

CS162
Operating Systems and
Systems Programming
Lecture 13

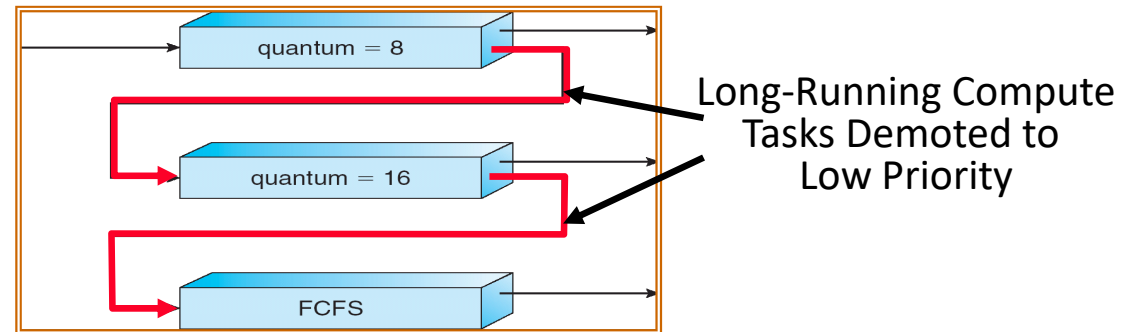
Scheduling 3: Proportional Share Scheduling,
Realtime Scheduling, Deadlock

March 5th, 2026

Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

Recall: Multi-Level Feedback Scheduling



- Another method for exploiting past behavior (first use in CTSS)
 - Multiple queues, each with different priority
 - » Higher priority queues often considered “foreground” tasks
 - Each queue has its own scheduling algorithm
 - » e.g. foreground – RR, background – FCFS
 - » Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
 - Job starts in highest priority queue
 - If timeout expires, drop one level
 - If timeout doesn’t expire, push up one level (or to top)

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
- 2000’s onward
 - Rise of streaming media, machine learning

Key Idea: Proportional-Share Scheduling

- The policies we've studied so far:
 - Always prefer to give the CPU to a prioritized job
 - Non-prioritized jobs may never get to run
- But priorities were a means, not an end:
 - Give priority to interactive tasks or I/O tasks for responsiveness
 - Lower priority given to long running tasks
- Instead, we can *share* the CPU *proportionally*
 - Give each job a share of the CPU according to its priority
 - Low-priority jobs get smaller share of CPU
 - But all jobs can at least make progress (no starvation)
- This idea is closely related to fair queueing

Lottery Scheduling

- Simple Idea:
 - Give each job some number of lottery tickets
 - On each time slice, randomly pick a winning ticket
 - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
 - To approximate SRTF, short running jobs get more, long running jobs get fewer
 - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
 - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses



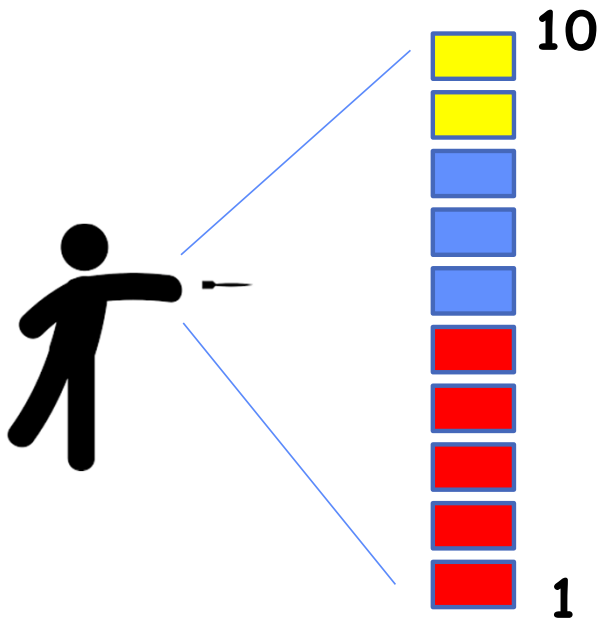
Lottery Scheduling Example (Cont.)

- Lottery Scheduling Example
 - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

- What if too many short jobs to give reasonable response time?
 - » If load average is 100, hard to make progress
 - » One approach: log some user out

Lottery Scheduling: Simple Mechanism



- $N_{ticket} = \sum N_i$
- Pick a number d in $1 \dots N_{ticket}$ as the random “dart”
- Jobs record their N_i of allocated tickets
- Order them by N_i
- Select the first j such that $\sum N_i$ up to j exceeds d .

Unfairness

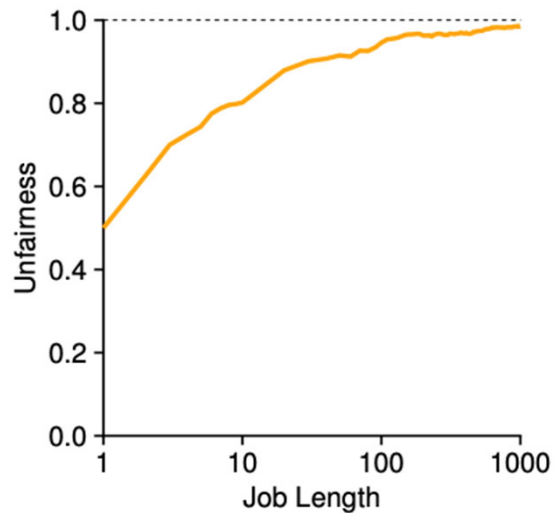


Figure 9.2: Lottery Fairness Study

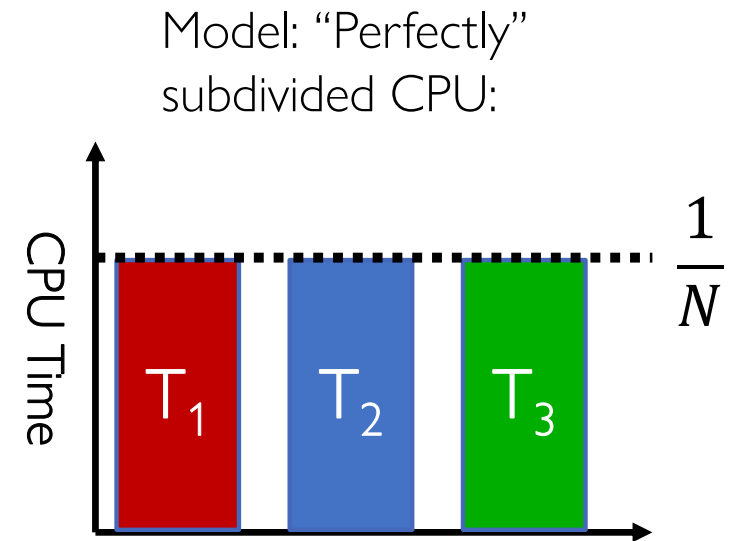
- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%,
 $U = \text{finish time of first} / \text{finish time of last}$
- As a function of run time

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 has 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, ...

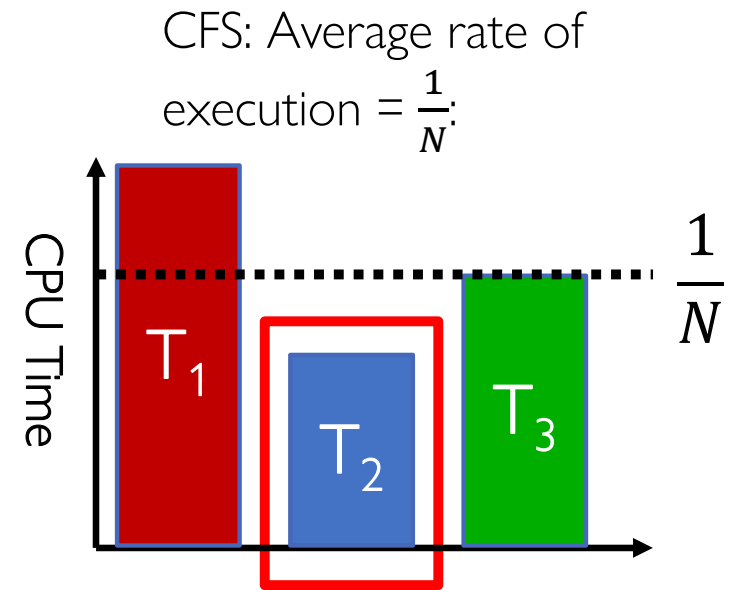
Linux Completely Fair Scheduler (CFS)

- Goal: Each process gets an equal share of CPU
 - N threads “simultaneously” execute on $\frac{1}{N}$ of CPU
 - The *model* is somewhat like simultaneous multithreading – each thread gets $\frac{1}{N}$ of the cycles
- In general, can't do this with real hardware
 - OS needs to give out full CPU in time slices
 - Thus, we must use something to keep the threads roughly in sync with one another



Linux Completely Fair Scheduler (CFS)

- Basic Idea: track CPU time per thread and schedule threads to match up average rate of execution
- Scheduling Decision:
 - “Repair” illusion of complete fairness
 - Choose thread with minimum CPU time
 - Closely related to Fair Queueing
- Use a heap-like scheduling queue for this...
 - $O(\log N)$ to add/remove threads, where N is number of threads
- Sleeping threads don’t advance their CPU time, so they get a boost when they wake up again...
 - Get interactivity automatically!



Linux CFS: Responsiveness/Starvation Freedom

- In addition to fairness, we want **low response time** and starvation freedom
 - Make sure that everyone gets to run at least a bit!
- Constraint 1: *Target Latency*
 - Period of time over which every process gets service
 - $Quanta = Target_Latency / n$
- Target Latency: 20 ms, 4 Processes
 - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
 - Each process gets **0.1ms** time slice (!!!)
 - Recall Round-Robin: large context switching overhead if slice gets to small

Linux CFS: Throughput

- Goal: Throughput
 - Avoid excessive overhead
- Constraint 2: Minimum Granularity
 - Minimum length of any time slice
- Target Latency 20 ms, Minimum Granularity 1 ms, 200 processes
 - Each process gets 1 ms time slice

Aside: Priority in Unix – Being Nice

- The industrial operating systems of the 60s and 70's provided priority to enforced desired usage policies.
 - When it was being developed at Berkeley, instead it provided ways to “be nice”.
- **nice** values range from -20 to 19
 - Negative values are “not nice”
 - If you wanted to let your friends get more time, you would nice up your job
- Scheduler puts higher nice-value tasks (lower priority) to sleep more ...
 - In $O(1)$ scheduler, this translated fairly directly to priority (and time slice)
- How does this idea translate to CFS?
 - Change the rate of CPU cycles given to threads to change relative priority

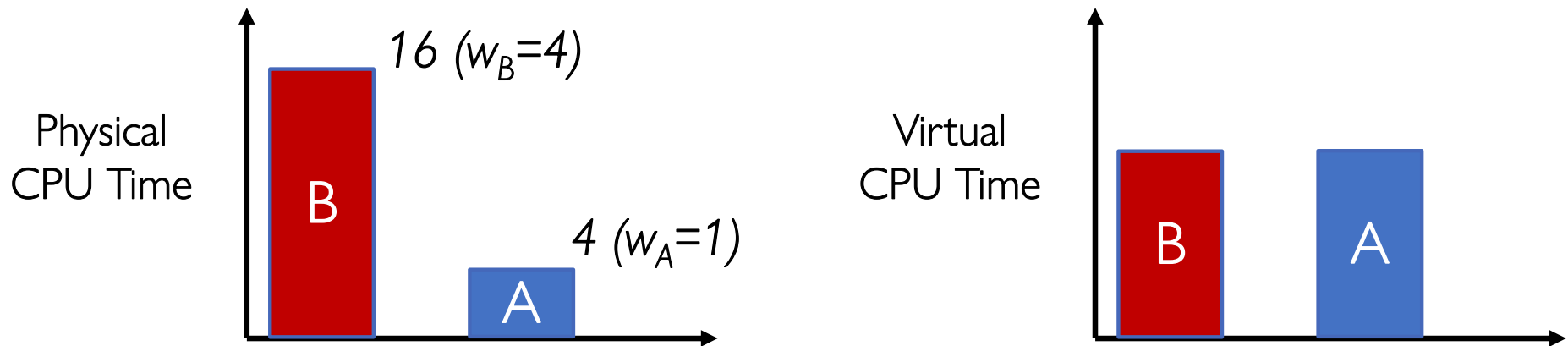
Linux CFS: Proportional Shares

- What if we want to give more CPU to some and less to others in CFS (proportional share) ?
 - Allow different threads to have different *rates* of execution (cycles/time)
- Use weights! Key Idea: Assign a weight w_i to each process i to compute the switching quanta Q_i
 - Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$
 - Weighted Share: $Q_i = \left(\frac{w_i}{\sum_p w_p} \right) \cdot \text{Target Latency}$
- Reuse **nice** value to reflect share, rather than priority,
 - Remember that lower nice value \Rightarrow higher priority
 - CFS uses nice values to scale weights exponentially: $\text{Weight} = 1024 / (1.25)^{\text{nice}}$
 - » Two CPU tasks separated by nice value of 5 \Rightarrow
Task with lower nice value has 3 times the weight, since $(1.25)^5 \approx 3$
- So, we use “Virtual Runtime” instead of CPU time
 - Virtual Runtime = Real CPU Time / Weight

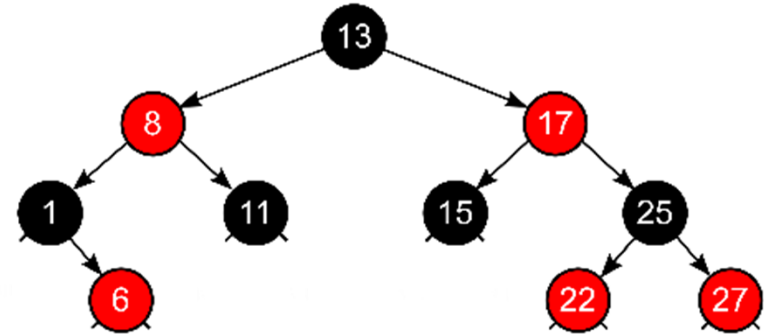
Example: Linux CFS: Proportional Shares

- Target Latency = 20ms
- Minimum Granularity = 1ms
- Example: Two CPU-Bound Threads
 - Thread A has weight 1
 - Thread B has weight 4
- Time slice for A? 4 ms
- Time slice for B? 16 ms

Linux CFS: Proportional Shares



- Track a thread's *virtual* runtime rather than its true physical runtime
 - Higher weight: Virtual runtime increases more slowly
 - Lower weight: Virtual runtime increases more quickly
- Scheduler's Decisions are based on Virtual CPU Time
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
 - $O(\log N)$ time to perform insertions/deletions
 - » Cache the item at far left (item with earliest vruntime)
 - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).



Administrivia

- Midterm 1 grading almost done!
 - Really close to releasing scores
 - I hear “probably tomorrow” from those in the know.
- Please look at solutions
 - Tomorrow will talk about Problem #3 solution in section (among normal topics)
- Project 1 due tomorrow
- Also due: Peer evaluations
 - Each partner gets 20 points x number of *other* partners to hand out for other partners
 - » i.e. for 4-partner group, each gets 60 points
 - These are a required mechanism for evaluating group dynamics
 - Project scores are a zero-sum game
 - » In the normal/best case, all partners get the same grade
 - » In groups with issues, we may take points from non-participating group members and give them to participating group members!

Administrivia (con't)

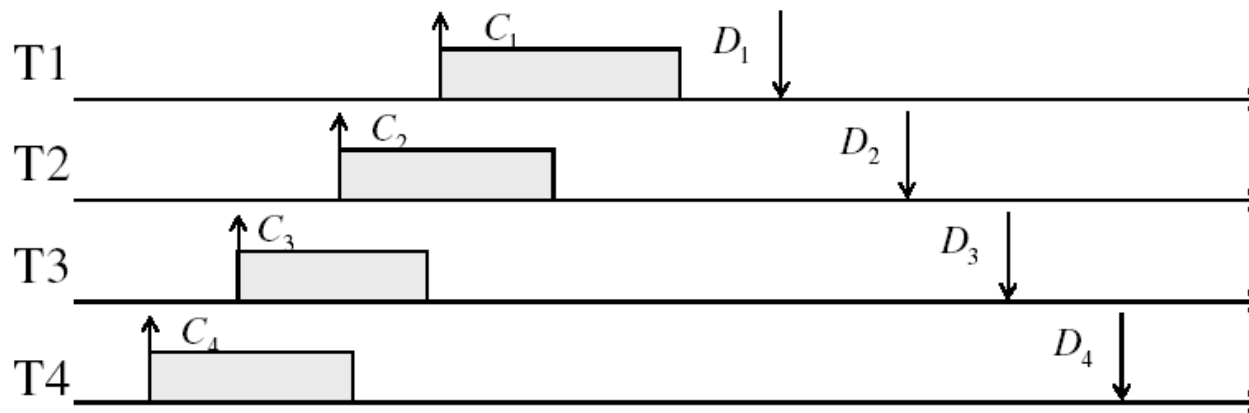
- Welcome to Homework 3:
 - Please get started earlier than last time!
 - Remember: Treat LLM use like cheating by asking a friend for help. Just say No.
- Project 2 starts on Saturday
- Midterm 2
 - Coming up in 3½ weeks! (Tuesday 3/31, First Tuesday after Spring Break)
 - Everything up to the midterm is fair game (before Spring Break).

Real-Time Scheduling

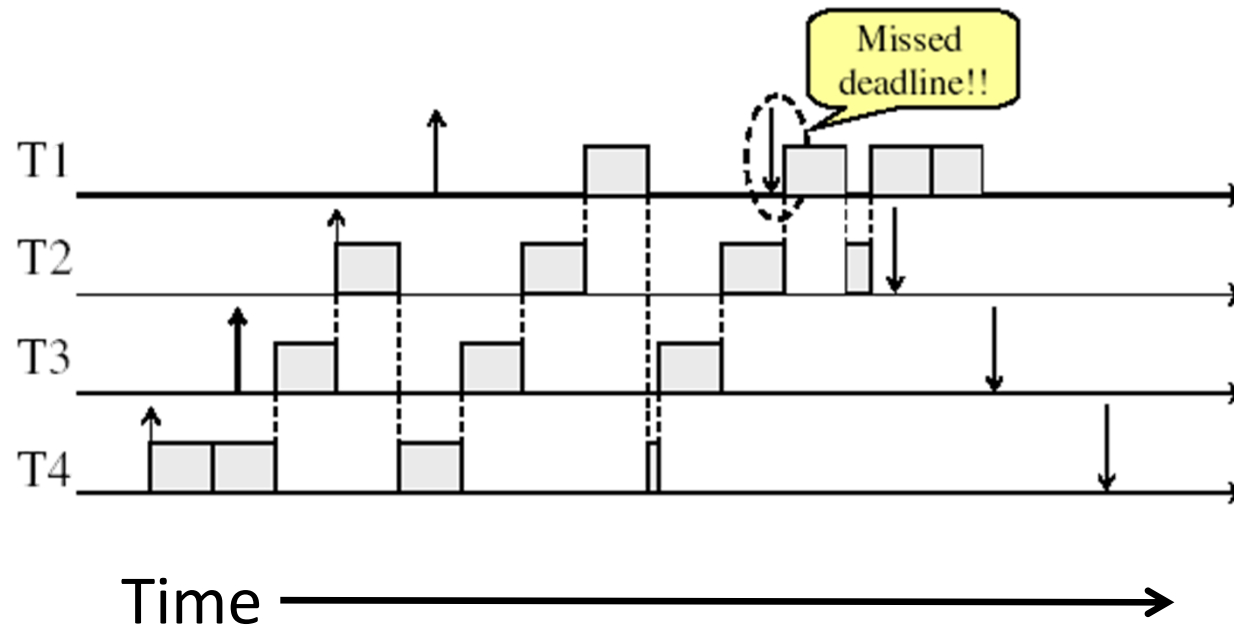
- Goal: **Predictability** of Performance!
 - We need to predict with confidence worst case response times for systems!
 - In RTS, performance guarantees are:
 - » Task- and/or class centric and often ensured a priori
 - In conventional systems, performance is:
 - » System/throughput oriented with post-processing (... wait and see ...)
 - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard real-time: for time-critical safety-oriented systems
 - Meet all deadlines (if at all possible)
 - Ideally: determine in advance if this is possible
 - **Earliest Deadline First (EDF), Least Laxity First (LLF), Rate-Monotonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)**
- Soft real-time: for multimedia
 - Attempt to meet deadlines with high probability
 - **Constant Bandwidth Server (CBS)**

Realtime Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup (not valid schedule yet, overlapping C):

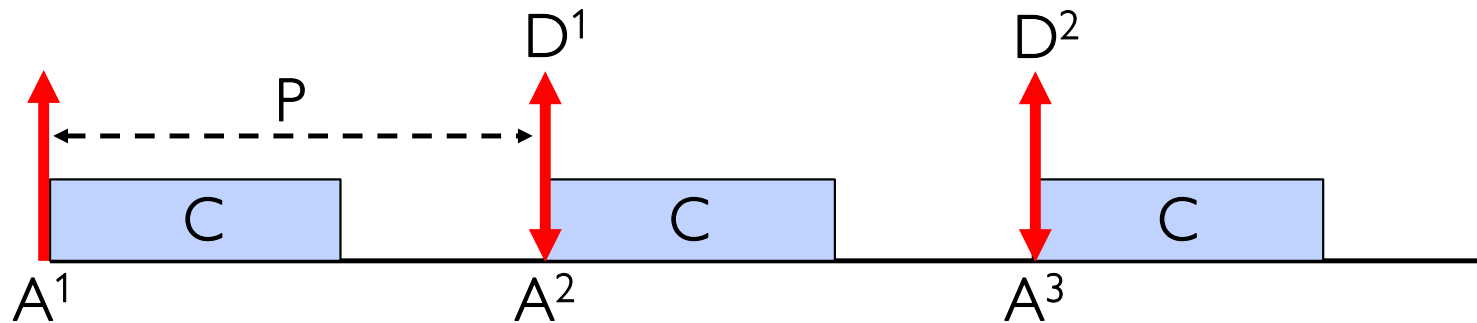


First Lesson: Round-Robin Scheduling Doesn't Work



- No dependence of scheduling choices on deadlines
 - So, not surprising it misses deadlines?
- But – what if we enhance workload model by adding periodicity?

Periodic (Realtime) Workload Model



- Tasks considered **periodic** with period P and computation C in each period:
 - Computation can be spread across period in many pieces
 - Said another way: Each task has periodic arrival times: (i.e. $A_i^{t+1} = A_i^t + P_i$ for each task)
- Each task is assigned a deadline at end of period: (i.e. $D_i^t = A_i^t + P_i$ for each task)
- We label these tasks: (P_i, C_i) for each task i
- Question: when have multiple concurrent (overlapping) tasks, how to schedule?

Rate Monotonic Scheduling (RMS)

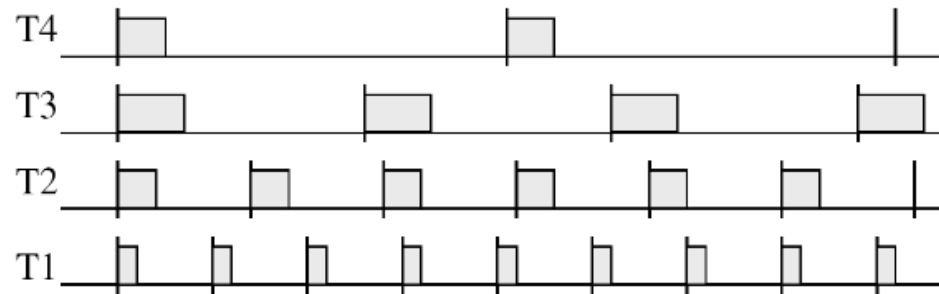
- Each process is assigned a (unique) priority based on its period (rate); always execute active job with highest priority
- The shorter the period the higher the priority
 $P_i < P_j \Rightarrow \pi_i > \pi_j$ (1 = low priority)
- Example: number the tasks in reverse order of priority

Process	Period	Priority (π)	Number
A	25	4	T1
B	60	2	T3
C	42	3	T2
D	100	1	T4

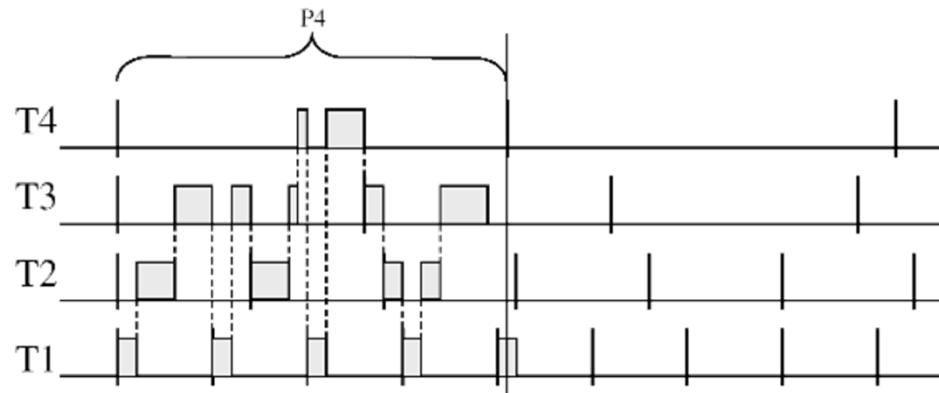
- Assumes a strict priority scheduler with n priorities ($n \Rightarrow$ number of tasks)
 - For example: realtime levels of Linux scheduler

Rate Monotonic Scheduling in action: Seems to work

- Example instance:



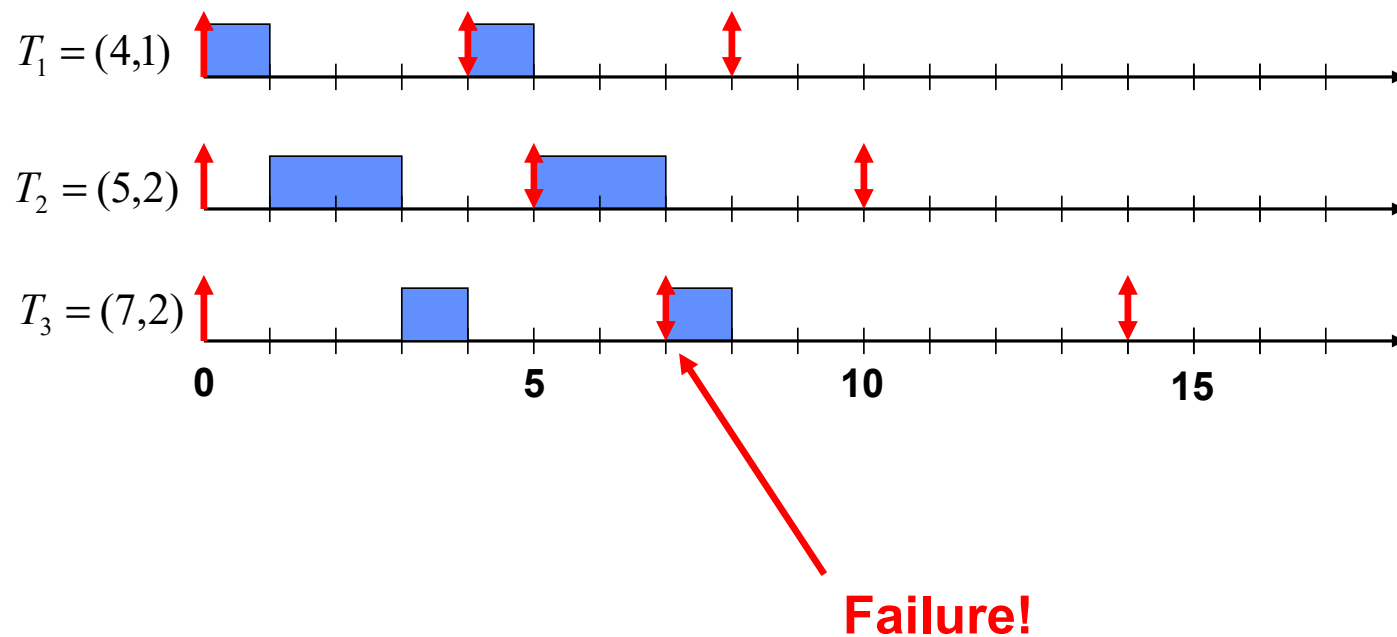
- RMA - Gant chart:



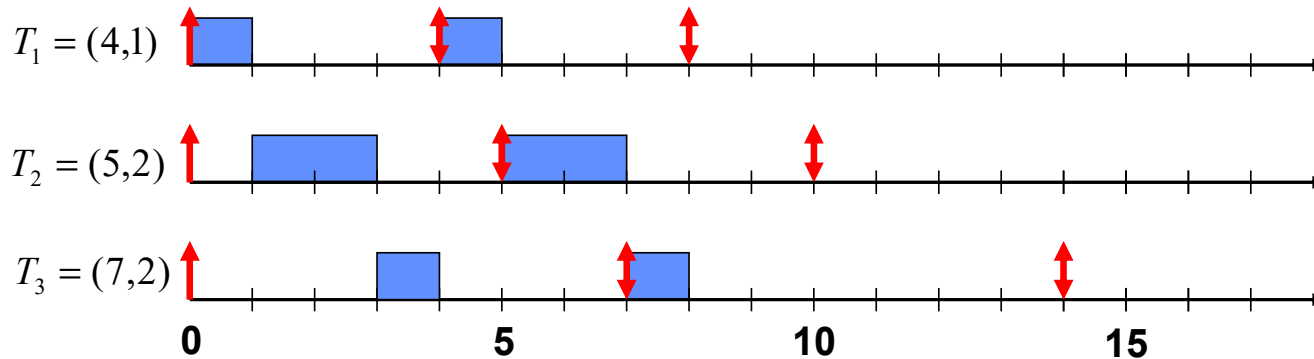
- If it meets deadlines, will continue to meet deadlines.
 - But does it always work?

No, RMS doesn't always work:

$T_i = (P_i, C_i)$ $P_i = \text{period}$ $C_i = \text{processing time}$



Compute *Utilization* to see if we are expecting too much:



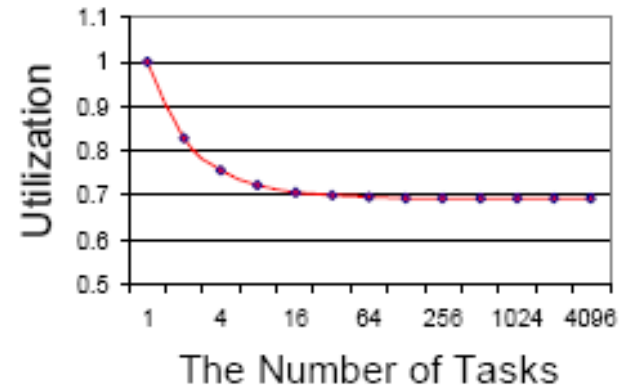
- Compute the *Utilization* of our tasks: $T_i \Rightarrow U_i = \frac{C_i}{P_i}$
 - $T_1 = (4,1) \Rightarrow U_1 = \frac{1}{4} = 0.25$
 - $T_2 = (5,2) \Rightarrow U_2 = \frac{2}{5} = 0.4$
 - $T_3 = (7,2) \Rightarrow U_3 = \frac{2}{7} \approx 0.286$
- Clearly, the sum of utilizations cannot be > 1 or we overload processor!
 - In our case: $\sum_{i=1}^n U_i \approx 0.936$
- **Not overloading our processor!** So, what gives?
 - Are we back to the fact that RMS does not look at deadlines?

RMS: Schedulability Test

- No: We don't need to look at deadlines as long as we meet schedulability criteria.
- Theorem (Utilization-based Schedulability Test):
 - A **periodic** task set T_1, T_2, \dots, T_n with $D_i = P_i$, $1 \leq i \leq n$, is schedulable by the rate monotonic scheduling algorithm if:

$$\sum_{i=1}^n \left(\frac{C_i}{P_i} \right) \leq n(2^{1/n} - 1), n = 1, 2, \dots$$

RM Utilization Bounds



- This schedulability test is “sufficient”!
 - i.e. for harmonic periods (P_j evenly divides P_i), the utilization bound is 100%
- Applying test to our failed Example #2, we get:

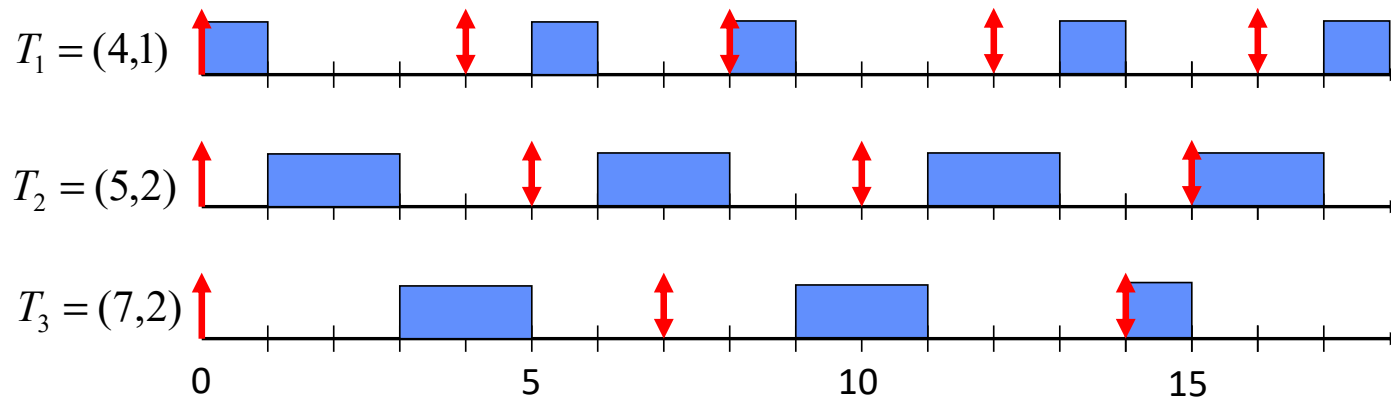
$$\sum_{i=1}^3 \left(\frac{C_i}{P_i} \right) \approx 0.936 > 3(2^{1/3} - 1) \approx 0.780$$

Does not satisfy schedulability condition!

$$n(2^{1/n} - 1) \rightarrow \ln 2 \text{ for } n \rightarrow \infty$$

Better Scheduling Policy: Earliest Deadline First (EDF)

- Preemptive priority-based dynamic scheduling *taking deadlines into account*
- Remember periodic task parameters:
 - Tasks **periodic** with period P and computation C in each period: (P_i, C_i) for each task i
 - Each task has periodic arrival times: (i.e. $A_i^{t+1} = A_i^t + P_i$ for each task)
 - Each task is assigned a (current) deadline as offset from arrival (i.e. $D_i^t = A_i^t + P_i$ for each task)
- EDF Policy: give CPU time to the active task with the closest absolute deadline:



• SUCCESS!

EDF: Schedulability Test

- Theorem (Utilization-based Schedulability Test):
 - A task set T_1, T_2, \dots, T_n with $D_i = P_i$ is schedulable by the earliest deadline first (EDF) scheduling algorithm if:

$$\sum_{i=1}^n \left(\frac{C_i}{P_i} \right) \leq 1$$

- Exact schedulability test (necessary + sufficient)
 - Proof: [Liu and Layland, 1973]
- For our problem Example #2:

$$\sum_{i=1}^3 \left(\frac{C_i}{P_i} \right) \approx 0.936 \leq 1$$

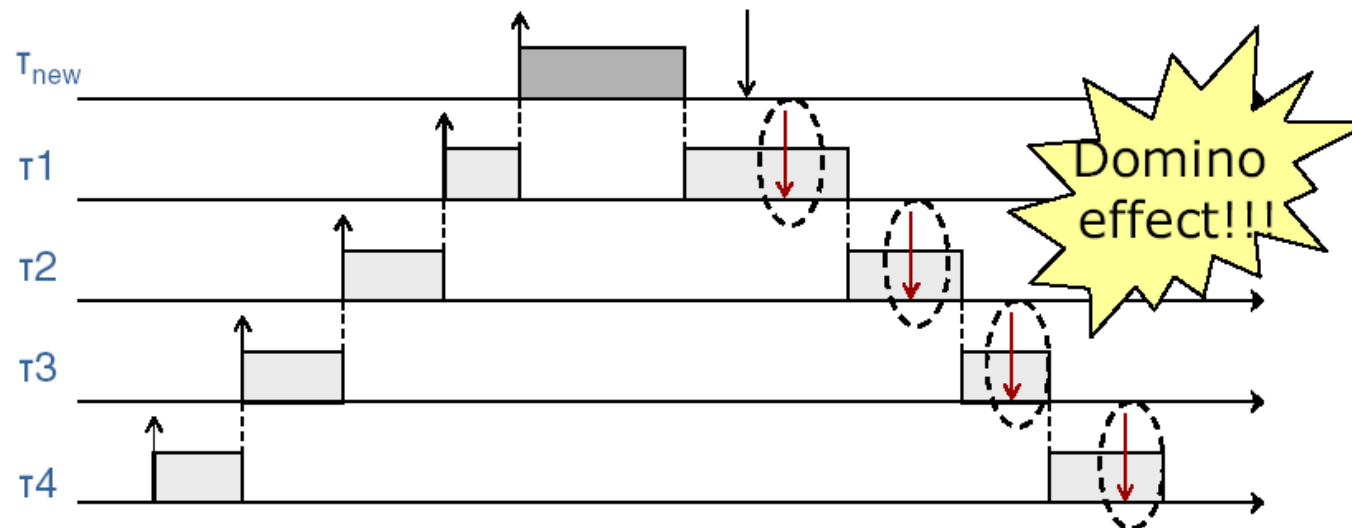
**Does Satisfy
schedulability condition!**



EDF Optimality

- EDF Properties:
 - EDF is optimal with respect to feasibility (i.e., schedulability)
 - EDF is optimal with respect to minimizing the maximum lateness
- Can be used with non-periodic tasks, but schedulability criteria gets a bit trickier!

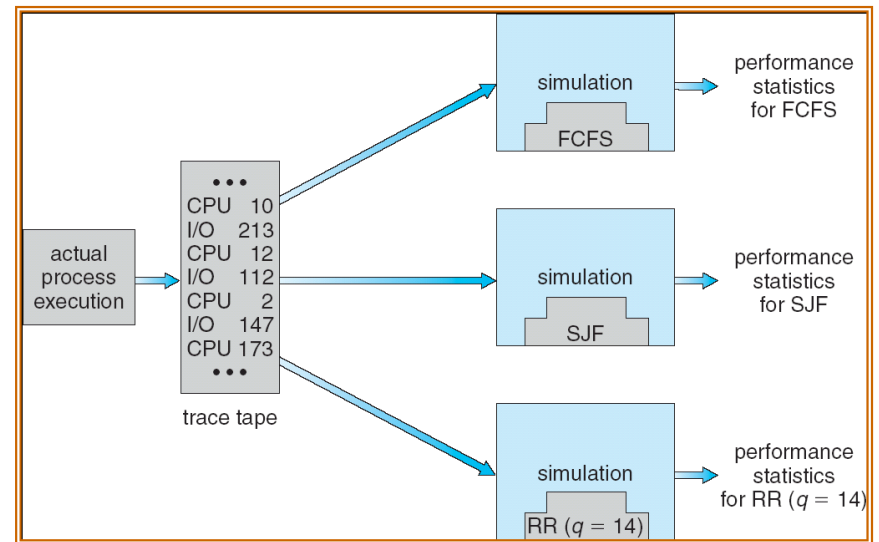
EDF Bad Example: Domino Effect



EDF minimizes lateness of the “most tardy task” [Dertouzos, 1974]

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
 - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
 - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
 - Build system which allows actual algorithms to be run against actual data
 - Most flexible/general

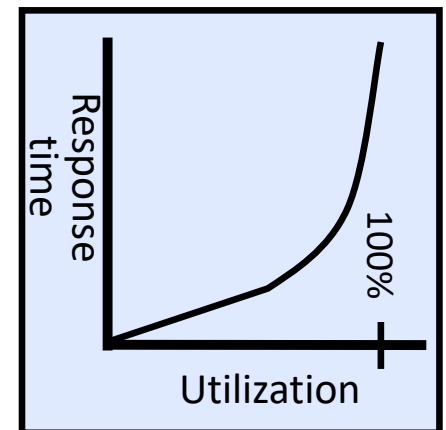


Choosing the Right Scheduler

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

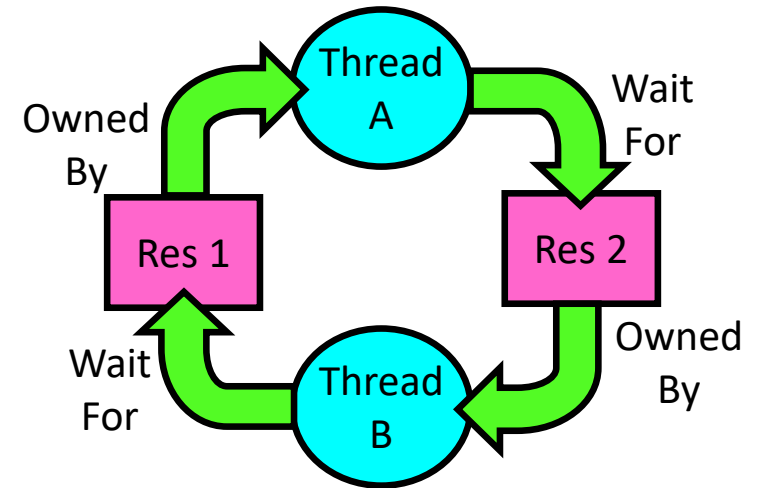
A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
 - When there aren't enough resources to go around
- When should you simply buy a faster computer?
 - (Or network link, or expanded highway, or ...)
 - One approach: Buy it when it will pay for itself in improved response time
 - » Perhaps you're paying for worse response time in reduced productivity, customer angst, etc...
 - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \Rightarrow 100%
- An interesting implication of this curve:
 - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
 - Argues for buying a faster X when hit “knee” of curve



Deadlock: A Deadly type of Starvation

- Starvation: thread waits indefinitely
 - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
 - Thread A owns Res 1 and is waiting for Res 2
 - Thread B owns Res 2 and is waiting for Res 1
- Deadlock \Rightarrow Starvation but not vice versa
 - Starvation can end (but doesn't have to)
 - Deadlock can't end without external intervention



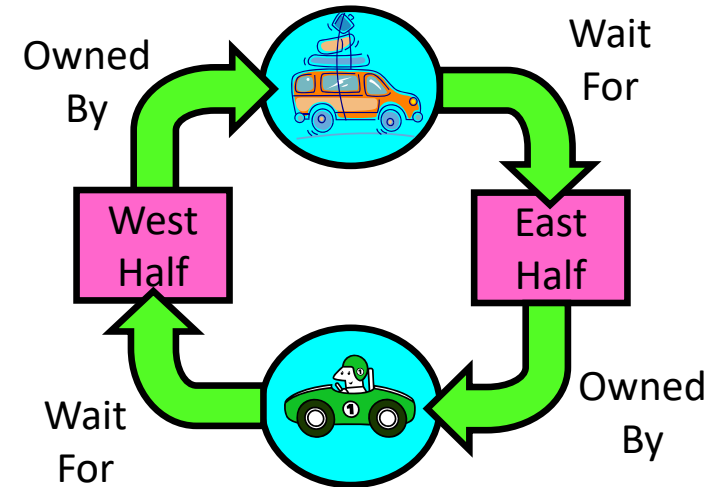
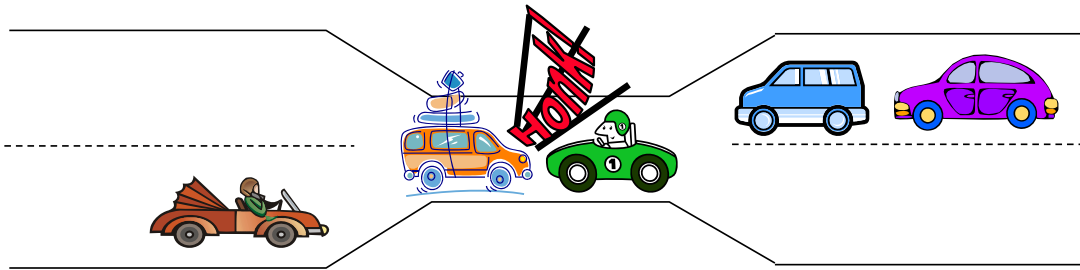
Example: Single-Lane Bridge Crossing



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
- For bridge: must acquire both halves
 - Traffic only in one direction at a time



- **Deadlock:** Shown above when two cars in opposite directions meet in middle
 - Each acquires one segment and needs next
 - Deadlock resolved if one car backs up (preempt resources and rollback)
 - » Several cars may have to be backed up
- Starvation (not Deadlock):
 - East-going traffic really fast \Rightarrow no one gets to go west

Deadlock with Locks

Thread A:

x.Acquire();

y.Acquire();

...

y.Release();

x.Release();

Thread B:

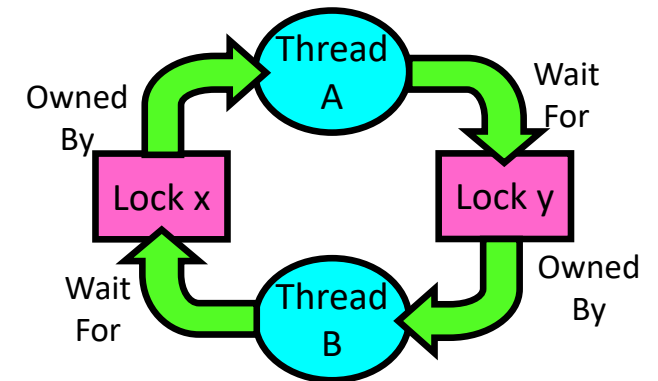
y.Acquire();

x.Acquire();

...

x.Release();

y.Release();



- This lock pattern exhibits *non-deterministic deadlock*
 - Sometimes it happens, sometimes it doesn't!
- This is really hard to debug!

Deadlock with Locks: “Unlucky” Case

Thread A:

x.Acquire();

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

...

y.Release();

x.Release();

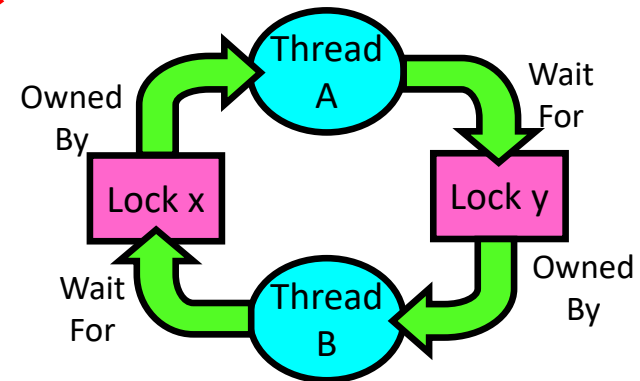
Thread B:

y.Acquire();

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

...

x.Release();
y.Release();



Neither thread will get to run \Rightarrow Deadlock

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)

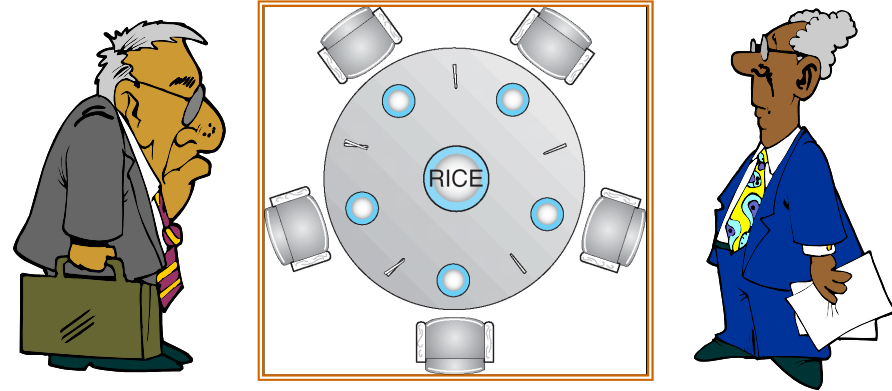
Free(1 MB)

Free(1 MB)

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

Dining Lawyers Problem

- Five chopsticks/Five lawyers (really cheap restaurant)
 - Free-for all: Lawyer will grab any one they can
 - Need two chopsticks to eat
- What if all grab at same time?
 - Deadlock!
- How to fix deadlock?
 - Make one of them give up a chopstick (Hah!)
 - Eventually everyone will get chance to eat
- How to prevent deadlock?
 - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
 - Can we formalize this requirement somehow?



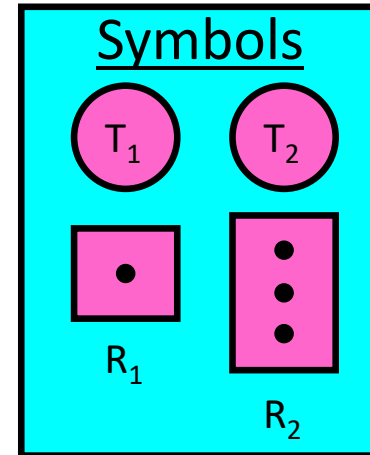
Four requirements for occurrence of Deadlock

- 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

Detecting 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 ()

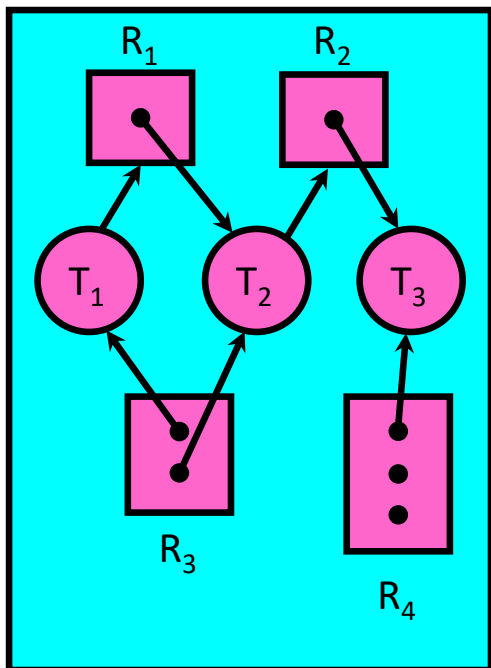


- Resource-Allocation Graph:

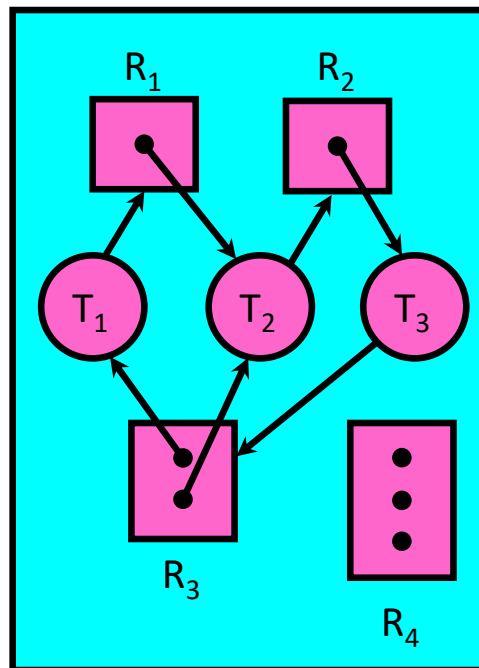
- V is partitioned into two types:
 - » $T = \{T_1, T_2, \dots, T_n\}$, the set threads in the system.
 - » $R = \{R_1, R_2, \dots, R_m\}$, the set of resource types in system
- request edge – directed edge $T_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow T_i$

Resource-Allocation Graph Examples

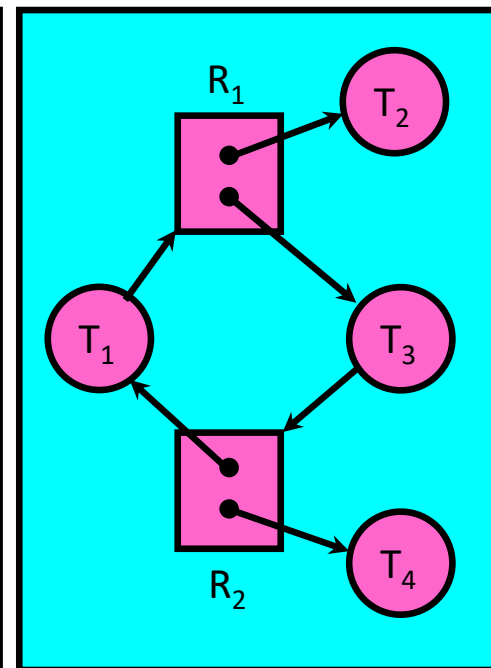
- Model:
 - request edge – directed edge $T_1 \rightarrow R_j$
 - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Deadlock Detection Algorithm

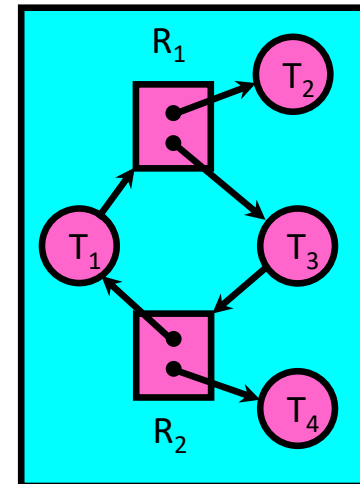
- Let $[X]$ represent an m-ary vector of non-negative integers (quantities of resources of each type):

$[FreeResources]$: 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



Conclusion

- Proportional Share Scheduling (Lottery Scheduling, Stride Scheduling CFS)
 - Give each job a share of the CPU according to its priority
 - Low-priority jobs get to run less often
 - But all jobs can at least make progress (no starvation)
- Four conditions for deadlocks
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
- Next Time: Techniques for addressing Deadlock
 - Deadlock prevention:
 - » write your code in a way that it isn't prone to deadlock
 - Deadlock recovery:
 - » let deadlock happen, and then figure out how to recover from it
 - Deadlock avoidance:
 - » dynamically delay resource requests so deadlock doesn't happen
 - » Banker's Algorithm provides an algorithmic way to do this
 - Deadlock denial:
 - » ignore the possibility of deadlock