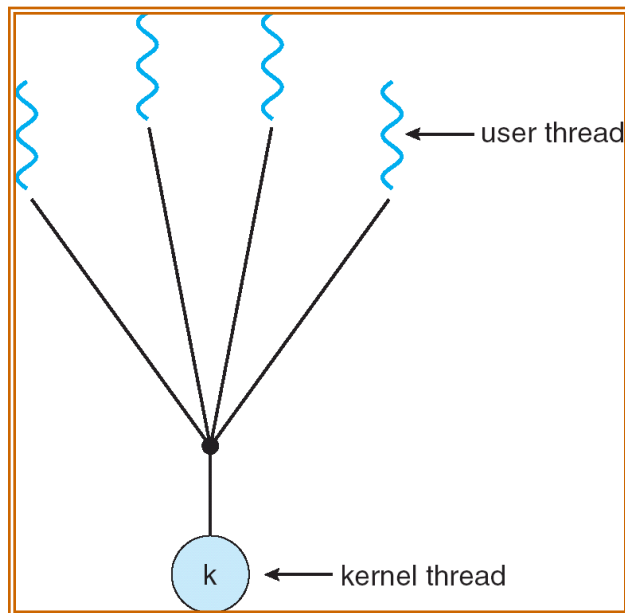
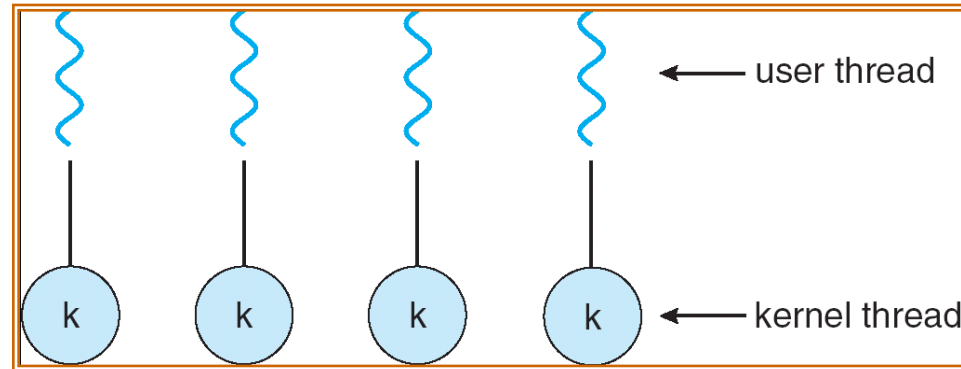
Summary

- From Priorities to Proportional Share: Linux CFS Scheduler
 - Fair fraction of CPU to threads, modulated by priority
- Real-time scheduling - meet deadlines, predictability
 - Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling
- Starvation vs. Deadlock
 - Starvation: thread waits indefinitely
 - Deadlock: circular waiting for resources
- Four conditions for deadlocks
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
- Techniques for addressing Deadlock
 - Detect deadlock and then recover
 - Ensure that system will *never* enter a deadlock
- Threads and Synchronization integral to modern languages

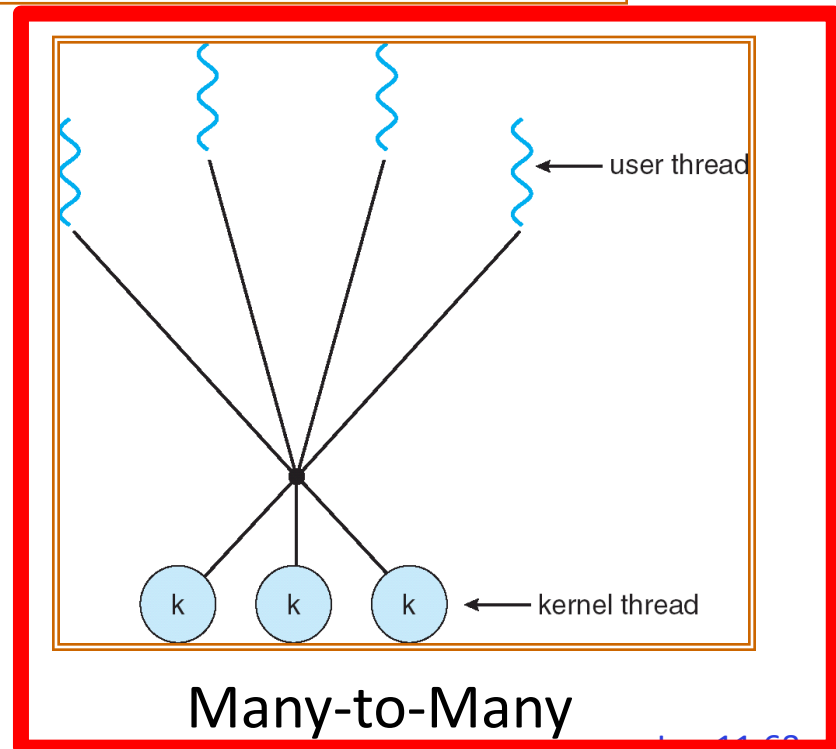
BONUS SLIDES

Remember this Slide?

Simple One-to-One Threading Model



Many-to-One



Many-to-Many

User-Mode Threads: Problems

- One user-level thread blocks on syscall: all user-level threads relying on same kernel thread also block
 - Kernel cannot intelligently schedule threads it doesn't know about
- Multiple Cores?
- No pre-emption: User-level thread must explicitly yield CPU to allow someone else to run

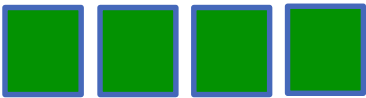
Go User-Level Thread Scheduler

Global Run Queue



Newly
created
goroutines

Local Run Queue



OS Thread
(M)

CPU Core

Local Run Queue

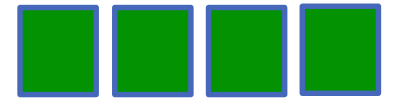


OS Thread
(M)

CPU Core

...

Local Run Queue



OS Thread
(M)

CPU Core

Why this approach?

- 1 OS (kernel-supported) thread per CPU core: allows go program to achieve *parallelism* not just *concurrency*
 - Fewer OS threads? Not utilizing all CPUs
 - More OS threads? No additional benefit
 - » We'll see one exception to this involving syscalls
- Keep goroutine on same OS thread: *affinity*, nice for caching and performance

Cooperative Scheduling

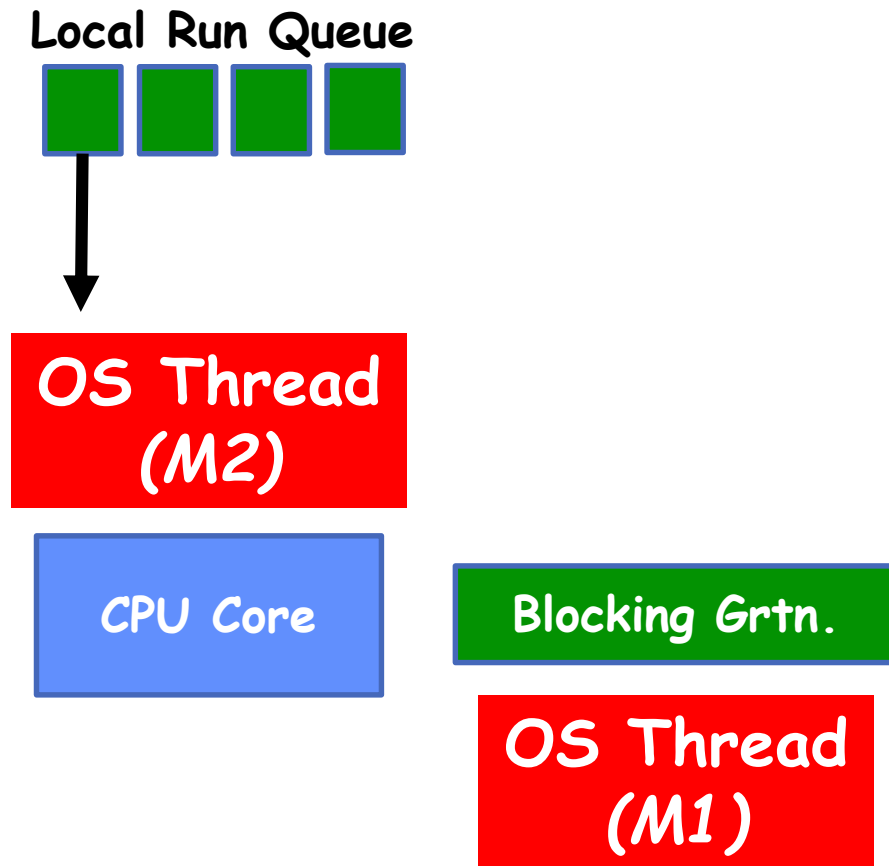
- No pre-emption => goroutines must yield to allow another thread to run
- Programmer does **not** need to do this explicitly
- Go runtime injects yields at safe points in execution
 - Sending/receiving with a channel
 - Acquiring a mutex
 - Making a function call
- But your code can still tie up the scheduler in "tight loops" (Go's developers are working on this...)

Dealing with Syscalls



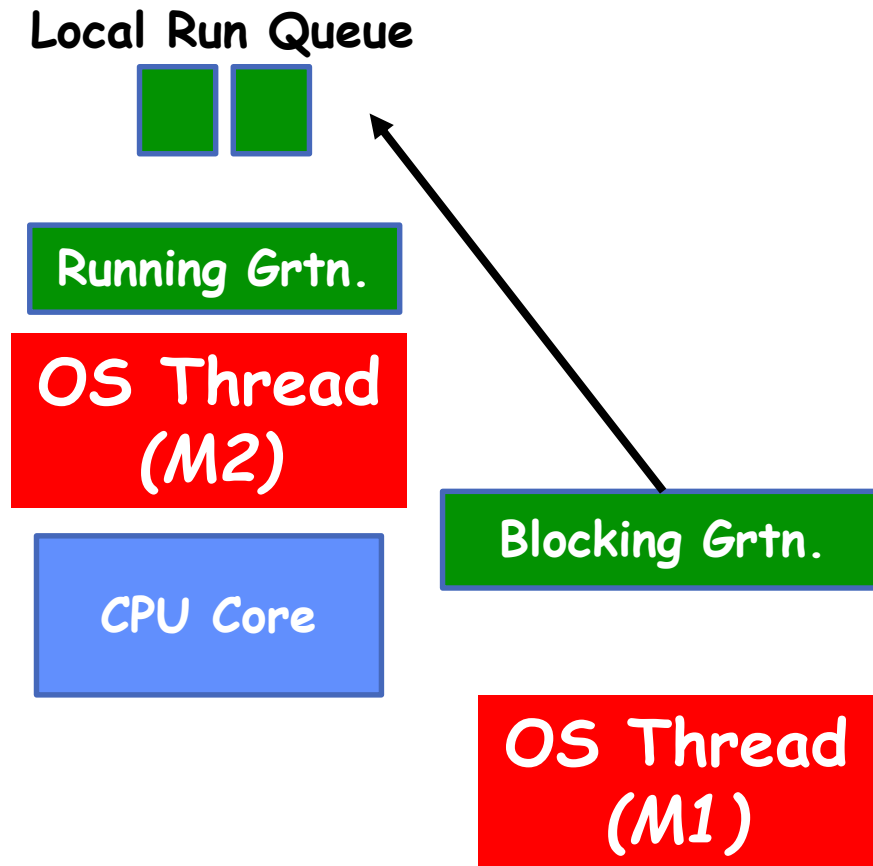
- What if a goroutine wants to make a blocking syscall?
- Example: File I/O

Dealing with Syscalls



- While syscall is blocking, allocate new OS thread (M2)
- M1 is blocked by kernel, M2 lets us continue using CPU

Dealing with Syscalls



- Syscall completes: Put invoking goroutine back on queue
- Keep *M1* around in a spare pool
- Swap it with *M2* upon next syscall, no need to pay thread creation cost