**Lunch on the Glade!**

**Friday 7th October, Noon**

**Prof Garcia (61C), Yan (61C), Crooks (162)** will be having lunch on the glade.

**Bring your own lunch and come join us!**
Recall: STCF

Schedule jobs in order of shortest completion time

Non-Preemptible

Requires knowledge of job completion time

Subject to Starvation

Time-Slicing

Approximate duration of CPU burst; encode it in priorities

Dynamically adapt priorities
Recall: Multi Level Feedback Queue

Rule 1
If Priority(A) > Priority(B), A runs (B doesn’t).

Rule 2
If Priority(A) = Priority(B), A & B run RR using quantum of queue.

Rule 3
A new job is placed in the topmost queue.

Rule 4
If a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.

Rule 5
After some time period S, move all the jobs in the system to the topmost queue.
Recall: Learning behaviour

- **P₁** Computes for 1 ms. Uses disk for 10 ms
- **P₂** Computes for 50 ms.

Schedule:

- \( q = 2 \text{ ms} \):
  - **P₁**

- \( q = 10 \text{ ms} \):
  - **P₁**

- \( q = 100 \text{ ms} \):
  - **P₂**

Timeline:

- **P₁**
- **P₁**
- **P₂**
- **P₁**
- **P₂**
Goals for Today

• What did “older” Linux schedulers do?

• Introducing the concept of proportional fair sharing and CFS

• Understanding deadlocks more formally
Recall: History of 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
At every context switch:
- Scan full list of processes in the ready queue
- Compute relevant priorities
- Select the best process to run

Scalability issues:
- Context switch cost increases as number of processes increase
- Single queue even in multicore systems
Case Study: Linux $O(1)$ Scheduler

Next process to run is chosen in constant time.

Priority-based scheduler with 140 different priorities.

Real-time/kernel tasks assigned priorities 0 to 99 (0 is highest priority).

User tasks (interactive/batch) assigned priorities 100 to 139 (100 is highest priority).
Case Study: O(1) Scheduler – User tasks

Per priority-level, each CPU has two ready queues

An active queue, for processes which have not used up their time quanta

An expired queue, for processes who have

Timeslices/priorities/interactivity credits all computed when jobs finishes timeslice

Timeslice depends on priority
User tasks – Priority Adjustment

User-task priority adjusted ±5 based on heuristics
- \( p \rightarrow \text{sleep\_avg} = \text{sleep\_time} - \text{run\_time} \)
- Higher sleep\_avg \( \Rightarrow \) more I/O bound the task, more reward (and vice versa)

Interactive Credit
- Earned when a task sleeps for a “long” time
- Spend when a task runs for a “long” time
- IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior

However, “interactive tasks” get special dispensation
- To try to maintain interactivity
- Placed back into active queue, unless some other task has been starved for too long...
O(1) Scheduler – Real tasks

Real-Time Tasks always preempt non-RT tasks

No dynamic adjustment of priorities

Scheduling schemes:
» SCHED_FIFO: preempts other tasks, no timeslice limit
» SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority
An aside: Real-Time Scheduling

Goal

Predictability of Performance!

We need to predict with confidence worst case response times for systems!

Real-time is about enforcing predictability, and does not equal fast computing.
Introducing the Completely Fair Scheduler

Key idea:

Proportional Fair Sharing

Give each job a share of the CPU according to its priority
Proportional Fair Sharing

Share the CPU proportionally

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)
Early Example: Lottery Scheduling

Give each job some number of lottery tickets

On each time slice, randomly pick a winning ticket

Each job gets at least one ticket

On average, CPU time is proportional to number of tickets given to each job
How to assign tickets?

Give Job A 50% of CPU, Job B 25%, Job C 10%

How can we use tickets to allow IO/interactive tasks to run quickly?
Assign tasks more tickets!

Can lottery scheduling lead to starvation?
   a) Yes  b) No

Can lottery scheduling lead to priority inversion?
**Temporary Unfairness**

Lose control over which job gets scheduled next.

Can suffer temporary bouts of unfairness

Given two jobs A and B of same run time (#Qs) that are each supposed to receive 50%,

\[ U \] = finish time of first / finish time of last

As a function of run time

Figure 9.2: Lottery Fairness Study
Stride Scheduling

Deterministic proportional fair sharing

Stride of each job is $\frac{\text{big} \cdot W}{N_i}$

The larger your share of tickets $N_i$, the smaller your stride

$W = 10,000,$

A = 100 tickets, B = 50, C = 250

A stride: 100, B: 200, C: 40
Stride Scheduling

Each job as a pass counter.

Scheduler picks a job with lowest pass, runs it, add its stride to its pass

Low-stride jobs (lots of tickets) run more often
Stride Scheduling

\( W = 10,000, \)
\( A = 200 \) tickets, \( B = 100 \) tickets, \( C = 50 \) tickets

Strides:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>200</td>
</tr>
</tbody>
</table>

Schedule

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>150</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>200</td>
</tr>
</tbody>
</table>

Ready Queue

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>150</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>300</td>
</tr>
</tbody>
</table>
Linux Completely Fair Scheduler (CFS)

CFS models an “ideal, precise multi-tasking CPU”

Each process gets an equal share of CPU

$N$ threads “simultaneously” execute on $\frac{1}{N}$ of CPU

Model: “Perfectly” subdivided CPU:

Each thread gets $\frac{1}{N}$ of the cycles

Optimise a global metric, not a local decision
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
**Linux CFS: Responsiveness**

**Constraint 1: Target Latency**

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
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
/**
 * Targeted preemption latency for CPU-bound tasks:
 * 
 * NOTE: this latency value is not the same as the concept of
 * 'timeslice length' - timeslices in CFS are of variable length
 * and have no persistent notion like in traditional, time-slice
 * based scheduling concepts.
 * 
 * (to see the precise effective timeslice length of your workload,
 * run vmstat and monitor the context-switches (cs) field)
 * 
 * (default: 6ms * (1 + ilog(ncpus)), units: nanoseconds)
 */

unsigned int syscall_sched_latency = 6000000ULL;
static unsigned int normalized_syscall_sched_latency = 6000000ULL;

/* Minimal preemption granularity for CPU-bound tasks: */

(unsigned long long) (1 + ilog(ncpus)), units: nanoseconds)

unsigned int syscall_sched_min_granularity = 750000ULL;
static unsigned int normalized_syscall_sched_min_granularity = 750000ULL;
Priorities in Unix

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

Easy to implement for \(O(1)\) scheduler, how does it work for CFS?

We want to implement proportional fair sharing
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

Reuse nice value to reflect share, rather than priority

CFS uses nice values to scale weights exponentially

\[ \text{Weight} = \frac{1024}{(1.25)^{\text{nice}}} \]
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: $Q_i = \left(\frac{w_i}{\sum w_p}\right) \cdot \text{Target Latency}$

$A = \left(\frac{1}{5}\right) \cdot 20 = 4$

$A = \left(\frac{4}{5}\right) \cdot 20 = 16$
**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
A and B got 50% of the CPU. Something went wrong!
**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

$$\text{Virtual Runtime} = \text{Virtual Runtime} + \left( \frac{\Sigma p w_p}{w_i} \right) \text{Physical Runtime}$$
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 = 20ms
Minimum Granularity = 1ms
A timeslice = 4ms
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 = 20\,\text{ms}
Minimum Granularity = 1\,\text{ms}
A timeslice = 4\,\text{ms}
B timeslice = 16\,\text{ms}

Virtual Runtime = 4 + \frac{\text{Physical Runtime}}{\text{Weight}} = 4 + \frac{4}{1} = 8
**Linux CFS: Proportional Shares**

**Target Latency = 20ms**
**Minimum Granularity = 1ms**

A timeslice = 4ms
B timeslice = 16 ms

Virtual Runtime = \(4 + \frac{\text{Physical Runtime}}{\text{Weight}}\) = 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
Linux CFS: Proportional Shares

Physical CPU Time

16 \ (w_B=4) \hspace{1cm} \text{B}

4 \ (w_A=1) \hspace{1cm} \text{A}

Virtual CPU Time

\text{B} \hspace{1cm} \text{A}
Summary: Schedulers in Linux

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

- **O(1) scheduler**
  - Linux 2.6 to 2.6.22
  - Proportional Fair Sharing (PFS)
  - Throughput and latency constraints
  - Gives all processes 1/N virtual time on CPU

- **CFS scheduler**
  - Linux 2.6.23 onwards
Goal 3: 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
Deadlock:

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

Cars on highway never do!

Purple car is starved
## Deadlock with Locks

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x.Acquire();</code></td>
<td><code>y.Acquire();</code></td>
</tr>
<tr>
<td><code>y.Acquire();</code></td>
<td><code>x.Acquire();</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>y.Release();</code></td>
<td><code>x.Release();</code></td>
</tr>
<tr>
<td><code>x.Release();</code></td>
<td><code>y.Release();</code></td>
</tr>
</tbody>
</table>

Will threads deadlock

a) Always  b) Never  c) Sometimes  d) I’m still trying to cross the road

This lock pattern exhibits \textit{non-deterministic deadlock}

A system is subject to deadlock if deadlock can happen \textit{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!
Five chopsticks/Five computer scientists

Need two chopsticks to eat
Free for all leads to deadlock
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 \( \{ T_1, \ldots, T_n \} \) 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 $R_j \rightarrow T_i$

Symbols

$T_1$ $T_2$
$R_1$ $R_2$
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 $X$

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

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

Add all threads to UNFINISHED

do {
    done = true
    Foreach thread in UNFINISHED {
        if ([Request_{node}] <= [Avail]) {
            remove thread from UNFINISHED
            [Avail] = [Avail] + [Alloc_{node}]
            done = false
        }
    }
} until(done)

Threads left in UNFINISHED ⇒ deadlocked

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

Looking at T1: [1,0] > [0,0]
Looking at T2: [0,0] <= [0,0]
Avail = [1,0]
UNFINISHED = T1, T3, T4

Looking at T3: [0,1] > [1,0]
Looking at T4 [0,0] <= [0,0]
Avail = [1,1]
UNFINISHED = T1, T3

Looking at T1: [1,0] <= [1,1]
Avail = [2,1]
UNFINISHED = T3

Looking at T3: [0,1] <= [2,1]
Avail = [2,2]
UNFINISHED = Empty!