Deadlock

Professor Natacha Crooks
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi
**Linux Completely Fair Scheduler (CFS)**

**Basic Idea**

Track CPU time per thread

**CFS: Average rate of execution**

$$\frac{1}{N}$$

**Scheduling Decision**

“Repair” illusion of complete fairness

Choose thread with minimum CPU time
Linux Completely Fair Scheduler (CFS)

Fair by construction

Scheduling Cost is $O(\log n)$
Threads are stored in a Red-Black tree.

Easy to capture interactivity
Sleeping threads don’t advance their CPU time, so automatically get a boost when wake up again
Low response time & Starvation-freedom
Make sure that everyone gets to run in a given period of time

Constraint 1: Target Latency

Period of time over which every process gets service

Quanta = Target_Latency / n
**Constraint 1: Target Latency**

\[ \text{Quanta} = \frac{\text{Target Latency}}{n} \]

**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
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
Linux CFS: Proportional Shares

Allow different threads to have different rates of execution (cycles/time)

Use weights!
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:** $Q_i = \left(\frac{w_i}{\sum_p w_p}\right) \cdot \text{Target Latency}$
Linux CFS: Proportional Shares

Target Latency = 20ms
Minimum Granularity = 1ms

Two CPU-Bound Threads
- Thread A has weight 1
- Thread B has weight 4

What should the time slice of A and B be?

Weighted Share: \( Q_i = \left( \frac{w_i}{\Sigma_p w_p} \right) \cdot \text{Target Latency} \)

\[ A = \left( \frac{1}{5} \right) \cdot 20 = 4 \]
\[ B = \left( \frac{4}{5} \right) \cdot 20 = 16 \]
Linux CFS: Proportional Shares

Target Latency = 20ms
Minimum Granularity = 1ms
A timeslice = 4ms
B timeslice = 16 ms

Recall: Run the thread with the lowest amount of CPU use
**Linux CFS: Proportional Shares**

Target Latency = 20ms  
Minimum Granularity = 1ms  

A timeslice = 4ms  
B timeslice = 16 ms

Recall: Run the thread with the lowest amount of CPU use
Linux CFS: Proportional Shares

Target Latency = 20ms
Minimum Granularity = 1ms
A timeslice = 4ms
B timeslice = 16 ms

Recall: Run the thread with the lowest amount of CPU use
Linux CFS: Proportional Shares

Target Latency = 20ms
Minimum Granularity = 1ms
A timeslice = 4ms
B timeslice = 16 ms

Recall: Run the thread with the lowest amount of CPU use
Linux CFS: Proportional Shares

A and B got 50% of the CPU. Something went wrong!

Recall: Run the thread with the lowest amount of CPU use

A 16

B 16

A B A A A
Virtual Runtime

Must 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

Virtual Runtime = Virtual Runtime + \( \frac{1}{w_i} \) Physical Runtime
Linux CFS: Proportional Shares

Target Latency = 20ms
Minimum Granularity = 1ms
A timeslice = 4ms
B timeslice = 16 ms

Recall: Run the thread with the lowest amount of CPU use
Linux CFS: Proportional Shares

Target Latency = 20 ms
Minimum Granularity = 1 ms
A timeslice = 4 ms
B timeslice = 16 ms

Virtual Runtime = 0 + Physical Runtime / Weight = 0 + 4/1
Linux CFS: Proportional Shares

Target Latency = 20ms
Minimum Granularity = 1ms
A timeslice = 4ms
B timeslice = 16 ms

Virtual Runtime = 0 + Physical Runtime / Weight = 0 + 16/4 = 4
**Linux CFS: Proportional Shares**

**Target Latency** = 20ms  
**Minimum Granularity** = 1ms  
A timeslice = 4ms  
B timeslice = 16 ms

**Virtual Runtime** = 4 + Physical Runtime / Weight = 4 + 4/1 = 8
Linux CFS: Proportional Shares

Target Latency = 20 ms
Minimum Granularity = 1 ms
A timeslice = 4 ms
B timeslice = 16 ms

Virtual Runtime = 4 + Physical Runtime / Weight = 4 + 16/4 = 8
Linux CFS: Proportional Shares

A “Physical” CPU utilization: $4 + 4 = 8$

B “Physical” CPU utilization: $16 + 16 = 32$

But equal virtual runtime!
CFS shares vruntime equally
What about new jobs or very sleepy jobs?
Linux CFS: Proportional Shares

Reuse nice value to reflect share, rather than priority

CFS uses nice values to scale weights exponentially

\[ \text{Weight} = \frac{1024}{(1.25)^\text{nice}} \]
Weight the real running time with priority of the task

Nice 0 is the reference: vruntime == real runtime

Nice < 0: vruntime increases slower than real time

Nice > 0: vruntime increases faster than real time
**Summary: Schedulers in Linux**

- **O(n) scheduler**
  - Linux 2.4 to Linux 2.6
  - Did not scale with large number of processes

- **O(1) scheduler**
  - Linux 2.6 to 2.6.22
  - Heuristics too complex

- **CFS scheduler**
  - Linux 2.6.23 onwards
  - Proportional Fair Sharing. Throughput and Latency constraints
  - Gives all processes 1/N *virtual time* on CPU
Summary: Schedulers in Linux

O(n) scheduler
Linux 2.4 to Linux 2.6

O(1) scheduler
Linux 2.6 to 2.6.22

CFS scheduler
Linux 2.6.23 onwards

Did not scale with large number of processes
Heuristics too complex
Proportional Fair Sharing. Throughput and Latency constraints
Gives all processes 1/N * virtual time * on CPU
Understanding Deadlock

I will if you will

I will if you will
Deadlock: A Deadly type of Starvation

Deadlock: cyclic 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: A Deadly type of Starvation

**Starvation:** thread waits indefinitely

**Deadlock implies starvation**
**but starvation does not imply deadlock**

**Starvation can end** (but doesn’t have to)
**Deadlock can’t end without external intervention**
Example: Single-Lane Bridge Crossing
Bridge Crossing Example

Each segment of road can be viewed as a resource.

Rules:
- Car must own the segment under them
- Must acquire segment that they are moving into
- For bridge: traffic only in one direction at a time
Bridge Crossing Example

Car must own the segment under them
Must acquire segment that they are moving into
For bridge: traffic only in one direction at a time
Bridge Crossing Example

Deadlock:
Circular waiting for resources
**Bridge Crossing Example**

**Deadlock:**
Circular waiting for resources

Could be resolved by “external” intervention:
- fork-lifting a car of the bridge (equivalent to killing a thread)
  - Asking cars to backup (equivalent to removing the resource from the thread)
Starvation does not mean deadlock!

Stop sign: purple car must wait for cars to release resources.

Cars on highway never do!

Purple car is starved
Deadlock with Locks

Thread A:
- `x.Acquire();`
- `y.Acquire();`
- ...
- `y.Release();`
- `x.Release();`

Thread B:
- `y.Acquire();`
- `x.Acquire();`
- ...
- `x.Release();`
- `y.Release();`

Will threads deadlock:
- a) Always
- b) Never
- c) Sometimes
- d) I’m still trying to cross the road

This lock pattern exhibits *non-deterministic deadlock*

A system is subject to deadlock if deadlock can happen in any execution.
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!
Dining Computer Scientists Problem

Five chopsticks/Five computer scientists

Need two chopsticks to eat
Free for all leads to deadlock
Intervention needed

Fixing deadlock needs external intervention!

How could we have prevented this?

- Give everyone two chopsticks
- Make everyone “give up” after a while
- Require everyone to pick up both chopsticks atomically
Four requirements for occurrence of deadlock

1) Mutual exclusion and bounded resources
   Only one thread at a time can use a resource.

2) Hold and wait
   Thread holding at least one resource is waiting to acquire additional resources held by other threads.
Four requirements for occurrence of deadlock

3) No preemption
Resources are released only voluntarily by the thread holding the resource, after thread is finished with it.

4) Circular wait
There exists a set \{T_1, ..., 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 $T_1, T_2, \ldots, T_n$

Resource types $R_1, R_2, \ldots, R_m$

CPU cycles, memory space, I/O devices

Each resource type $R_i$ has $W_i$ instances

Each thread

Request() / Use() / Release() a resource:
Detecting Deadlock: Resource-Allocation Graph

Resource-Allocation Graph

- $V$ is partitioned into two types:

$$T = \{T_1, T_2, \ldots, T_n\},$$
the set threads in the system.

$$R = \{R_1, R_2, \ldots, R_m\},$$
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

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)

- $[\text{FreeResources}]$: Current free resources each type
- $[\text{Request}_X]$: Current requests from thread $X$
- $[\text{Alloc}_X]$: Current resources held by thread $X$
Deadlock Detection Algorithm

**See if tasks can eventually terminate on their own**

\[
\text{[Avail]} = \text{[FreeResources]}
\]

Add all threads to UNFINISHED

do {
    done = true
    Foreach thread in UNFINISHED {
        if \((\text{[Request}_{\text{node}}] \leq \text{[Avail]})\) {
            remove thread from UNFINISHED
            \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{\text{node}]}
            done = false
        }
    }
} until(done)

**Threads left in UNFINISHED \Rightarrow deadlocked**
Deadlock Detection Algorithm

\[
[A \text{vail}] = [\text{FreeResources}]
\]
Add all threads to UNFINISHED
\[\text{do} \quad \text{done} = \text{true} \quad \text{Foreach thread in UNFINISHED} \quad \text{if} \quad ([\text{Request}] \leq [\text{Avail}]) \quad \text{remove thread from UNFINISHED} \quad [\text{Avail}] = [\text{Avail}] + [\text{Alloc}] \quad \text{done} = \text{false} \quad \text{done} = \text{false} \quad \text{until}(\text{done}) \]

Threads left in UNFINISHED ⇒ deadlocked

\[
[\text{Avail}] = \{0,0\}
\]
UNFINISHED = T1, T2, T3, T4

Looking at T1: \([1,0] > [0,0]\)

Looking at T2: \([0,0] \leq [0,0]\)
Avail = \([1,0]\)
UNFINISHED = T1, T3, T4

Looking at T3: \([0,1] > [1,0]\)

Looking at T4
\([0,0] \leq [0,0]\)
Avail = \([1,1]\)
UNFINISHED = T1, T3

Looking at T1: \([1,0] \leq [1,1]\)
Avail = \([2,1]\)
UNFINISHED = T3

Looking at T3: \([0,1] \leq [2,1]\)
Avail = \([2,2]\)
UNFINISHED = Empty!
How should a system deal with deadlock?

**Deadlock prevention**
Write your code in a way that it isn’t prone to deadlock

**Deadlock recovery**
Let deadlock happen, and figure out how to recover from it

**Deadlock avoidance**
Dynamically delay resource requests so deadlock doesn’t happen

**Deadlock denial**
Ignore the possibility of deadlock
Deadlock prevention

Condition 1: Mutual exclusion and bounded resources
=> Provide sufficient resources

Condition 2: Hold and wait
⇒ Abort request or acquire requests atomically

Condition 3: No preemption
=> Preempt threads

Condition 4: Circular wait
=> Order resources and always acquire resources in the same way
**Condition 1 Fix: (Virtually) Infinite Resources**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>AllocateOrWait(1 MB)</td>
<td>AllocateOrWait(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
<tr>
<td>Free(1 MB)</td>
<td>Free(1 MB)</td>
</tr>
</tbody>
</table>

With virtual memory we have “infinite” space so everything will always succeed
Condition 2 Fix: Request Resources Atomically

Rather than:

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();
Condition 3 Fix: Preemption

Force thread to give up resource

Common technique in databases using database aborts
- A transaction is "aborted": all of its actions are undone, and the transaction must be retried

Common technique in wireless networks:
- Everyone speaks at once. When a resource collision is detected, retry at a new, random time
**Condition 4 Fix: Circular Waiting**

**Force all threads to request resources in the same order**

**Thread A:**

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

**Thread B:**

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

**Thread A:**

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

**Thread B:**

```plaintext
x Acquire();
y Acquire();
...
y.Release();
x.Release();
```
Condition 4 Fix: Circular Waiting

Garcia: first 1 then 5
Crooks: first 2 then 1
Turing: first 3 then 2
Johnson: first 4 than 3
Liskov: first 5 then 4

If ensure that Garcia graphs chopstick 5 followed by 1, no deadlock!
How should a system deal with deadlock?

**Deadlock prevention**
Write your code in a way that it isn’t prone to deadlock

**Deadlock recovery**
Let deadlock happen, and figure out how to recover from it

**Deadlock avoidance**
Dynamically delay resource requests so deadlock doesn’t happen

**Deadlock denial**
Ignore the possibility of deadlock
Techniques for Deadlock Avoidance

Attempt 1

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.
Techniques for Deadlock Avoidance

This does not work!

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

- Wait?
- But it’s already 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

*Deadlock avoidance: prevent system from reaching an unsafe state*
Deadlock Avoidance: Three States

Thread A:
- x.Acquire();
- y.Acquire();
- ...
- y.Release();
- x.Release();

Thread B:
- y.Acquire();
- x.Acquire();
- ...
- x.Release();
- y.Release();

A acquires x.

There exists a sequence A-A(y), A-R(y), A-R(x), B-A(y), B-A(x), B-R(x), B-R(y) => safe state

B acquires y.

No sequence that won’t lead to deadlock. => unsafe state
Banker’s Algorithm for Avoiding Deadlock

Banker’s algorithm ensures never enter an unsafe state.

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.
Banker’s Algorithm for Avoiding Deadlock

\[ \text{[Avail]} = \text{[FreeResources]} \]
Add all threads to UNFINISHED

do {
  done = true
  Foreach thread in UNFINISHED {
    if (\text{[Request}_{\text{thread}} \leq \text{[Avail]}]) {
      remove thread from UNFINISHED
      \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{\text{thread}}]
      done = false
    }
  }
  done = false
}while (!done)

\[ \text{[Avail]} = \text{[FreeResources]} \]
Add all threads to UNFINISHED

do {
  done = true
  Foreach threads in UNFINISHED {
    if (\text{[Max}_{\text{threads}}]-\text{[Alloc}_{\text{thread}}] \leq \text{[Avail]}) {
      remove thread from UNFINISHED
      \text{[Avail]} = \text{[Avail]} + \text{[Alloc}_{\text{thread}}]
      done = false
    }
  }
  done = false
}while (!done)
Banker’s Algorithm for Avoiding Deadlock

Step 1: “Assume” request is made

Step 2: If request is made, is system still in SAFE state? There exists a sequence \( \{T_1, T_2, \ldots, T_n\} \) such that all transactions finish

Step 3: If SAFE, grant resources. If UNSAFE, delay
Banker's Algorithm for Avoiding Deadlock

\[ \text{[Avail]} = \text{[FreeResources]} \]
Add all threads to UNFINISHED
\[
\text{do } \{
\text{done} = \text{true} \\
\text{Foreach threads in UNFINISHED } \{
\text{if } ([\text{Maxthreads}] - [\text{Alloc}_{\text{thread}}]) \leq [\text{Avail}] ) \{ \\
\text{remove thread from UNFINISHED} \\
[\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{\text{thread}}] \\
\text{done} = \text{false}
\} \\
\}
\} \text{ until(done)}

\text{Thread A:}
\begin{align*}
\text{x.Acquire();} \\
\text{y.Acquire();} \\
\text{...} \\
\text{y.Release();} \\
\text{x.Release();}
\end{align*}

\text{Thread B:}
\begin{align*}
\text{y.Acquire();} \\
\text{x.Acquire();} \\
\text{...} \\
\text{x.Release();} \\
\text{y.Release();}
\end{align*}

\text{When Thread A acquires x:}
\text{Avail} = [0,1] \\
\text{For A: } [1,1] - [1,0] \leq [0,1] \\
\text{Update Avail to } = [1,1]. \\
\text{Remove A from UNFINISHED} \\
\text{For B: } \\
[1,1] - [0,0] \leq [1,1] \\
\text{Update Avail to } = [1,1]. \\
\text{Remove B from UNFINISHED} \\
\text{Safe state!}

\text{When Thread B acquires y:}
\text{Avail} = [0,0] \\
\text{For A: } [1,1] - [1,0] \leq [0,0] \\
\text{For B: } [1,1] - [0,1] \leq [0,0] \\
\text{UNFINISHED not empty} \\
\text{Unsafe state! Must delay acquiring y!}
Summary

Deadlock => Starvation, Starvation does not imply deadlock

Four conditions for deadlocks
Mutual exclusion
Hold and wait
No preemption
Circular wait

Techniques for addressing deadlock: prevention, recovery, avoidance, or denial

Banker’s algorithm for avoiding deadlock