People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University M'hamed Bouguerra - Boumerdes
Institute of Electrical and Electronics Engineering
Electronics Department

Option: Computer engineering
Homework Submission
May 01, 2023

Title

EE426: Operating system/Case study: Linux Kernel Scheduling techniques

---

Authored by :
Agli Wafa

Instractor :
Dr.Rachid Naamane

Session : 2022/2023

Contents

1 Introduction

Linux is an open-source operating system used in various devices, such as servers, smartphones, and embedded systems.
The Linux kernel manages tasks and resources of the system, including the scheduling of tasks to be executed by the processor. The Linux kernel scheduler is **responsible for deciding which task to run on the processor at any given time**.
Different scheduling techniques are used by the Linux kernel scheduler to efficiently schedule tasks, including round-robin, priority-based, and real-time scheduling.
Understanding the different Linux kernel scheduling techniques is crucial for developers and system administrators who want to optimize the performance of their Linux-based systems. With this in mind, **what are the different Linux kernel scheduling techniques, and how do they work?**

2 Problem Statement

Discuss, in a very clear manner, what scheduling techniques does Linux kernel use?

3 Solution

Before going through the subject, we will first start by answering some important questions.

3.1 What is a scheduler?

A scheduler is a software component of an operating system that manages the allocation of system resources, such as processor time, memory, and input/output devices, to different tasks or processes that are competing for these resources.
The scheduler decides which task should be executed next based on a set of predefined criteria, such as priority, deadline, and fairness, among others.

3.2 Why does Linux use process scheduling?

Linux uses process scheduling to manage the execution of multiple processes or threads that are competing for system resources, such as the CPU, memory, and input/output devices. Process scheduling ensures that these resources are utilized efficiently and fairly among the different processes, and that each process is executed in a timely manner according to its priority, deadline, or other criteria. By using process scheduling, Linux can support multitasking, multiprocessing, and multithreading, which enable users to run multiple applications, perform concurrent tasks, and achieve higher system performance and responsiveness.

3.3 How does the Linux kernel interpret processes?

In Linux, process scheduling is actually thread scheduling . When a process is scheduled, one of its threads is scheduled to run on a processor. Even if a process has multiple threads, Linux treats it as if it were single-threaded, so scheduling a process requires only one action. However, if a process has N threads, then N scheduling actions are needed to cover all of its threads.
In a multi-threaded process, the threads are related because they share resources such as

the memory address space. Linux threads are sometimes referred to as "lightweight processes" or "Tasks", and they are represented by a data structure in the kernel called task struct . The term "lightweight" emphasizes the fact that the threads within a process share resources, as opposed to separate processes which have their own independent resources.

3.4 What are Linux kernel scheduling techniques?

The Linux operating system differentiates two sorts of processes:

1. Real-Time processes : they are critical and cannot be postponed under any circumstances, regardless of system load.
2. Fair/conventional processes : They do not have particularly severe time limits, therefore they may suffer delays if the system is busy.

The Linux kernel has two scheduling strategies (classes) based on this process distinction: real-time scheduling and conventional or fair scheduling. A scheduling class is a set of function pointers, defined through **struct sched-class**

```
struct sched_class {
    const struct sched_class *next;
    ...
    struct task_struct * (*pick_next_task) (struct rq *rq, struct task_struct *prev);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);
    ...
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    ...
};
```

Each scheduling algorithm gets an instance of **struct sched-class** and connects the function pointers with their corresponding implementations.

1. **Real-Time scheduling:** Real-time scheduling is used for time-critical applications that require guaranteed response times. This technique ensures that the system responds to such applications within a specific timeframe.
   (a) **SCHED-FIFO: First come First served policy (FCFS):** is a non-preemptive First come, First served policy. As a result, the scheduler picks processes depending on their arrival time.
   (b) **SCHED-RR:Round-Robin Scheduling:** This technique assigns equal time slices to each task in a cyclic manner, ensuring fair distribution of the processor time.

2. **Conventional scheduling:**
   (a) **SCHED-NORMAL:Completely Fair Scheduler (CFS):** The CFS became part of the Linux 2.6.23 kernel in 2007, it is a scheduling technique that ensures fairness in distributing processor time to tasks. It uses a mathematical formula to determine which task should be given processor time next, based on the amount of time each task has already consumed.

CFS defines the following time constraints:

1. **Target Latency:** This is the minimum time (20ms, typically) during which every runqueue job is scheduled at least once.

2. **Minimum Granularity:** The smallest time slice that a task can receive when it is scheduled to run on the CPU.

The time required to complete a job on a processor is calculated dynamically using CFS, unlike RR and other traditional scheduling methods. A 1/N slice of the goal latency, where N is the number of tasks, is given to each job in the runqueue (runnable tasks). Because the number of runnable tasks fluctuates as time passes due to task termination, blocking, and the spawning of new ones, this makes the time slice provided to each task dynamic.
Other aspects in CFS also play a role in making the time slice dynamic. But, before we get into those we need to point out an important scenario. When the number of processes increases to a large number, the 1/N timeslice becomes very small, rendering it impractical and may even be less than the time overhead of context switching. For this reason, the Minimum Granularity (MG) will be used instead when 1/N is less than MG. CFS also uses for scheduling two important fields in a **task-struct** called the nice value and virtual runtime.

1. **Priorities in CFS:** The priority managed by the CFS is called the nice value in particular, which ranges between -20 and 19. Lower values mean higher priorities (i.e., -20 for the highest priority and 19 for the lowest priority). The default nice value is 0 unless otherwise inherited from a parent process. Each nice value has a corresponding weight value.There is an inverse relationship between weight values and nice values. The weight value determines how large proportion of CPU time a thread gets compared to other threads.

2. **virtual runtime:** Time accounting in the CFS is done by using the so-called virtual runtime. For a given thread, its virtual runtime is defined as follows:

$$\text{virtual runtime} = \frac{(\text{actual runtime}) \times 1024}{\text{weight}} \tag{1}$$

   Since a weight is proportional to a priority, the virtual runtime of a high priority thread goes slower than that of a low priority thread, when the actual runtime is the same. Note that in the above, 1024 is the weight value for the nice 0 . Thus, the virtual runtime for the thread of nice 0 is equal to its actual runtime.

3.5 How frequently should the CFS scheduler reschedule?

1. If the time slice 1/N is higher than the Minimum Granularity, CFS will reschedule after each target latency period (or somewhat less if good values are used).

2. If, on the other hand, 1/N is less than MG, it will reschedule every N*MG period.

Table 1: Comparison of Round Robin and CFS scheduling algorithms

Round Robin	**CFS**
The time slice is Static and not dependent on the number of processes in the system	The time slice is Dynamic and can be changed according to the number of processes in the system.
when a process finishes its time slice, RR picks the next process from the cyclic list of ready processes	when a process finishes its time slice, CFS picks the next process with the lowst virtual runtime from the red-black tree

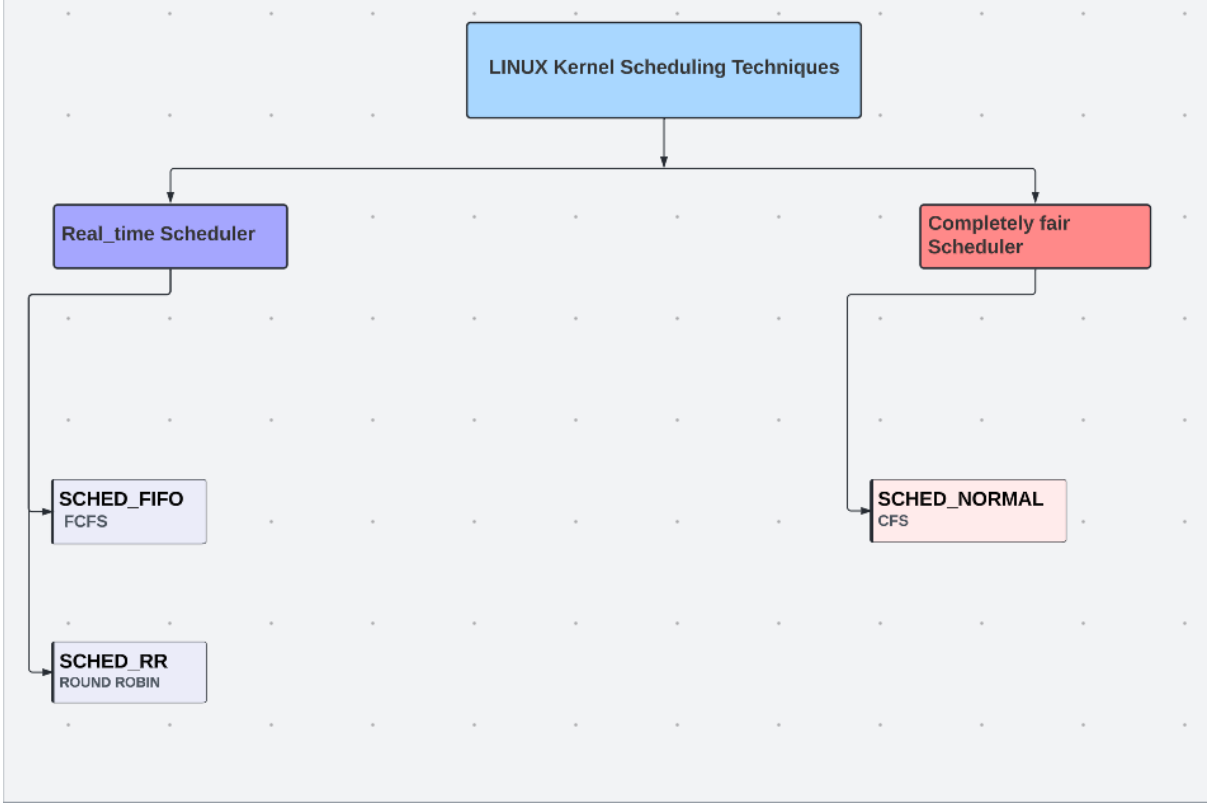

Figure 1: Linux scheduling techniques

4 Outcomes and Results

FCFS is a simple scheduling algorithm that prioritizes processes based on their arrival time. The first process to arrive is the first to be served, and subsequent processes are served in the order in which they arrive. While FCFS is easy to implement and ensures fairness, it may lead to long waiting times for processes that arrive later.

RR, on the other hand, is a preemptive scheduling algorithm that assigns a fixed time quantum to each process. When a process is scheduled, it is allowed to run for its time quantum before being preempted and moved to the back of the queue. This algorithm ensures that no process monopolizes the CPU for an extended period of time and can provide better response times than FCFS.

CFS is a more complex scheduling algorithm that uses a priority-based approach to manage the CPU. It divides the CPU time between all running processes based on their priority level and the amount of time they have already received. This ensures that all processes receive an equal share of the CPU time and prevents any process from starving.

In summary, while FCFS is the simplest scheduling algorithm and RR provides better response times, CFS is the most sophisticated algorithm that provides fair and efficient allocation of CPU resources among all running processes. The choice of algorithm depends on the specific requirements of the system and the types of processes being managed.

5 Conclusion

In conclusion, Linux Kernel Scheduling techniques play a crucial role in determining the performance and responsiveness of Linux-based systems. By using various scheduling algorithms, such as round-robin, completely fair scheduler, and first come first serve, the Linux kernel scheduler can effectively manage and allocate system resources to tasks. Developers and system administrators need to understand the different scheduling techniques and their implications to optimize the performance of Linux-based systems for their intended purposes.

6 Bibliography

References

[1] Naamane Rachid, A. 2023. Lecture notes. Chapter 2,PROCESS MANAGEMENT, 93 pages.

[2] Nikita Ishkov, B. February 2015. A complete guide to Linux process scheduling. University of Tampere.

[3] Xoviabcsr, C. 2017. Operating System 22 Completely Fair Scheduling (CFS). Retrieved from `https://youtu.be/scfDOof9pww`.

[4] JINKYU KOO, D. January 2015 . https://helix979.github.io/jkoo/.