CS162
Operating Systems and Systems Programming
Lecture 12

Scheduling
Core Concepts and Classic Policies

Professor Natacha Crooks
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi
Recall: STCF

Schedule jobs in order of shortest completion time

Requires knowledge of job completion time

Subject to Starvation

Approximate duration of CPU burst; encode it in priorities

Dynamically adapt priorities
Recall: Multi Level Feedback Queue

Rule 1
If \text{Priority}(A) > \text{Priority}(B), A \text{ runs} (B \text{ doesn’t}).

Rule 2
If \text{Priority}(A) = \text{Priority}(B), A \& B \text{ 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: 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

$P_1$ Computes for 1 ms. Uses disk for 10 ms

$P_2$ Computes for 50 ms.

$q = 2$ ms

$q = 10$ ms

$q = 100$ ms

Schedule: $P_1$, $P_1$, $P_2$, $P_1$, $P_2$
Many many different variants of MLQF

Change how prevent starvation

Change constants

Change scheduling policies within each queue

Most modern schedulers are variants of MLQF queues
What’s important?

**IO-bound jobs have high priorities.** Get scheduled quickly. Run for short quantas.

**Compute-bound jobs have low priority.** Run with low time quantas. Run when IO bound jobs blocked on IO.

To prevent starvation, all jobs get a chance to run in a given period $S$.

No job says in the lower queue for ever. Account for changes in workload.
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**
Case Study: Linux $O(n)$ Scheduler

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->sleep_avg = sleep_time – run_time
» Higher sleep_avg ⇒ 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...
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%,

\[ U = \text{finish time of first} / \text{finish time of last} \]

As a function of run time

Figure 9.2: Lottery Fairness Study
**Stride Scheduling**

**Deterministic proportional fair sharing**

**Stride** of each job is \( \frac{\text{big} \# W}{N_i} \)

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

\[ W = 10,000, \]
\[ A = 200 \text{ tickets, } B = 100 \text{ tickets, } C = 50 \text{ tickets} \]

Strides:

\[ 50, 100, 200 \]

Schedule:

\[ 50, 100, 100, 150, 200, 200, 200, 200 \]

Ready Queue:

\[ 50, 100, 100, 150, 200, 200, 200, 200, 250, 250, 300, 200 \]
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.
Linux CFS: Responsiveness

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

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
**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 sysctl_sched_latency
    = 6000000ULL;
static unsigned int normalized_sysctl_sched_latency
    = 6000000ULL;
```

```
/*
 * Minimal preemption granularity for CPU-bound tasks:
 * 
 * (default: 0.75 msec "(1 + ilog(ncpus)), units: nanoseconds)
 */

unsigned int sysctl_sched_min_granularity
    = 7500000ULL;
static unsigned int normalized_sysctl_sched_min_granularity
    = 7500000ULL;
```
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
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:** $Q_i = \left(\frac{w_i}{\sum_p w_p}\right) \cdot \text{Target Latency}$
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 = 20 ms
Minimum Granularity = 1 ms

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

\[
A = \left(\frac{1}{5}\right) \cdot 20 = 4 \\
B = \left(\frac{4}{5}\right) \cdot 20 = 16
\]
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

A 0
B 0
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
B

16
16
**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 + \((1/w_i)\) 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** = 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 = 20ms
Minimum Granularity = 1ms

A timeslice = 4ms
B timeslice = 16ms

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

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

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
Linux CFS: Proportional Shares

Physical CPU Time

16 (\(w_B=4\))

B

4 (\(w_A=1\))

A

Virtual CPU Time

B

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
  - Throughput and Latency constraints
  - Gives all processes $1/N \times$ virtual time * on CPU

- **CFS scheduler**
  - Linux 2.6.23 onwards