Linux process scheduling

David Morgan

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)

cpu

Task picked to run
(decreasing wait time)

Priority may also be meritorious

task picked to run
(most patient among top priority)

Increasing priority

decreasing wait time
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

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 requirements

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

Former priority scale*

*reflected in literature
"Each process is given a fixed time quantum, after which it is preempted and moved to the expired array. Once all the tasks from the active array have exhausted their time quantum and have been moved to the expired array, while the expired array becomes the active array."

Input to scheduling decisions

NAME
 sched_setscheduler

SYNOPSIS
 int sched_setscheduler( pid_t pid, int policy, const struct sched_param *param );

struct sched_param {
    int sched_priority;
};

DESCRIPTION
Scheduling Policies
The scheduler is the kernel component that decides which runnable process will be executed by the CPU next. Each process has an associated scheduling policy and a static scheduling priority, sched_priority; these are the settings that are modified by sched_setscheduler(). The scheduler makes its decisions based on knowledge of the scheduling policy and static priority of all processes on the system.

(see sched_setscheduler.man.abridged.txt)
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;
- **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 (heavy loops)

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.

**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

input sequence: jan, feb, mar, apr, may, june, july, aug, sept, oct, nov, dec (chronological)

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

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

input sequence: july, feb, may, aug, mar, oct, apr, jan, june, sept, nov

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>jacobs</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 indicates greatest variance (most “underserved”). Smallest gets CPU. While it runs its metric rises while the others’ all fall till one of them undercuts, then it becomes the new running process.

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

insert 1 insert 2 insert 3 insert 4 insert 5

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

Building tree, mid-course re-balancing

insert 1 insert 2 insert 3 insert 4 insert 5

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

final tree unbalanced

final tree balanced
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”