CS 370: OPERATING SYSTEMS
[SCHEDULING ALGORITHMS & DEADLOCKS]

Shrideep Pallickara
Computer Science
Colorado State University

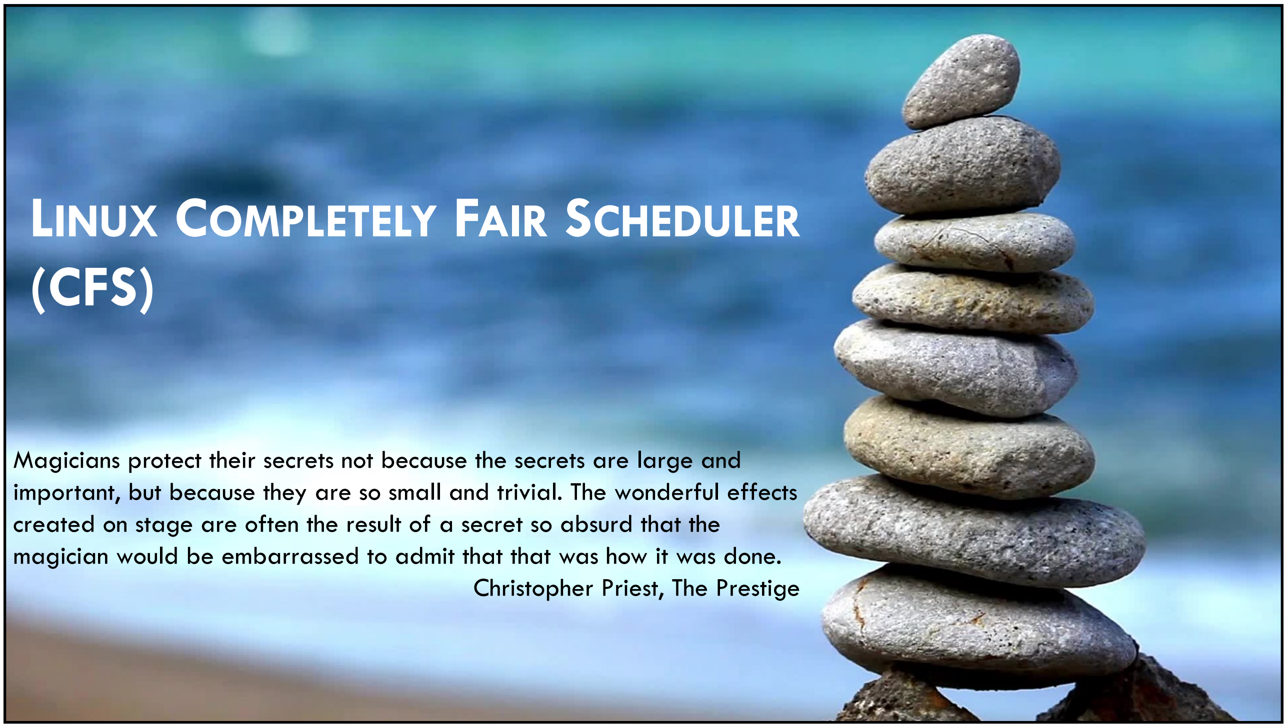
Frequently asked questions from the previous class survey

- ☐ Is starvation still a problem in modern, multicore systems?
- ☐ Where is the scheduler?
- ☐ What does overhead refer to?

Thilina Buddhika*, Ryan Stern*, Kira Lindburg*, Kathleen Ericson*, and Shrideep Pallickara. Online Scheduling and Interference Alleviation for Low-latency, High-throughput Processing of Data Streams. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 28(12) pp 3553-3569. 2017.

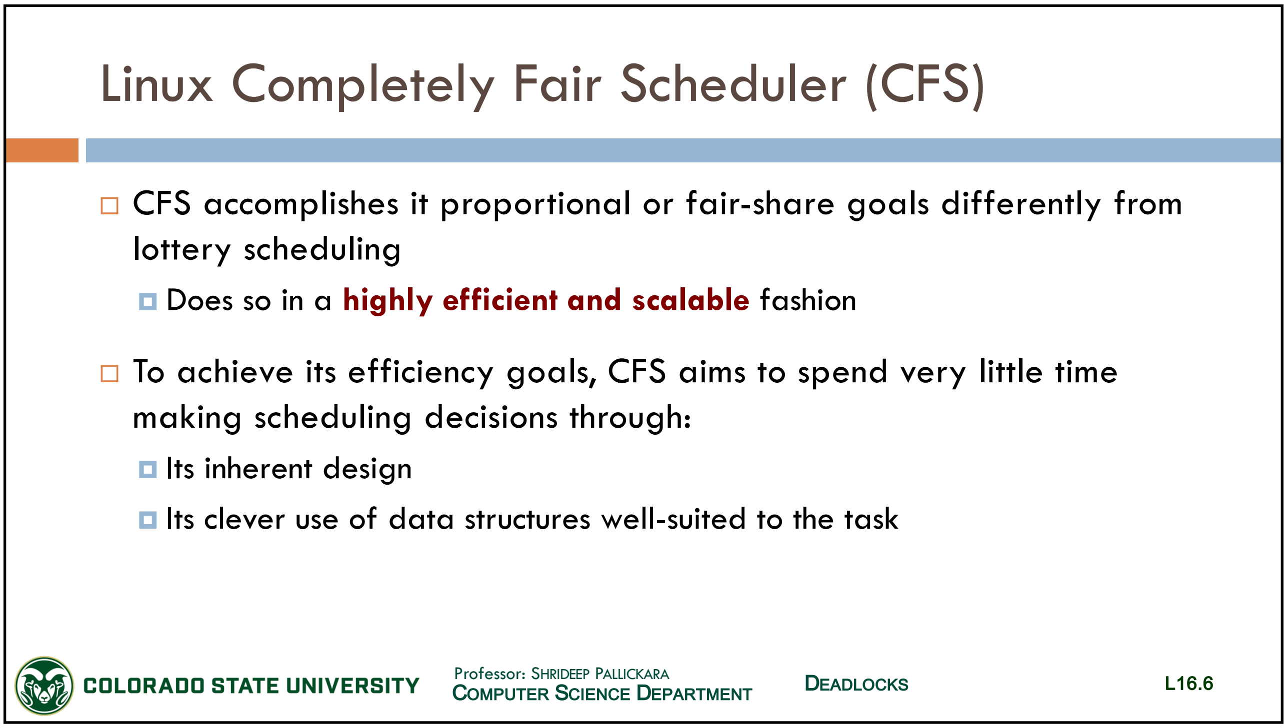
Topics covered in today's lecture

- Wrap-up of CPU Scheduling Algorithms
  - CFS
  - Idle Threads in Windows
- Deadlocks
- Deadlock characterization
- Deadlock vs Starvation
- Resource allocation graph



LINUX COMPLETELY FAIR SCHEDULER (CFS)

> Magicians protect their secrets not because the secrets are large and important, but because they are so small and trivial. The wonderful effects created on stage are often the result of a secret so absurd that the magician would be embarrassed to admit that that was how it was done.
>
> Christopher Priest, The Prestige



Linux Completely Fair Scheduler (CFS)

- CFS accomplishes it proportional or fair-share goals differently from lottery scheduling
  - Does so in a **highly efficient and scalable** fashion
- To achieve its efficiency goals, CFS aims to spend very little time making scheduling decisions through:
  - Its inherent design
  - Its clever use of data structures well-suited to the task

CFS: Basic Operation

- Whereas most schedulers are based around the concept of a fixed time slice, CFS operates a bit differently
- GOAL: Fairly divide a CPU evenly among all competing processes
  - Does so through a simple counting-based technique known as **virtual runtime** (**`vruntime`**)

`vruntime`

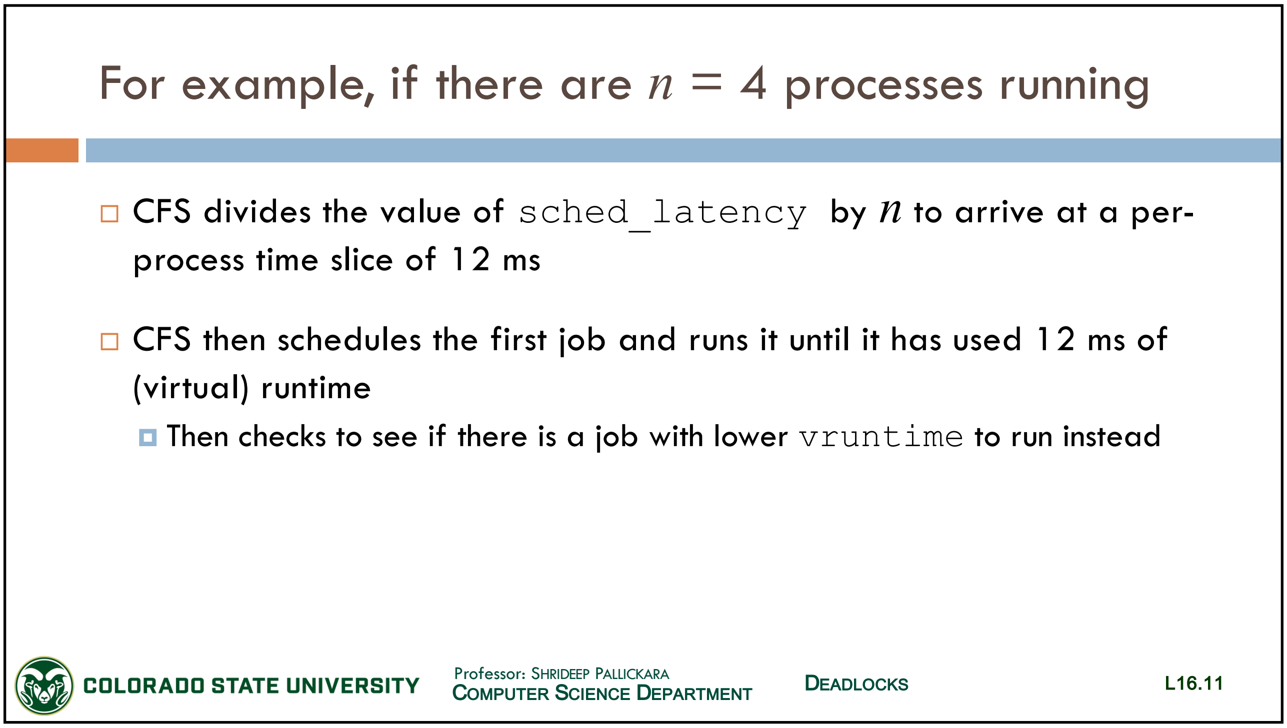
- As each process runs, it **accumulates** `vruntime`
- In the most basic case, each process's `vruntime` increases at the same rate, in proportion with physical (real) time
- When a scheduling decision occurs, CFS will pick the process with the ***lowest*** `vruntime` to run next

How does the scheduler know when to stop the currently running process, and run the next one?

- Trade-off Space:
  - If CFS switches too often?
    - Fairness is increased: CFS will ensure that each process receives its share of CPU even over miniscule time windows
    - But at the cost of performance (too much context switching)
  - If CFS switches less often?
    - Performance is increased (reduced context switching)
    - But at the cost of near-term fairness

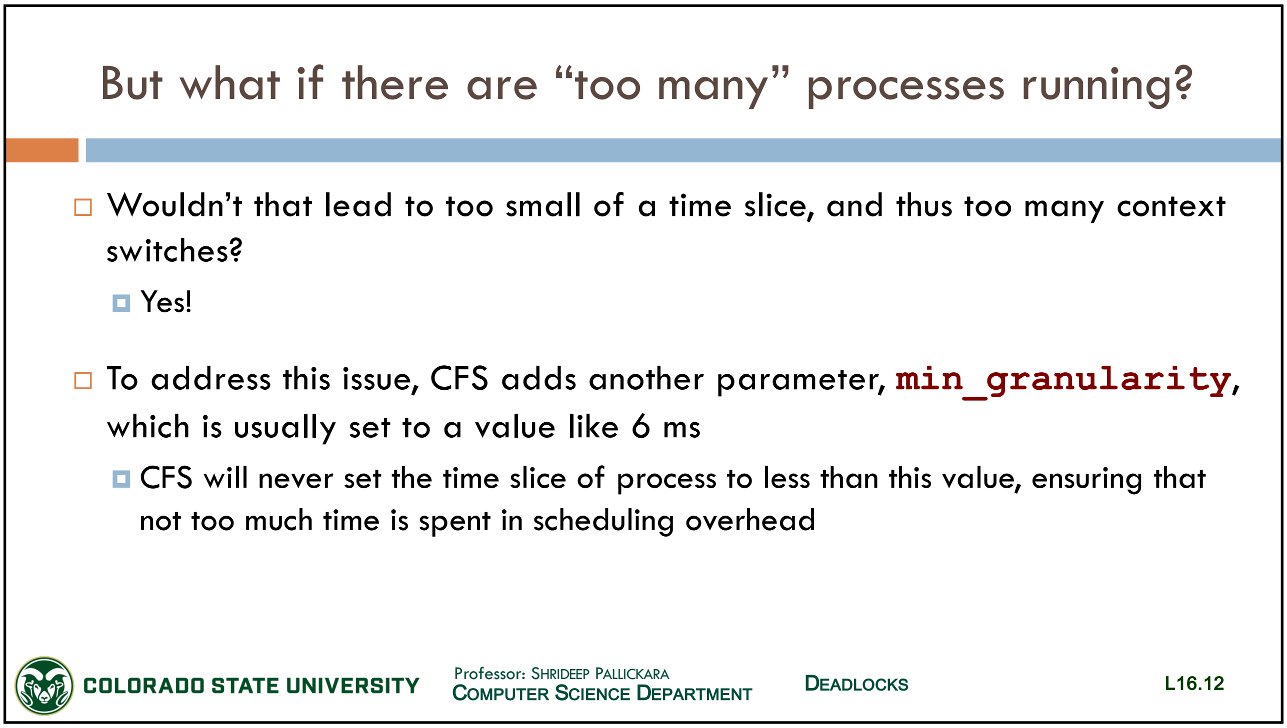
CFS manages this trade-off through various control parameters

- **`sched_latency`**
  - CFS uses this value to determine how long one process should run before considering a switch
    - Effectively determining its **time slice but in a dynamic fashion**
- A typical `sched_latency` value is 48 (milliseconds)
  - CFS divides this value by the number (n) of processes running on the CPU to determine the time slice for a process
    - And thus, ensures that over this period of time, CFS will be completely fair



For example, if there are $n = 4$ processes running

- CFS divides the value of `sched_latency` by n to arrive at a per-process time slice of 12 ms
- CFS then schedules the first job and runs it until it has used 12 ms of (virtual) runtime
  - Then checks to see if there is a job with lower `vruntime` to run instead



But what if there are "too many" processes running?

- Wouldn't that lead to too small of a time slice, and thus too many context switches?
  - Yes!
- To address this issue, CFS adds another parameter, **`min_granularity`**, which is usually set to a value like 6 ms
  - CFS will never set the time slice of process to less than this value, ensuring that not too much time is spent in scheduling overhead

For example, if there are ten processes running

- Our original calculation would divide `sched_latency` by ten to determine the time slice (result: 4.8 ms)
  - However, because of min granularity, CFS will set the time slice of each process to 6 ms instead
- Although CFS won't (quite) be perfectly fair over the target scheduling latency (`sched_latency`) of 48 ms, it will be close
  - While still achieving high CPU efficiency

CFS utilizes a **periodic timer interrupt**

- CFS can only make decisions at **fixed time intervals**
- This interrupt goes off frequently (e.g., **every 1 ms**)
  - Giving CFS a chance to wake up and determine if the current job has reached the end of its run
- If a job has a time slice that is *not a perfect multiple* of the timer interrupt interval?
  - That is OK
  - CFS tracks `vruntime` precisely, which means that over the long haul, it will eventually approximate ideal sharing of the CPU

CFS: WEIGHTING OR NICENESS

Weighting (Niceness)

- CFS also enables controls over process priority to give some processes a higher share of the CPU.
  - It does this *not with tickets*, but through a classic UNIX mechanism known as the **nice** level of a process
- The nice parameter can be set anywhere from −20 to +19 for a process, with a default of 0
  - Positive nice values imply *lower* priority and negative values imply *higher* priority
  - When you're too nice, you just don't get as much (scheduling) attention, alas!

CFS maps the nice value of each process to a weight

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

These weights allow us to compute the effective time slice of each process

- As we did before, but now accounting for their priority differences

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency}$$

Example: Assume there are two jobs **A** and **B**

- **A** has a higher priority by assigning it a nice value of −5:
- **B** has the default priority (nice value equal to 0)
- Note: weight_A (from the table) is 3121, whereas weight_B is 1024
- **A**'s time-slice: 3121/[3121+ 1024] ~ ¾
- **B**'s time-slice: 1024/[3121+ 1024] ~ ¼

The way CFS calculates `vruntime` must also be adapted

- The new formula, which takes the actual run time that process i has accrued ($\texttt{runtime}_\texttt{i}$) and scales it inversely by the weight of the process
  - By dividing the default weight of 1024 ($\texttt{weight}_0$) by its weight, $\texttt{weight}_\texttt{i}$

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

N.B: When a scheduling decision occurs, CFS will pick the process with the *lowest* **`vruntime`** to run next

EFFICIENT DATA STRUCTURES

Using efficient data structures

- Knowing *which data structure to use when* is a hallmark of good design
- When picking a data structure for a system you are building, carefully consider its access patterns and its frequency of usage
  - By understanding these, you will be able to implement the right structure for the task at hand

Schedulers and data structures

- When the scheduler has to find the next job to run, it should do so as quickly as possible
- Simple data structures like **lists don't scale**: modern systems sometimes comprise1000s of processes
  - Searching through a long-list every so many milliseconds is wasteful

CFS addresses this by keeping processes in a **red-black tree**

- A red-black tree is one of many types of **balanced trees**; in contrast to a simple binary tree
  - Binary trees can degenerate to list-like performance under worst-case insertion patterns
  - Balanced trees do a **little extra work to maintain low depths**, and thus ensure that operations are **logarithmic** (and not linear) in time
  - Worst case search, insert, delete: O(log n)
    - Amortized: O(log n), O(1), O(1)

CFS and red-black trees

- Processes are ordered in the tree by `vruntime`, and most operations (such as insertion and deletion) are logarithmic in time, *i.e.*, $O(\log n)$
  - When n is in the thousands, logarithmic is noticeably more efficient than linear
- CFS does not keep *all* process in this structure; rather, only running (or runnable/ready) processes
- If a process goes to sleep (say, waiting on an I/O to complete, or for a network packet to arrive), it is removed from the tree and kept track of elsewhere

DEALING WITH I/O AND SLEEPING PROCESSES

Dealing With I/O And Sleeping Processes [1/2]

- One problem with picking the lowest `vruntime` to run next arises with jobs that have gone to sleep for a long period of time
- Imagine two processes, **A** and **B**
  - **A** runs continuously, and **B** which has gone to sleep for a long period of time (say, 10 seconds)
  - When **B** wakes up, its `vruntime` will be 10 seconds behind **A**'s
  - Thus (if we're not careful), **B** will now *monopolize* the CPU for the next 10 seconds while it catches up, effectively starving **A**

Dealing With I/O And Sleeping Processes [1/2]

- CFS handles this case by **altering** the `vruntime` of a job when it wakes up
- Specifically, CFS sets the `vruntime` of that job to the minimum value found in the tree
  - In this way, CFS **avoids starvation**, but not without a cost
    - Jobs that sleep for short periods of time frequently do not ever get their fair share of the CPU

IDLE THREADS

Dispatcher in Windows XP

- Use a **queue** for each scheduling priority
- **Traverse** the queues from highest to lowest
  - *Until* it finds a thread that is ready to run
- If no ready thread is found?
  - Dispatcher will execute a special thread: **idle thread**

Idle thread in Windows

- Primary purpose is to **eliminate a special case**
  - Cases when no threads are runnable or ready
  - Idle threads are always in a *ready* state
    - If not already running
- Scheduler can always find a thread to execute
- If there are other eligible threads?
  - Scheduler will never select the idle thread

Idle threads in Windows

- Windows thread priorities go from 0-31
  - Idle thread priority can be thought of as −1
- Threads in the system idle process can also implement CPU power saving
  - On x86 processors, run a loop of **halt** instructions
  - Causes CPU to **turn off internal components**
    - Until an interrupt request arrives
  - Recent versions also **reduce the CPU clock speed**

Time consumed by the idle process

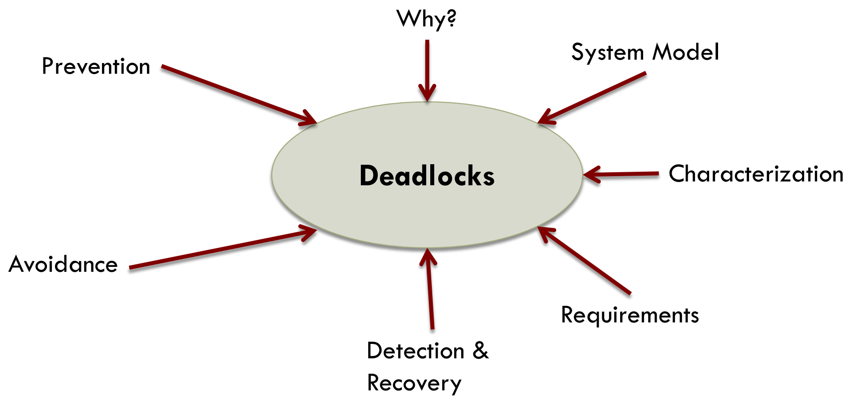
- It may seem that the idle process is monopolizing the CPU
  - It is merely acting as a *placeholder during free time*
  - Proof that no other process wants that CPU time

Afraid of what the truth might bring
He locks his doors and never leaves
Desperately searching for signs
To terrify, to find a thing
He battens all the hatches down
And wonders why he hears no sound
Frantically searching his dreams
He wonders what it's all about
Telescope, Cage the Elephant

A waiting process is never again able to change state
It is waiting for resources held by other processes

DEADLOCKS

What we will look at …



Why?

System Model

Prevention

Deadlocks

Characterization

Avoidance

Requirements

Detection &
Recovery

For many applications, processes need exclusive accesses to multiple resources

- Process A: Asks for scanner and is granted it
- Process B: Asks CD recorder first and is granted it
- Process A: Now asks for CD recorder
- Process B: Now asks for Scanner

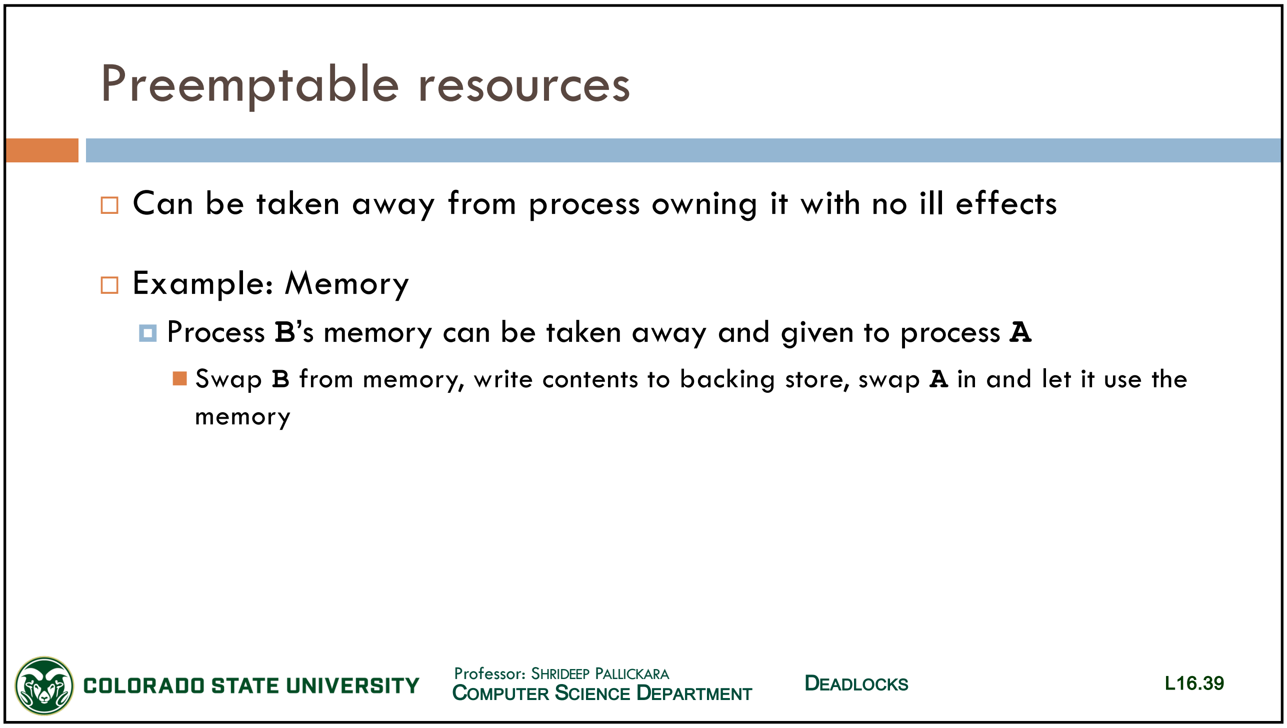
- Both processes are blocked and will remain so forever!
  - **Deadlock**

Other deadlock situations

- Distributed systems involving multiple machines
- Database systems
  - Process 1 locks record R1
  - Process 2 locks record R2
  - Then, processes 1 and 2 try to lock each other's record
    - Deadlock
- **Deadlocks can occur in hardware or software resources**

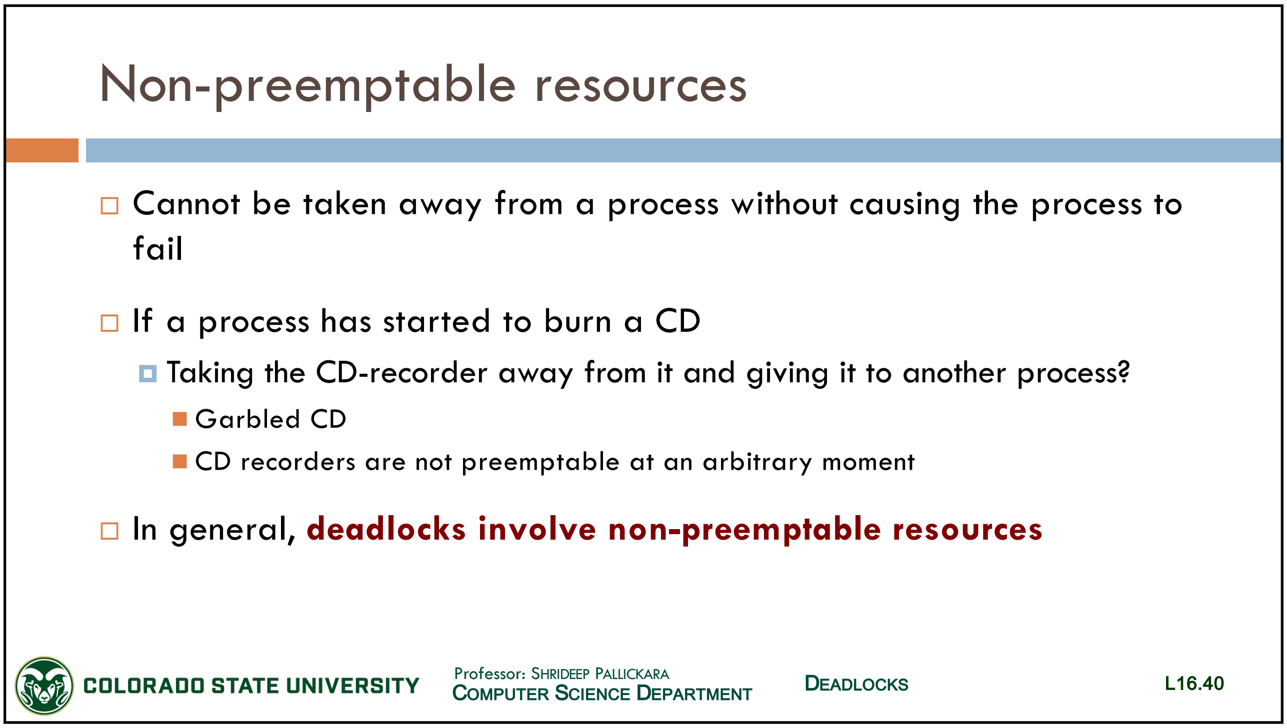
Resource Deadlocks

- Major class of deadlocks involves resources
  - Can occur when processes have been granted access to devices, data records, files, etc.
  - Other classes of deadlocks: communication deadlocks, two-phase locking
- Related concepts
  - Livelocks and starvation



Preemptable resources

- Can be taken away from process owning it with no ill effects
- Example: Memory
  - Process **B**'s memory can be taken away and given to process **A**
    - Swap **B** from memory, write contents to backing store, swap **A** in and let it use the memory



Non-preemptable resources

- Cannot be taken away from a process without causing the process to fail
- If a process has started to burn a CD
  - Taking the CD-recorder away from it and giving it to another process?
    - Garbled CD
    - CD recorders are not preemptable at an arbitrary moment
- In general, **deadlocks involve non-preemptable resources**

Some notes on deadlocks

- The OS typically does not provide deadlock prevention facilities

- Programmers are *responsible* for designing deadlock free programs

System model

- **Finite** number of resources
  - Distributed among *competing processes*

- Resources are *partitioned* into different ***types***
  - Each *type* has a number of identical instances
  - Resource type examples:
    - Memory space, files, I/O devices

A process must utilize resources in a sequence

- **Request**
  - Requesting resource must *wait until it can acquire* resource
  - `request(),open(),allocate()`
- **Use**
  - Operate on the resource
- **Release**
  - `release(), close(), free()`

For kernel managed resources, the OS maintains a system resource table

- Is the resource free?
  - Record process that the resource is allocated to
- Is the resource allocated?
  - Add to queue of processes waiting for resource
- For resources not managed by the OS
  - Use `wait()` and `signal()` on semaphores

Deadlock: Formal Definition

- *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*
- Because all processes are waiting, none of them can cause events to wake any other member of the set
  - Processes continue to **wait forever**

DEADLOCK CHARACTERIZATION

Deadlocks: Necessary Conditions (I)

- **Mutual Exclusion**
  - At least one resource held in *nonsharable* mode
  - When a resource is being used
    - Another requesting process must wait for its release
- **Hold-and-wait**
  - A process must hold one resource
  - Wait to acquire additional resources
    - Which are currently held by other processes

Deadlocks: Necessary Conditions (II)

- **No preemption**
  - Resources cannot be preempted
  - Only voluntary release by process holding it
- **Circular wait**
  - A set of {P_0, P_1, …, P_n} waiting processes must exist
    - $P_0 \rightarrow P_1$; $P_1 \rightarrow P_2$, …, $P_n \rightarrow P_0$
  - Implies hold-and-wait

DEADLOCKS VS. STARVATION

Deadlocks vs. Starvation [1/2]

- Deadlocks and starvation are both **liveness** concerns
- Starvation
  - Task fails to make progress for an indefinite period of time
- Deadlock is a *form of starvation*, BUT with a stronger condition
  - A **group of tasks** forms a **cycle** where *none* of the tasks makes progress
    - Because each task is waiting for some other task in the cycle to take action

Deadlocks vs. Starvation [2/2]

- Deadlock implies starvation (literally for the dining philosophers problem)
- Starvation DOES NOT imply deadlock

Also …

- Just because a system can suffer deadlock or starvation does not mean that it always will
  - A system is *subject to starvation* if a task could starve in some circumstances
  - A system is *subject to deadlock* if a group of tasks could deadlock in some circumstances
- **Circumstances** impact whether a deadlock or starvation may occur
  - Choices made by scheduler, number of tasks, workload or sequence of requests, which tasks win races to acquire locks, order of task activations, etc.

RESOURCE ALLOCATION GRAPH

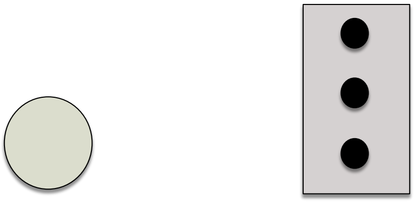
Resource allocation graph

- ☐ Used to describe deadlocks precisely
- ☐ Consists of a set of vertices and edges
- ☐ Two different sets of nodes
    - P: the set of all **active processes** in system
    - R: the set of all **resource types** in the system

Directed edges

- **Request** edge
  - P_i has requested an instance of resource type R_j
  - Directed edge from process P_i to resource R_j
  - Denoted P_i → R_j
  - *Currently waiting* for that resource
- **Assignment** edge
  - Instance of resource R_j assigned to process P_i
  - Directed edge from resource R_j to process P_i
  - Denoted R_j → P_i

Representation of Processes and Resources

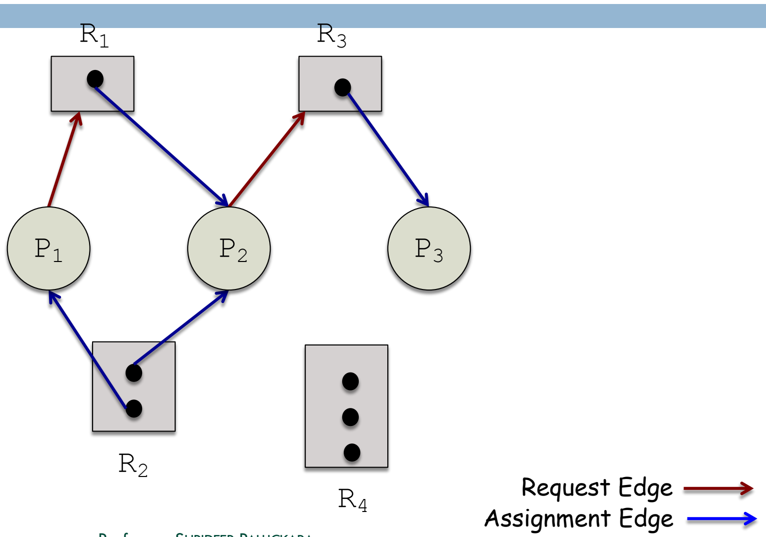


Processes

Resources

A resource type may have
multiple instances

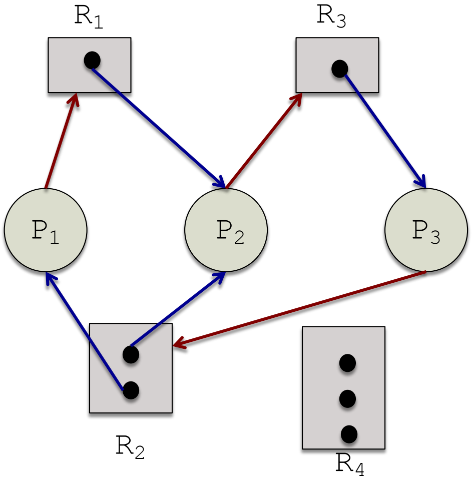
Resource Allocation Graph example



Determining deadlocks

- If the graph contains **no cycles**?
  - No process in the system is deadlocked
- If there is a **cycle** in the graph?
  - If each resource type has *exactly one* instance
    - Deadlock ***has*** occurred
  - If each resource type has *multiple* instances
    - A deadlock ***may have*** occurred

Resource Allocation Graph: Deadlock example

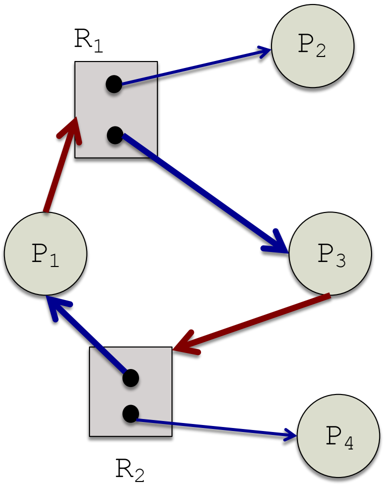


Two cycles

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Resource Allocation Graph: Cycle but not a deadlock



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

P_4 may release instance of R_2 allocate to P_3 and break cycle

Resource Allocation Graphs and Deadlocks

- If the graph does not have a cycle
  - No deadlock
- If the graph does have a cycle
  - System may or may not be deadlocked

Methods for handling deadlocks

- Use protocol to **prevent** or **avoid** deadlocks
  - Ensure system never enters a deadlocked state
- Allow system to enter deadlocked state; BUT
  - **Detect** it and **recover**
- Ignore problem, pretend that deadlocks never occur

Problems with undetected deadlocks

- Resources held by processes that cannot run
- More and more processes enter deadlocked state
  - When they request more resources
- **Deterioration** in system performance
  - Requires restart

When is ignoring the problem viable?

- When they occur infrequently (once per year)
  - Ignoring is the *cheaper* solution
  - Prevention, avoidance, detection and recovery
    - Need to run constantly

The contents of this slide-set are based on the following references

- *Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Operating Systems: Three Easy Pieces. 1st edition. CreateSpace Independent Publishing Platform. ISBN-13: 978-1985086593. [Chapter 9]*
- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 5, 7]
- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 7]
- *Thomas Anderson and Michael Dahlin. Operating Systems Principles and Practice. 2nd Edition. ISBN: 978-0985673529.* [Chapter 6]