Implementation of the “Least Slack Time (LST)” scheduling policy in Linux Kernel

Eva Papadogiannaki
papadogian@csd.uoc.gr
Process Scheduling

• Switching from one process to another in a very short time frame

• Scheduler:
  • When to switch processes
  • Which process to choose next

• Major part of the operating system kernel
Scheduler

• Allows the execution of multiple tasks at the “same” time

• Responsible for:
  • Task coordination among processors
  • Avoiding task starvation and preserving fairness
  • Taking into account system-level tasks (e.g., drivers)
Linux Scheduler

• Executes multiple programs at the “same” time, sharing the CPU with users of varying needs
  • Minimizes response time
  • Maximizes overall CPU utilization

• Ideal Scheduling: $n$ tasks share $\frac{100}{n}$ percentage of CPU effort each

• Preemptive: Higher priority processes evict lower-priority running processes

• Quantum Duration
  • Variable
  • Keep it as long as possible, while keeping good response time
History of schedulers in Linux

• Linux v1.2 – Round Robin
• Linux v2.2 – Scheduling Classes & Policies, Categorizing tasks as non/real-time, non-preemptible
• Linux v2.4 – Division in epochs, goodness of function
• Linux v2.6.0 – v2.6.22 – O(1), Runqueues & priority arrays
• Linux v2.6.23 (and after) – Completely Fair Scheduler (CFS)
Completely Fair Scheduler (CFS)

- Virtual time concept
- Running tasks are sorted using “vruntime”
- Time-ordered red-black tree instead of queue
- Maintains balance in providing processor time to tasks
- At each scheduling invocation:
  - The vruntime of the current task is incremented (time spent in processor)
  - The scheduler chooses the leftmost leaf in the tree (lowest vruntime)
- The leftmost node is cached (O(1))
- Reinsertion of a preempted task takes O(logn)
Scheduling Classes and Policies

Modular design to easily support different scheduling policies
- Each task belongs to a scheduling class
- The scheduling class defines the scheduling policy
- Scheduling policy is set by sched_setscheduler()
- Some scheduling policies:
  - SCHED_NORMAL – Default Linux task policy (CFS, fair)
  - SCHED_FIFO – Special time-critical tasks (real-time)
  - SCHED_RR – Round-robin scheduling (real-time)
Linux Kernel source files

• Browse easily through the Linux Kernel source files using this link
  https://elixir.bootlin.com/linux/v2.6.38.1/source

• Actual context switch code, runqueue struct definition, etc.
  • kernel/sched.c
    https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched.c

• Implementation of Completely Fair Scheduling (CFS)
  • kernel/sched_fair.c
    https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched_fair.c

• Implementation of Real-Time Scheduling (RT)
  • kernel/sched_rt.c
    https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched_rt.c

• Tasks are abstracted as struct sched_entity and struct sched_rt_entity (for rt class); Also, check struct sched_class
  • include/linux/sched.h
    https://elixir.bootlin.com/linux/v2.6.38.1/source/include/linux/sched.h
Some code snippets
A task’s scheduling state (defined in include/linux/sched.h)

```c
/* Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 */

#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_BLOCKED 4
#define TASK_STOPPED 8
#define TASK_TRACING 16
#define EXIT_ZOMBIE 32
#define TASK_ZOMBIE 64
#define TASK_WAKEKILL 128
#define TASK_WAKING 256
#define TASK穑 (512)

#define TASK_STATE_TO_CHAR_STR "RDYPIZKMKK"

extern char __assert_task_state[1 + (1 << ((sizeof(TASK_STATE_TO_CHAR_STR) - 1) - floor(log(TASK穑) + 1)));

/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_RT (TASK二线城市 | TASK_STOPPED)
```
Some code snippets

The rq (runqueue) struct (defined in kernel/sched.c)
Some code snippets

The schedule function (in kernel/sched.c)
Some code snippets

The pick_next_task function (in kernel/sched.c)

```c
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* Optimization: we know that if all tasks are in
     * the fair class we can call that function directly: */
    if (likely(rq->nr_running == rq->cfs.nr_running)) {
        p = fair_sched_class.pick_next_task(rq);
        if (likely(p))
            return p;
    }

    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }

    BUG(); /* the idle class will always have a runnable task */
```

Some code snippets

The sched_class struct (defined in include/linux/sched.h)
Handling struct sched_class for fair vs. rt scheduling class

```c
/* All the scheduling class methods:
 */

static const struct sched_class fair_sched_class = {
    .enqueue_task = enqueue_task_fair,
    .dequeue_task = dequeue_task_fair,
    .yield_task = yield_task_fair,
    .check_preempt_curch = check_preempt_wakeup,
    .pick_next_task = pick_next_task_fair,
    .put_prev_task = put_prev_task_fair,
};

#ifdef CONFIG_SMP
    .select_task_rq = select_task_rq_fair,
    .rq_online = rq_online_fair,
    .rq_offline = rq_offline_fair,
    .task_waking = task_waking_fair,
    .set_curr_task = set_curr_task_fair,
    .task_tick = task_tick_fair,
    .task_fork = task_fork_fair,
    .prio_changed = prio_changed_fair,
    .switched_to = switched_to_fair,
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    .task_move_group = task_move_group_fair,
#endif

/ kernel/sched.c

static const struct sched_class rt_sched_class = {
    .enqueue_task = enqueue_task_rt,
    .dequeue_task = dequeue_task_rt,
    .yield_task = yield_task_rt,
    .check_preempt_curch = check_preempt_curch_rt,
    .pick_next_task = pick_next_task_rt,
    .put_prev_task = put_prev_task_rt,
};

#ifdef CONFIG_SMP
    .select_task_rq = select_task_rq_rt,
    .set_cpus_allowed = set_cpus_allowed_rt,
    .rq_online = rq_online_rt,
    .rq_offline = rq_offline_rt,
    .pre_schedule = pre_schedule_rt,
    .post_schedule = post_schedule_rt,
    .task_woken = task_woken_rt,
    .switched_from = switched_from_rt,
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    .task_move_group = task_move_group_fair,
#endif

/ kernel/sched.c
```
Assignment 4 – LST Scheduling Algorithm

• Implement the Least Slack Time (LST) scheduling algorithm, which assigns priority based on the slack time of a process

  \[
  \text{Slack\_time} = \text{deadline} - (\text{computation\_time} - \text{elapsed\_runtime}) - \text{time}
  \]

• Use your code from Assignment 3
  • You will need the set\_deadlines system call you implemented

• Use the guidelines from the previous assignment to compile Linux Kernel and run it
### Assignment 4 – LST Scheduling Algorithm

<table>
<thead>
<tr>
<th></th>
<th>arrival</th>
<th>duration</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>T3</td>
<td>5</td>
<td>10</td>
<td>29</td>
</tr>
</tbody>
</table>

- At time $t=0$: Only task T1, has arrived. T1 is executed till time $t=4$.
- At time $t=4$: T2 has arrived.
  
  Slack time of T1: $33-4-6=23$  
  Slack time of T2: $28-4-3=21$  
  
  Hence T2 starts to execute till time $t=5$ when T3 arrives.
- At time $t=5$:  
  Slack time of T1: $33-5-6=22$  
  Slack Time of T2: $28-5-2=21$  
  Slack Time of T3: $29-5-10=12$  
  
  Hence T3 starts to execute till time $t=13$
- At time $t=13$:  
  Slack Time of T1: $33-13-6=14$  
  Slack Time of T2: $28-13-2=13$  
  Slack Time of T3: $29-13-2=14$  
  
  Hence T2 starts to execute till time $t=15$

- At time $t=15$:  
  Slack Time of T1: $33-15-6=12$  
  Slack Time of T3: $29-15-6=2=12$  
  
  Hence T3 starts to execute till time $t=16$
- At time $t=16$:  
  Slack Time of T1: $33-16-6=11$  
  Slack Time of T3: $29-16-6=12$  
  
  Hence T1 starts to execute till time $t=18$ and so on..

Assignment 4 – Pre-processing and Filtering

- Before `schedule()` selects the next process
- Scan all processes in the runqueue list and find if there is any process that has a deadline (deadline != -1)
  - Calculate its slack time
    - If this process has exceeded the given deadline, remove this process from the runqueue list so it'll never be executed
    - If not, iterate the runqueue list `rq`. For each process `p`, check if `p` has less slack value
      - If so execute process `p` first
Assignment 4 – Demonstrate the new scheduler

Create at least 1 demo.
This demo should do the following:

• Create up to 10 child processes
• For each child process, the parent process will set its remaining computation time and its deadline (set deadline to gettimeofday() + 100 seconds)
• Each child process will spin for the given computation time (e.g., use while, for)
Guidelines for Assignment 4

• Browse kernel code with: https://elixir.bootlin.com/linux/v2.6.38.1/source

• Another way to map source code is by using ctag: http://www.tutorialspoint.com/unix_commands/ctags.htm

• Understand how the scheduler works
  • For example, you can start with printing inside the schedule() function

• Follow the function call path from schedule in order to find out how the next task is picked

• Use the printk() function often, its syntax is close to printf and it’s an easy way to observe the kernel’s behaviour from the user level (with dmesg)

• Reuse existing code snippets within the kernel source code (e.g., to traverse data structures or access members in struct nodes)

• Compile after small changes in the source code (good for easy debugging)

• Submit anything you can that helps you show your effort!