Using SCHED_DEADLINE

Controlling CPU Bandwidth

Steven Rostedt
rostedt@goodmis.org
srostedt@redhat.com
What is SCHED_DEADLINE?

A new scheduling class
others are: SCHED_OTHER, SCHED_FIFO, SCHED_RR
Constant Bandwidth Scheduler
Earliest Deadline First
Other Schedulers

SCHED_OTHER

Completely Fair Scheduler (CFS)
Uses “nice” priority
Each task gets a fair share of the CPU bandwidth

SCHED_FIFO

First in, first out
Each task runs till it gives up the CPU or a higher priority task preempts it

SCHED_RR

Like SCHED_FIFO but same prio tasks get slices of CPU
SCHED_RR (Round Robin)

RR Prio 5
50%

CPU 1

RR Prio 5
50%

RR Prio 5
100%

CPU 2
You have two programs running on the same CPU

One runs a nuclear power plant
   Requires 1/2 second out of every second of the CPU
The other runs a washing machine
   Requires 50 millisecond out of every 200 milliseconds
Which one gets the higher priority?
Priorities
Priorities  Nuke > Washing Machine
Priorities  Nuke < Washing Machine
Rate Monotonic Scheduling (RMS)

Computational time vs Period
Can be implemented by SCHED_FIFO
Smallest period gets highest priority
Compute computation time (C)
Compute period time (T)

\[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \]
Rate Monotonic Scheduling (RMS)

Add a Dishwasher to the mix...

Nuclear Power Plant: $C = 500\text{ms}$ $T = 1000\text{ms}$
Dishwasher: $C = 300\text{ms}$ $T = 900\text{ms}$
Washing Machine: $C = 100\text{ms}$ $T = 800\text{ms}$

\[ U = \frac{500}{1000} + \frac{300}{900} + \frac{100}{800} = .958333 \]
Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling (RMS)

Failed!
Rate Monotonic Scheduling (RMS)

Computational time vs Period
Can be implemented by SCHED_FIFO
Smallest period gets highest priority
Compute computation time (C)
Compute period time (T)

\[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \]
Rate Monotonic Scheduling (RMS)

Computational time vs Period
Can be implemented by SCHED_FIFO
Smallest period gets highest priority
Compute computation time (C)
Compute period time (T)

\[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(\sqrt{2} - 1\right) \]
Rate Monotonic Scheduling (RMS)

Add a Dishwasher to the mix...

Nuclear Power Plant: $C = 500\text{ms}$ $T = 1000\text{ms}$

Dishwasher: $C = 300\text{ms}$ $T = 900\text{ms}$

Washing Machine: $C = 100\text{ms}$ $T = 800\text{ms}$

\[
U = \frac{500}{1000} + \frac{300}{900} + \frac{100}{800} = .958333
\]
Rate Monotonic Scheduling (RMS)

Add a Dishwasher to the mix...

Nuclear Power Plant: $C = 500\text{ms}$ $T=1000\text{ms}$
Dishwasher: $C = 300\text{ms}$ $T = 900\text{ms}$
Washing Machine: $C = 100\text{ms}$ $T = 800\text{ms}$

\[
U = \frac{500}{1000} + \frac{300}{900} + \frac{100}{800} = 0.958333
\]

\[
U \leq n \left( \sqrt[3]{2} - 1 \right) = 3 \left( \frac{3}{\sqrt[3]{2}} - 1 \right) = 0.77976
\]
Rate Monotonic Scheduling (RMS)

\[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left( \frac{n}{\sqrt{2}} - 1 \right) \]

\[ \lim_{n \to \infty} n \left( \frac{n}{\sqrt{2}} - 1 \right) = \ln 2 \approx 0.693147 \]
SCHED_DEADLINE

Utilizes Earliest Deadline First (EDF)
Dynamic priority
The task with next deadline has highest priority

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = 1$$
Earliest Deadline First (EDF)
Earliest Deadline First (EDF)
Earliest Deadline First (EDF)
Earliest Deadline First (EDF)
Earliest Deadline First (EDF)

;) HAPPY :;)

![Diagram of EDF scheduling algorithm with task deadlines and deadlines met]

- Task 1: 4-9
- Task 2: 3-6
- Task 3: 1-2
- Task 4: 0-1
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Earliest Deadline First (EDF)
Earliest Deadline First (EDF)
Implementing SCHED_DEADLINE

Two new syscalls

```c
sched_getattr(pid_t pid, struct sched_attr *attr,
              unsigned int size, unsigned int flags)
```

```c
sched_setattr(pid_t pid, struct sched_attr *attr,
              unsigned int flags)
```
Implementing SCHED_DEADLINE

```
struct sched_attr {
    u32 size;              /* Size of this structure */
    u32 sched_policy;      /* Policy (SCHED_*) */
    u64 sched_flags;       /* Flags */
    s32 sched_nice;        /* Nice value (SCHED_OTHER, SCHED_BATCH) */
    u32 sched_priority;    /* Static priority (SCHED_FIFO, SCHED_RR) */
    u64 sched_runtime;     /* Remaining fields are for SCHED_DEADLINE */
    u64 sched_deadline;
    u64 sched_period;
};
```
Implementing SCHED_DEADLINE

```c
struct sched_attr attr;

ret = sched_getattr(0, &attr, sizeof(attr), 0);
if (ret < 0)
    error();

attr.sched_policy = SCHED_DEADLINE;
attr.sched_runtime = runtime_ns;
attr.sched_deadline = deadline_ns;

ret = sched_setattr(0, &attr, 0);
if (ret < 0)
    error();
```
Most use cases are buggy
Most tasks will not give up the CPU

SCHED_OTHER
Gives up current CPU time slice

SCHED_FIFO / SCHED_RR
Gives up the CPU to a task of the SAME PRIORITY
Voluntary scheduling among same priority tasks
Buggy code!

again:
    pthread_mutex_lock(&mutex_A);
    B = A->B;

    if (pthread_mutex_trylock(&B->mutex_B)) {
        pthread_mutex_unlock(&mutex_A);
        sched_yield();
        goto again;
    }
sched_yield()

What you want for SCHED_DEADLINE!

Tells the kernel the task is done with current period

Used to relinquish the rest of the runtime budget
Donut Hole Puncher!
Deadline vs Period

Can't have offset holes in our donuts
Have a specific deadline to make within a period

runtime $\leq$ deadline $\leq$ period

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} = 1$$
Multi processors!

It's all fun and games until someone throws another processor into your eye
Multi processors! (Dhall's Effect)

M CPUs
M+1 tasks
One task with runtime 999ms out of 1000ms
M tasks of runtime of 10ms out of 999ms
All start at the same time
The M tasks have a shorted deadline
All M tasks run on all CPUs for 10ms
That one task now only has 990 ms left to run 999ms.
Multi processors!

EDF can not give you better than $U = 1$
No matter how many processors you have

Two methods
Partitioning (Bind each task to a CPU)
Global (let all tasks migrate wherever)
Neither give better than $U = 1$ guarantees
Multi processors!

**EDF partitioned**

Can not always be used:

\[
U_{t1} = 0.6 \\
U_{t2} = 0.6 \\
U_{t3} = 0.5
\]

The above would need special scheduling to work anyway

To figure out the best utilization is the bin packing problem

Sorry folks, it's NP complete
Don't even bother trying
Multi processors!

Global Earliest Deadline First (gEDF)

Can not guarantee deadlines of $U > 1$ for all cases

But special cases can be satisfied for $U > 1$

$D_i = P_i$

$U_{\text{max}} = \max\{C_i/P_i\}$

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq M - (M - 1) \times U_{\text{max}}$$
Multi processors!

\[ M = 8 \]
\[ U_{\text{max}} = 0.5 \]

\[
U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq M - (M - 1) \times U_{\text{max}}
\]

\[
U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 8 - (7) \times .5 = 4.5
\]
The limits of SCHED_DEADLINE

Runs on all CPUs (well sorta)
- No limited sched affinity allowed
- Global EDF is the default
- Must account for sched migration overheads

Can not have children (no forking)
- Your SCHED_DEADLINE tasks have been fixed

Calculating Worse Case Execution Time (WCET)
- If you get it wrong, SCHED_DEADLINE may throttle your task before it finishes
Setting task affinity on SCHED_DEADLINE is not allowed

But you can limit them by creating new sched domains

- CPU sets
- Implementing Partitioned EDF
Giving SCHED_DEADLINE Affinity

cd /sys/fs/cgroup/cpuset
mkdir my_set
mkdir other_set
echo 0-2 > other_set/cpuset.cpus
echo 0 > other_set/cpuset.mems
echo 1 > other_set/cpuset.sched_load_balance
echo 1 > other_set/cpuset.cpu_exclusive
echo 3 > my_set/cpuset.cpus
echo 0 > my_set/cpuset.mems
echo 1 > my_set/cpuset.sched_load_balance
echo 1 > my_set/cpuset.cpu_exclusive
echo 0 > cpuset.sched_load_balance
Giving SCHED_DEADLINE Affinity

cat tasks | while read task; do
echo $task > other_set/tasks
done

echo $sched_deadline_task > my_set/tasks
Calculating WCET

Today's hardware is extremely unpredictable
Worse Case Execution Time is impossible to know
Allocate too much bandwidth instead
Need something between RMS and CBS
GRUB (not the boot loader)

Greedy Reclaim of Unused Bandwidth

Allows for SCHED_DEADLINE tasks to use up the unused utilization of the CPU (or part of it)

Allows for tasks to handle WCET of a bit more than calculated.

Not mainline yet, but we are working on that
Documentation/scheduler/sched_deadline.txt
http://disi.unitn.it/~abeni/reclaiming/rtlws14-grub.pdf
http://www.evidence.eu.com/sched_deadline.html
https://cs.unc.edu/~anderson/papers/rtj06a.pdf
Questions?