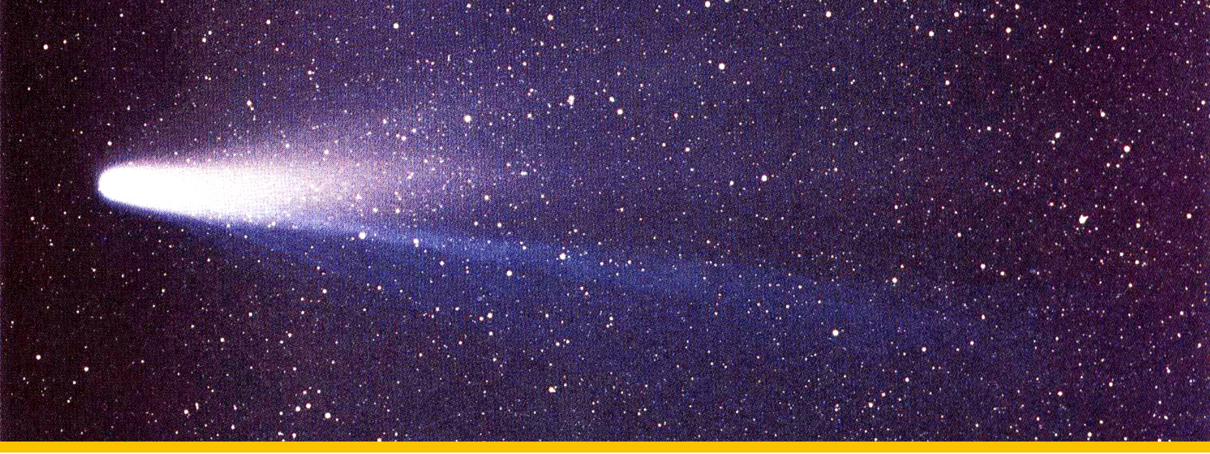


BTI1341 – Operating Systems

Part 1: Virtualization – 2) Scheduling

Autumn 2025-26 `master@352a46f (20250901-155000)`

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Outline

- Basic Scheduling
- Multi-Level Feedback Queues
- Proportional Share Scheduling
- Linux Scheduling
- Multiprocessor Scheduling
- Appendix

Basic Scheduling

Introduction

We understand the basic *mechanisms* used by the OS for process switching:

- Limited direct execution
- Timer interrupts

But when and why does an OS switch processes?

- Such decisions are part of scheduling, the responsible OS component is the scheduler
- OSes use different strategies or policies (also called disciplines) for scheduling
- Optimal scheduling can be quite complicated

Scheduling Policies: Assumptions

We will now evaluate different scheduling policies. To do so, we will make the following (unrealistic!) assumptions for now:

1. All jobs[1] require the *same amount of time to run*
2. All jobs *arrive* at the same time
3. When running, jobs are *not interrupted* until finished
4. We know exactly *how long* each job has to run for completion
5. They perform only work on the CPU, *no I/O*

---

[1] A process is often called a job in scheduling.

Scheduling Metric 1: $T_{\text{turnaround}}$

Often, when evaluating things, we need a metric. Let us define our first scheduling metric:

Definition

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

The turnaround time of a job is the time when it completes minus the time at which it arrived.

Note: For now, $T_{\text{arrival}} = 0$, thus $T_{\text{turnaround}} = T_{\text{completion}}$ (*Assumption 2*).

Scheduling Policy 1: FIFO

FIFO scheduling means: *First In, First Out* (sometimes also FCFS: *First Come, First Served*).

Example: All jobs take 10 secs, job `A` randomly run first.

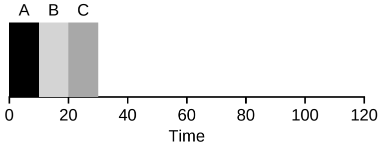


Figure: A FIFO Scheduling Example

Courtesy of [ADAD18]

$T_{\text{turnaround}}$ for `A`, `B` and `C`: 10, 20, 30.

Average: $\frac{10 + 20 + 30}{3} = 20$

The Problem with FIFO

If we relax *Assumption 1* (all jobs require the same amount of time), FIFO runs into trouble (convoy effect , [BGMP79]):

Example: Job `A` now takes 100 secs.

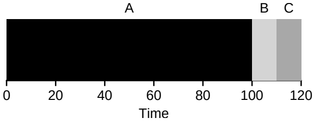


Figure: The Issue with FIFO

Courtesy of [ADAD18]

Job	Arrival	Duration
A	0	100
B	0	10
C	0	10

$T_{\text{turnaround}}$ for `A`, `B` and `C`: 100, 110, 120.

Average: $\dfrac{100 + 110 + 120}{3} = 110$

Scheduling Policy 2: SJF

Without further relaxing assumptions, a simple idea solves the convoy problem: SJF or *Shortest Job First* scheduling:

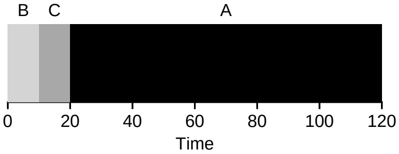


Figure: A SJF Scheduling Example

Courtesy of [ADAD18]

$T_{\text{turnaround}}$ for `A`, `B` and `C`: **120**, 10, 20.

Average: $$\frac{120+10+20}{3} = 50$$

The Problem with SJF

If and only if *Assumption 2* (all jobs arrive at the same time) holds, *SJF can be proven optimal*. As this is not realistic, we drop *Assumption 2*:

Example: Jobs `B` and `C` now arrive late.

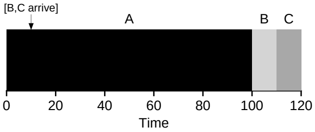


Figure: SJF with Late Arrival

Courtesy of [ADAD18]

Job	Arrival	Duration
A	0	100
B	10	10
C	10	10

$T_{\text{turnaround}}$ for `A`, `B` and `C`: 100, 100 $(110 - 10)$, 110 $(120 - 10)$.

Average: $\frac{100 + 100 + 110}{3} = 103.33$

Interlude: Preemptive Scheduling

So far, we have assumed that the scheduler may not interrupt a running job (*Assumption 3*). To develop better scheduling policies, we need to drop this assumption.

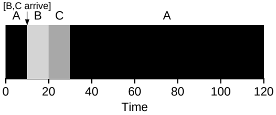
Definition

A preemptive scheduler is a scheduler which can interrupt a running job. To do so, it uses the mechanisms we have introduced earlier.

Scheduling Policy 3: STCF

Without *Assumption 3*, jobs may be interrupted any time. Using this, we find the STCF (*Shortest Time-to-Completion First*) policy:

Example: When jobs `B` and `C` arrive, `A` is preempted.



Job	Arrival	Duration
A	0	100
B	10	10
C	10	10

Figure: A STCF Scheduling Example

Courtesy of [ADAD18]

$T_{\text{turnaround}}$ for `A`, `B` and `C`: 120, 10 $(20 - 10)$, 20 $(30 - 10)$.

Average: $$\frac{120 + 10 + 20}{3} = 50$$

Scheduling Metric 2: T_{response}

If we could rely on *Assumption 4* (knowing how long a job takes), STCF would be a great policy. However:

- In reality, we only rarely know the job duration
- Nowadays, systems are expected to be *interactive*

Thus, for *general purpose* OSes,[2] a different metric becomes important as well:

Definition

$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$

The response time of a job is the difference between the time it is first scheduled and the time at which it arrived.

---

[2] There are also specialized OSes for *batch*- and *realtime* processing.

Revisiting STCF

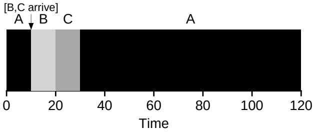


Figure: STCF Again

Job	Arrival	Duration
A	0	100
B	10	10
C	10	10

T_{response}: $\texttt{A} = 0$, $\texttt{B} = 0$, $\texttt{C} = 10$, Average: 3.33.

What happens when multiple jobs arrive at the same time? What is the problem with STCF?

Scheduling Policy 4: Round Robin

Simple idea: do not complete jobs but run them for a time slice (or scheduling quantum). Time slices are multiples of the timer interrupt.

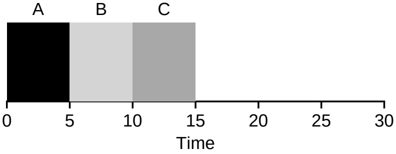


Figure: SJF Again

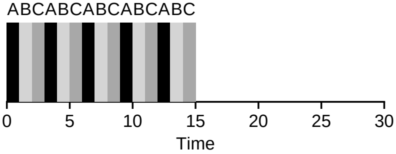


Figure: Round Robin Scheduling

Courtesy of [ADAD18]

Scheduling Policy 4: Round Robin (cont.)

Example (previous slide): Jobs `A`, `B` and `C` arrive at time 0 and run for 5 secs each.

Metrics for SJF scheduling:
$T_{\text{turnaround}}$: $\texttt{A} = 5$, $\texttt{B} = 10$, $\texttt{C} = 15$, average: 10.
T_{response}: $\texttt{A} = 0$, $\texttt{B} = 5$, $\texttt{C} = 10$, average: 5.

Metrics for round robin scheduling:
$T_{\text{turnaround}}$: $\texttt{A} = 13$, $\texttt{B} = 14$, $\texttt{C} = 15$, average: 14.
T_{response}: $\texttt{A} = 0$, $\texttt{B} = 1$, $\texttt{C} = 2$, average: 1.

Amortization

For round robin, the length of the time slice is relevant:

- Responsiveness becomes better, the shorter the time slice
- But: shorter time slices lead to increased context-switching overhead

Amortization helps in solving this fundamental tradeoff.

Example: Assuming cost for a context-switch is 1 ms.

- If length of time slice is 10 ms, 10% of the time are spent in context switches
- Increasing time slice to 100 ms: reduces overhead to $\sim 1\%$

Considering I/O

Finally, programs which do not perform any I/O at all seldom exist in practice. We must drop *Assumption 5*. During I/O, a job is *blocked* and cannot use the CPU. Thus:

- The scheduler must schedule a different job when I/O starts
- When I/O finishes, the scheduler must again decide about scheduling
  - The first job
  - The currently running job
  - A different job

STCF I/O Example I

For this example, assume two jobs, `A` and `B`, arriving at the same time and requiring 50 ms of CPU time each. `A` makes an I/O request every 10 ms which takes 10 ms to complete.

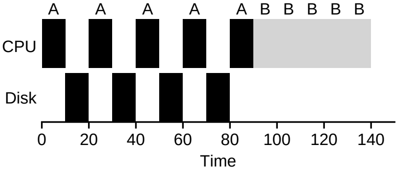


Figure: STCF Waiting for I/O

Courtesy of [ADAD18]

$T_{\text{turnaround}}$: $\texttt{A} = 90$, $\texttt{B} = 140$, average: 115.
T_{response}: $\texttt{A} = 0$, $\texttt{B} = 90$, average: 45.

STCF I/O Example II

Solution: Treat CPU usage of `A` as individual sub-jobs. At start, the STCF scheduler then has the choice to run `A` with 10 ms or `B` with 50 ms job duration.

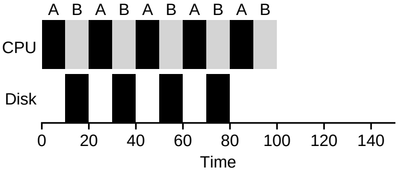


Figure: STCF with Overlapping

Courtesy of [ADAD18]

$T_{\text{turnaround}}$: $\texttt{A} = 90$, $\texttt{B} = 100$, average: 95.
T_{response}: $\texttt{A} = 0$, $\texttt{B} = 10$, average: 5.

Multi-Level Feedback Queues

Recap

Until now, we have made some observations regarding scheduling:

- Nothing is known about *arrival time* or *duration* of a job
- Achieving good *turnaround- and response time* simultaneously is desired but hard in practice
  - STCF would be optimal, if job duration would be known
  - Round robin is good for interactivity but terrible for turnaround time
- We have different types of *workload*: Batch (i.e. long running, non-interactive) and interactive jobs
  - It is unknown (at least so far…) to which type a job belongs

Motivating Multi-Level (Feedback) Queue Scheduling

MLFQ scheduling tries to optimize *turnaround- and response time* at the same time. For this, two main ideas are applied:

1. Use more than one queue for scheduling
2. Observe the behavior of a job and adjust its priority continuously

Using more than one queue enables a classification of jobs using **priorities**.

Observing a process gives information about its runtime behavior: Is it using only the CPU? Does it perform a lot of I/O? This helps in adjusting priority. *Learn from the past to predict the future.*

MLFQ Example

In this example, there are 4 jobs: `A` and `B` (high prio) in queue Q8, `C` (medium prio) in queue Q4 and `D` (low prio) in queue Q1. Queue numbering is not relevant.

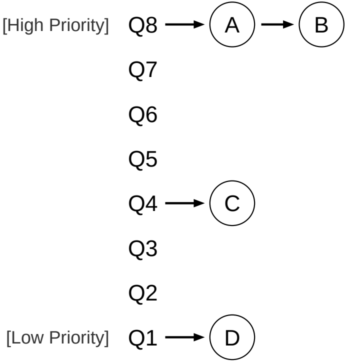


Figure: Example for MLFQ with 8 Queues

Courtesy of [ADAD18]

Basic Rules

In the following, we assume these basic rules when discussing MLFQ scheduling:

- There is a number of *distinct queues*, each with a different priority
- A job can only be in a single queue at any time
- There can be more than one job per queue; these are scheduled using round robin
- *Ready jobs* in queues with higher priority are run first

In summary:

Rule 1 If Prio(`A`) > Prio(`B`): Run `A`

Rule 2 If Prio(`A`) = Prio(`B`): Run `A` and `B` in round robin

Key Question: Adjusting Priority

MLFQ adjusts the priority of a job due to its observed behavior:

- A job performing a *lot of I/O* gets a *high* priority
- A job using the *CPU* a lot gets a *low* priority

Let us add some rules for this:

Rule 3	A new job is placed in the queue with the highest priority
Rule 4a	If it uses up its whole time slice, its priority is *reduced*
Rule 4b	If it yields the CPU before using up the time slice, its priority *stays the same*

Example: Batch Job

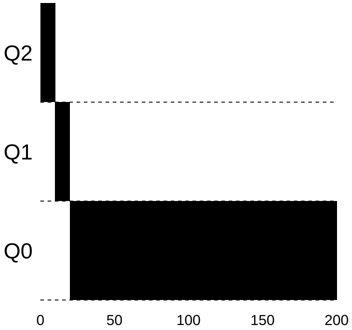


Figure: MLFQ Example for a Single Batch Job

Courtesy of [ADAD18]

Example: Batch and Interactive Job

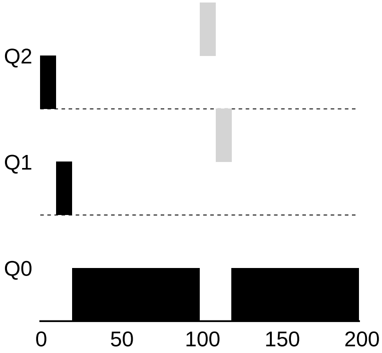


Figure: MLFQ Example for Batch- and Interactive Jobs

Courtesy of [ADAD18]

Notice: MLFQ first assumes a job to be short. If it is, it completes quickly – if not, it will move down the queues. **Thus, MLFQ approximates SJF!**

Example: Batch and I/O Jobs

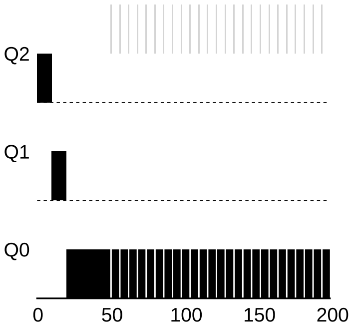


Figure: MLFQ Example Batch- and I/O Intensive Job

Courtesy of [ADAD18]

Due to Rule 4b, the I/O intensive job keeps its high priority (and thus its interactivity).

Problem 1: Starvation

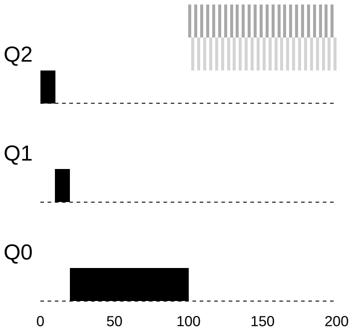


Figure: MLFQ Starvation

Courtesy of [ADAD18]

Too many interactive jobs may starve a batch job.
Or: a batch job might change behavior and become interactive (again)...

Solution: Priority Boosts

A simple solution for starvation is to periodically boost the priority of all jobs:

Rule 5 After a given time period, move all jobs to the queue with the highest priority

This solves two problems at once:

- No starvation: Every job periodically runs in the queue with the highest priority
- Behavior change: A batch job can become interactive again

Example: Priority Boosts

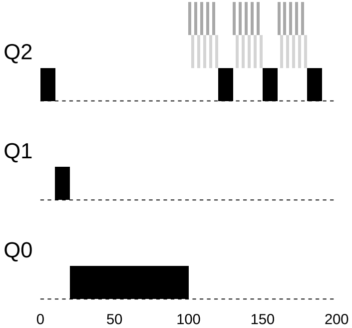


Figure: MLFQ with Priority Boosts

Courtesy of [ADAD18]

The batch job is moved to Q2 due to periodic priority boosts.

Problem 2: Gaming the Scheduler

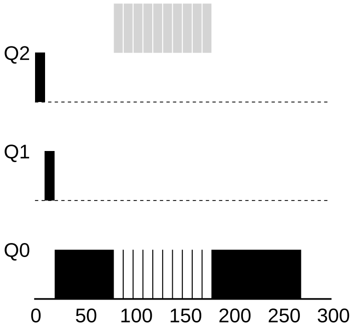


Figure: Gaming MLFQ

Courtesy of [ADAD18]

Gaming is an attack on the scheduler, in which a job cleverly yields its time slice to gain a lot of total CPU time. When could this be a problem?

Solution: Better CPU Accounting

Rules 4a and 4b enable gaming of the scheduler. The solution is better **accounting**: Track the CPU time spent over multiple context switches and move a job to the next priority queue if it has used up all assigned time.

We thus change the rules:

~~Rule 4a~~ ~~If it uses up its whole time slice, its priority is *reduced*~~

~~Rule 4b~~ ~~If it yields the CPU before using up the time slice, its priority *stays the same*~~

Rule 4 When a job uses up all its assigned time at a given priority (regardless how often it has yielded the CPU), it is moved to the next lower priority

Example: CPU Accounting

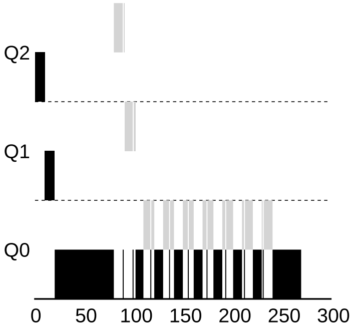


Figure: MLFQ with Gaming Tolerance

Courtesy of [ADAD18]

The gaming job is moved to Q0 due to better CPU accounting.

MLFQ Parametrization

MLFQ is an advanced scheduling policy which improves turnaround- and response time. However, for practical implementation, many questions must be solved:

- How many queues?
- How long are the time slices? Are they different per queue?
- At which interval should priority boosts occur?
    - If too long, jobs may starve
    - If too short, response time may degrade
- Are all jobs run in all queues? Are some queues reserved for the OS?
- Can the user influence scheduling decisions?

Proportional Share Scheduling

Proportional Shares: Basic Idea

Different idea: Do not optimize for turnaround or response time, but try to guarantee a *certain amount of CPU time* for each job. This is called proportional-share or fair-share scheduling.

One solution: Measure CPU time per job and distribute it over all running jobs. →Difficult to implement.

Another idea: *Use randomness!*[3] This is easier to implement (needs almost no state) and fast.

---

[3] ☞ Using randomness is often a good solution – keep it in mind!

Lottery Scheduling

In lottery scheduling , each job has a *certain amount of tickets*. The percent of tickets a job has, represents its share of CPU time. Tickets are numbered. Periodically (e.g. every time slice), a ticket number is drawn at random and the job holding the ticket is scheduled.

Example:

Job `A` has tickets $0 \ldots 74$ and job `B` tickets $75 \ldots 99$.
The scheduler draws the following numbers:

```
63  85  70  39  76  17  29  41  36  39  10  99  68  83  63  62  49  49
```

This corresponds to the following schedule:

```
63  85  70  39  76  17  29  41  36  39  10  99  68  83  63  62  49  49
A       A   A       A   A   A   A   A   A       A       A   A   A   A
    B           B                           B       B
```

Implementing Lottery Scheduling

```
int counter             = 0;
int winner              = random() % totaltickets; // get winner
struct node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
      break; // found the winner
    current = current->next;
}

// current is the winner: schedule it...
```

Source: `ostep-code/cpu-sched-lottery/lottery.c`

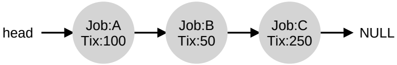


Figure: Lottery Implementation Using (Sorted) List

Courtesy of [ADAD18]

Lottery Fairness

Example: 2 jobs, 100 tickets each, *same* job length.

Unfairness Metric: $U = \dfrac{T_{\text{completion}}(A)}{T_{\text{completion}}(B)}$

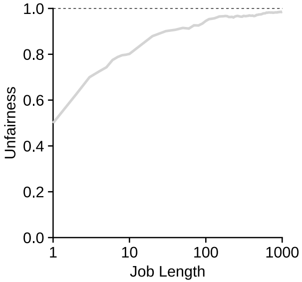


Figure: Fairness of Lottery Scheduling

Courtesy of [ADAD18]

★ Stride Scheduling

Stride scheduling is a *deterministic* ticket-based policy. Idea: Use inverse proportion of ticket shares to decide, which job to run. We define:

Stride $S(x) = \dfrac{C}{\text{Tickets}(x)}$

Where C is the *stride constant* (some large number) and $\text{Tickets}(x)$ the number of tickets a job x has

Pass $P(x)$ is the total *accumulated* stride of job x

The scheduler then simply runs the job with the *lowest* pass value and increments it with the job's stride.

Problem compared to lottery scheduling: Global state (what if a new job enters?)

★ Stride Example

$C = 10000$

Tickets per job: $\texttt{A} = 100$, $\texttt{B} = 50$, $\texttt{C} = 250$

Stride per job: $\texttt{A} = 100$, $\texttt{B} = 200$, $\texttt{C} = 40$

$P(A)$	$P(B)$	$P(C)$	Job run
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Linux Scheduling

Linux Completly Fair Scheduler

The Linux completly fair scheduler (CFS) is a highly efficient scheduler, trying to minimize overhead. It *has no traditional time slices* but adjusts them dynamically depending on the number of jobs. A good overview is given in [Jon].

Basic idea: virtual runtime (`vruntime`) is accumulated per job, the job with lowest `vruntime` is scheduled next.

Problem: When to schedule the next job? For this, CFS uses parameters and some clever weighting to decide.

CFS Basic Idea

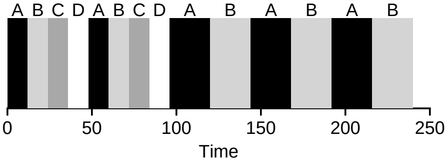


Figure: Completly Fair Scheduling, Basic Idea

Courtesy of [ADAD18]

CFS Parameters

The two most important parameters for CFS are: `sched_latency` and `min_granularity`.[4] (See: [linb],[linc])

- `sched_latency`: Time before considering a context switch

  Defaults to $6\text{ms} \cdot (1 + \log_2(\texttt{n\_cpus}))$.
  Example: 18ms.

- `min_granularity`: When there are many jobs, time slices get too small. This is the minimal value used in every case.

  Defaults to $0.75\text{ms} \cdot (1 + \log_2(\texttt{n\_cpus}))$.
  Example: 2.25ms.

---

[4] The current values (nanoseconds) for your machine can be found in `/sys/kernel/debug/sched/latency_ns` and `/sys/kernel/debug/sched/min_granularity_ns`!

CFS Weighting

CFS supports UNIX nice levels -20 (highest) to 19 (lowest) for modifying job priorities.[5] Instead of using priority queues, a weight value (see next slide) is applied for calculating the effective time slice of a job (k is job number, n is total job count):

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

Additionally, the weight of a job must also be considered when calculating **`vruntime`**:

$$\texttt{vruntime}_k = \texttt{vruntime}_k + \frac{\texttt{weight}_0}{\texttt{weight}_k} \cdot \texttt{curtime}_k$$

($\texttt{weight}_0$ is weight at priority 0, $\texttt{curtime}_k$ is the time the job has spent in the last time slice)

---

[5] See "`man nice`" for details.

CFS Weight Constants

```
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
const int sched_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,
};
```

Source: [lina]

Example

Assuming two jobs, `A` (nice level -5) and `B` (normal nice level, 0). Thus: $\texttt{weight}_A = 3121$ and $\texttt{weight}_B = 1024$. `sched_latency` is 18ms.

$$\texttt{time\_slice}_A = \frac{3121}{(3121 + 1024)} \cdot 18 \approx \frac{3}{4} \cdot 18 \approx 13.55\text{ms}$$

$$\texttt{time\_slice}_B = \frac{1024}{(3121 + 1024)} \cdot 18 \approx \frac{1}{4} \cdot 18 \approx 4.45\text{ms}$$

Note: An interesting property of the weights is that they *preserve proportionality*: If the nice levels would have been 5 and 10, the given jobs would have been scheduled in the same manner!

Jobs Sleeping or Waiting for I/O

There is an issue when simply choosing the process with the lowest `vruntime`: Jobs which are *sleeping* or *waiting for I/O* do not aggregate `vruntime`. Thus, when such a job wakes up, it would be scheduled for a long time in order to catch up!

CFS handles this by modifying `vruntime` when a job wakes up: it sets the value to the minimum value found for all jobs in the system. The same applies to new jobs, when they are scheduled for the first time.

Multiprocessor Scheduling

Introduction

So far, we have only looked at schedulling on a single CPU. With multiple CPUs (think todays multicore architectures), reality is much more complex. Here we provide only a short overview of multiprocessor scheduling to achieve a basic understanding.

Some of the main problems are:

- Issues due to CPU caches
  - Cache coherence
  - Cache affinity
- Synchronization issues, e.g. when all CPUs share a scheduling queue[6]
- Increased schedulling overhead
- ...

---

[6] Synchronization will be an important topic later in this course.

CPU Caches

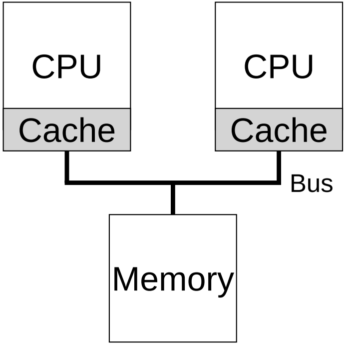


Figure: Two CPUs with Caches and Shared Memory

Courtesy of [ADAD18]

Note: In practice, multiple caches form a *hierarchy* of caches.

CPU Cache Issues

Cache Coherence : It must be ensured that all caches maintain the same state regarding a data item. E.g.

- An item is read/manipulated on CPU1 and stored in the local cache
- What if the same value is read or written on CPU2 (maybe later)?
- Caches need to either *update* or *invalidate* their state correctly

Cache Affinity : When a process runs on a CPU for some time, it builds up a lot of state in the cache. It will often make sense to *reschedule it on the same CPU* as otherwise performance may degrade.

Single- and Multi-Queue Scheduling

Scheduling all jobs for all CPUs in a *single queue* is possible. There are some issues however:

- Scalability/overhead: the queue requires synchronization
- Work required to maintain cache affinity

Another approach is to use *multiple queues*, e.g. one per CPU. This reduces synchronization overhead and fixes cache affinity, but:

- More complex implementation
- Introduces load imbalance (what if a CPU is done with all of its jobs?)

In practice, both approaches can be found.

Appendix

Bibliography I

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[BGMP79] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price, *The convoy phenomenon*, SIGOPS Oper. Syst. Rev. **13** (1979), no. 2, 20–25.

[Jon] M. Jones, *Inside the Linux 2.6 Completely Fair Scheduler*, `https://web.archive.org/web/20210420104226/https://developer.ibm.com/technologies/linux/tutorials/l-completely-fair-scheduler/`.

[lina] *Linux kernel, CFS core.c*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/sched/core.c`.

[linb] *Linux kernel, CFS documentation*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/scheduler/sched-design-CFS.rst`.

[linc] *Linux kernel, CFS fair.c*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/sched/fair.c`.