Assignment 4 Tutorial
Linux Scheduler

Michalis Pachilakis
HY345: Operating Systems
Outline

• Linux Scheduler

• Scheduler internals

• History

• Assignment 4
Linux Kernel

• Heart of the Operating System

• Interface between resources and user processes

• What the Kernel does:
  • Memory Management
  • Process Management
  • Device Drivers
  • System calls
Process Management

- Multitasking operating systems
  - Tasks must run in parallel

- Usually, tasks are more than the CPU cores

- Need to make it possible to execute tasks at the same time
Scheduler

- Coordinates how tasks share the available processors
- Prevents task starvation and preserves fairness
- Take into account system tasks
Task Types

- Balance between two types of processes
  - Batch processes
  - I/O Bound tasks

- Preemption: temporarily evict a running task

- Quantum: Variable but keep it as long as possible
Priority

- Linux provides Priority-based scheduling

- A number determines how important a task is
Process Descriptor and scheduler design

- Scheduler needs information for each process
  - `task_struct` holds multiple information about each process
- Scheduler supports a modular design to easily support different scheduling policies
  - Each task belongs to a scheduling class
  - The scheduling class defines the scheduling policy
  - Some scheduling policies:
    - SCHED_NORMAL - Default Linux task policy (CFS, fair)
    - SCHED_FIFO - Special time critical tasks (real-time)
    - SCHED_PR - Round-robin scheduling (real-time)

```c
static const struct sched_class fair_sched_class = {
  .next = idle_sched_class,
  .enqueue_task = enqueue_task_fair,
  .dequeue_task = dequeue_task_fair,
  .yield_task = yield_task_fair,
  .check_preempt_curr = check_preempt_wakeup,
  .pick_next_task = pick_next_task_fair,
  .put_prev_task = put_prev_task_fair,
};

https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched_fair.c
```
The schedule function: schedule(void)

- **Main scheduler function is schedule(void)** - kernel/sched.c
  - It replaces the currently executing process with another
- **Called from different places**
- **Periodic scheduler**
- **Current task enters sleep state**
- **Sleeping task wakes up**
Run queue

- Data structure that manages active processes
- Holds tasks in runnable state
Every task_struct has a sched_entity
- It’s a schedulable object
- Contains timing information used for load balancing and scheduling.
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](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](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](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](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](https://elixir.bootlin.com/linux/v2.6.38.1/source/include/linux/sched.h)
History

- **Genesis**
  - Circular queue
  - Round-robin policy
- **Linux v2.4 - O(n) scheduler**
  - Each task runs a quantum of time in each epoch
  - Epoch advances after all runnable tasks have their quantum
  - At the beginning of each epoch, all tasks get a new quantum
- **Linux v2.6 - O(1) Scheduler**
  - Division between real-time and normal tasks
  - One list per priority
- **Linux v2.6.23 - CFS**
  - Introduced in 2007, Improved in 2016
Completely Fair Scheduling

- Models an “ideal, precise multitasking CPU”

- Ideal scheduling: n tasks share 100/n percentage of CPU effort each

- Fairness:
  - Tasks get their share of the CPU relative to others
  - A task should run for a period proportional to its priority
Completely Fair Scheduling

- Time-ordered red-black tree
  - Runnable tasks are sorted by vruntime

- When a task is executed its vruntime increases
  - Moves to the right of the tree

- Scheduler always selects leftmost leaf
  - Task with the smallest vruntime

- The leftmost node is cached (O(1) access)
- Reinsertion of a preempted task takes O(logn)
Assignment 4 - Group Fairness Scheduling algorithm

- Implement the Group Fairness scheduling algorithm, which assigns equal portion of the CPU to groups and equal equal portion to the processes inside a group

- Process runtime: \( T(process\_params, number\_of\_groups) = \frac{100}{number\_of\_groups/number\_of\_processes\_in\_group(process\_params.group\_name)} \)

- Use your code from Assignment 3

- Use the guidelines from Assignment 3 to compile and run the Linux kernel
Assignment 4 - Group Fairness
Scheduling algorithm

- Each process is assigned to a group during creation

- Each groups get an equal share of the CPU
  - For N groups: $\frac{100}{N}$

- Each process inside a group gets an equal percentage of the group’s share
  - For M processes inside the group: $\frac{100}{N \times M}$

- Process runtime: $T(\text{process_params, number_of_groups}) = \frac{100}{\text{number_of_groups/number_of_processes_in_group(process_params.group_name)}}$
Assignment 4 - Group Fairness
Scheduling algorithm

- Process A1 starts and it’s assigned to group A:
  - Process A1 gets 100% of the CPU since A is the only group and process A1 is the only process in this group (100/1/1)

- Process A2 starts and it’s assigned to group A:
  - Process A2 gets 50% of the CPU since A is the only group and process A2 is the only process in this group (100/1/2). The portion of A1 also need to be recalculated

- Process B1 starts and it’s assigned to group B:
  - Process B1 gets 50% of the CPU since now there are 2 groups (A, B) and on B there is only one process B1 (100/2/1)
  - Process A1 and A2 need to recalculate their CPU portions since now there are 2 groups (100/2/2) and they need to update their portion to 25%

- Process A3 start and it’s assigned to group A:
  - Process A3 gets 16.6% of the CPU since there are 2 groups and 3 processes to group A (100/2/3)
  - Processes A1 and A2 need to recalculate their portions
  - Process B1 portion doesn’t change since it’s in a different group
Assignment 4 - Helpful tips

- When a new process is starting or the scheduler selects the next process
- Scan all the processes in the run queue list
- Count the number of different groups and number of processes in the groups
- Update the portions of CPU time per slice for each process
- Processes can be added or removed, so remember to check
Assignment 4 - Demo

- Create simple demo processes
  - Each process sets its parameters

- Each processes should spin for some time
  - Infinite loop, not sleep

- The scheduler should print:
  - The PID of the task it selected
  - Its parameters
  - Its portion of the CPU
Assignment 4 - More Notes

• 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!
Assignment 4 - Turnin

1. bZImage

2. Modified or created source files

3. Test programs and headers in Guest OS

4. README - Document your effort, and it can go a long way!