Linux process scheduling

General “neediness” categories

- realtime processes
  - whenever they demand attention, need it immediately
- other processes
  - interactive – care about responsiveness
    - demand no attention most of the time, don’t need it
    - demand it occasionally, need it immediately then
  - batch – don’t care about responsiveness
    - demand attention frequently, don’t need it immediately
General strategies

- favor all realtime processes ahead of all other processes
- favor interactive processes ahead of batch processes
  - by explicitly identifying and applying different formulas, or (pre-kernel-2.6.23 O(1) scheduler)
  - by applying a common formula (wait-time based) tending to float interactives and sink batches (current kernel 2.6.23+ CFS scheduler)

General scheduling basics

- multiple processes chosen to run for brief intervals one-after-the-other
- choice based on process “merit” or “deservedness”
- different possible “merit” characteristics
  - time a process has spent waiting (patience)
  - relative importance of a process (priority)
- linux considers several characteristics in combination
- always chooses the “most deserving” process
Patience may be meritorious

runqueue
(double-linked list of process descriptors)

task picked
to run

decreasing wait time

cpu

Priority may also be meritorious

task picked to run
(most patient among top priority)

cpu

A
B
C

decreasing wait time

decreasing priority
Five “scheduling classes”

FOR REALTIME PROCESSES
SCHED_FIFO (first-in first-out)
SCHED_RR (round robin)
SCHED_NORMAL a.k.a SCHED_OTHER
SCHED_BATCH
SCHED_IDLE

FOR "REGULAR" PROCESSES

Different schedulers

The “realtime scheduler”
SCHED_FIFO (first-in first-out)
SCHED_RR (round robin)
SCHED_NORMAL
SCHED_BATCH
SCHED_IDLE
The “completely fair scheduler (CFS)”

Priority scale

- **REALTIME**
- **REGULAR**
  - 0
  - 1
  - 99

Increasing priority

Realtime requirements

- low latency
- deterministic response time
- settings
  - financial trading
  - medical devices
  - defense
  - industrial automation
  - autonomous (self-driving) cars
Realtime trumps regular

Input to scheduling decisions
**Input to scheduling decisions**

**DESCRIPTION**
Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies:

- **SCHED_OTHER**: the standard round-robin time-sharing policy;
- **SCHED_BATCH**: for "batch" style execution of processes; and
- **SCHED_IDLE**: for running very low priority background jobs.

The following "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable processes are selected for execution:

- **SCHED_FIFO**: a first-in, first-out policy; and
- **SCHED_RR**: a round-robin policy.

**Scheduling Policies**

**SCHED_FIFO**: First In-First Out scheduling

SCHED_FIFO can only be used with static priorities higher than 0, which means that when a SCHED_FIFO process becomes runnable, it will always immediately preempt any currently running SCHED_OTHER, SCHED_BATCH, or SCHED_IDLE process.

**SCHED_OTHER**: Default Linux time-sharing scheduling

SCHED_OTHER can only be used at static priority 0. SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms.

**Scheduling class implementation**

Resembles object-oriented class hierarchy

Correct handler selected per scheduling class of each particular process

Extensible, for implementing future scheduling classes with new scheduling algorithms

"Completely Fair Scheduler,” Linux Journal, August 2009
Two demo programs

```c
[root@frausto ~]
[process_priority.c] $  process_priority-realtime-version.c

47, 4&47, 48

< sp.sched_priority=0;
< sched_setscheduler(0, SCHED_OTHER, &sp);
--
> sp.sched_priority=50;
> sched_setscheduler(0, SCHED_FIFO, &sp);

[root@frausto ~]
```

DESCRIPTION:

Currently, Linux supports the following "normal"
(i.e., non-real-time) scheduling policies:

- **SCHED_OTHER**: the standard round-robin time-sharing policy;
- **SCHED_BATCH**: for "batch" style execution of processes; and
- **SCHED_IDLE**: for running very low priority background jobs.

The following "real-time" policies are also supported, for special
time-critical applications that need precise control over the way in
which runnable processes are selected for execution:

- **SCHED_FIFO**: a first-in, first-out policy; and
- **SCHED_RR**: a round-robin policy.

Scheduling Policies:

- **SCHED_FIFO**: First In-First Out scheduling

  SCHED_FIFO can only be used with static priorities higher
  than 0, which means that when a SCHED_FIFO process
  becomes runnable, it will always immediately preempt
  any currently running SCHED_OTHER, SCHED_BATC
  or SCHED_IDLE process.

- **SCHED_OTHER**: Default Linux time-sharing scheduling

  SCHED_OTHER can only be used at static priority 0.
  SCHED_OTHER is the standard Linux time-sharing
  scheduler that is intended for all processes that do not
  require the special real-time mechanisms.

---

**Binary trees**

- elements have up to 2 child elements
- left child sorts less, right more, than parent
- tree has a depth
- tree has a balance, comparing depths of its left and
  right trees (greater difference, less balance)
Binary tree of months,
for days-per-month determination

Depth: 4
Max comparisons: 6
Average comparisons: 3.5

A skewed tree

Depth: 12
Max comparisons: 12
Average comparisons: 6.5

input sequence: apr, aug, dec, feb, jan, july, june, mar, may, nov, oct, sept (alphabetical)
A balanced tree

input sequence: jul, feb, may, aug, dec, mar, oct, apr, jan, jun, sept, nov

Depth: 4
Max comparisons: 4
Average comparisons: 3.1

search cost O(log N)
2 levels \(\Rightarrow\) 3 elements \(\Rightarrow\) 2 comparisons
3 levels \(\Rightarrow\) 7 elements \(\Rightarrow\) 3 comparisons
4 levels \(\Rightarrow\) 15 elements \(\Rightarrow\) 4 comparisons

Binary tree of last names,
for data record determination

Database

<table>
<thead>
<tr>
<th>Recno</th>
<th>name</th>
<th>rank</th>
<th>serial no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>miller</td>
<td>corporal</td>
<td>4-139</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>major</td>
<td>3-209</td>
</tr>
<tr>
<td>3</td>
<td>baker</td>
<td>private</td>
<td>7-981</td>
</tr>
<tr>
<td>4</td>
<td>smith</td>
<td>lieutenant</td>
<td>3-101</td>
</tr>
<tr>
<td>5</td>
<td>anders</td>
<td>private</td>
<td>8-388</td>
</tr>
<tr>
<td>6</td>
<td>brown</td>
<td>sargeant</td>
<td>8-231</td>
</tr>
<tr>
<td>7</td>
<td>jacob</td>
<td>captain</td>
<td>6-495</td>
</tr>
<tr>
<td>8</td>
<td>johnson</td>
<td>general</td>
<td>4-556</td>
</tr>
</tbody>
</table>
Numbers developed to reflect variance between ideal and actual CPU utilization for each process.

Smallest number ➔ greatest variance (most “underserved”)

Smallest gets CPU. While it runs its metric rises while the others fall till one of them undercut, then it becomes the new running process.

Binary tree of number metrics, for process determination

Tree balance

- depends on insertion sequence
- balance achievable independent of sequence, by performing mid-course re-balancing
  - during insertion, whenever an insertion upsets the balance, re-balance dynamically before inserting next element
  - tree never gets unbalanced, so final result is always balanced
Building tree, no rebalancing

input sequence: 1, 2, 3, 4, 5

Building tree, mid-course re-balancing

input sequence: 1, 2, 3, 4, 5
More information

- scheduler author Ingo Molnar
  https://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

- “Inside the Linux 2.6 Completely Fair Scheduler”

- “Multiprocessing with the Completely Fair Scheduler”