Process Management in Linux

Prof. Dr. Peter Braun

6. Dezember 2024

1 Titlepage

Process Management in Linux

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

2 Learning Goals

Learning Goals

- Understand the concept of processes
- Learn about process states and transitions
- Tell between CPU-bound and IO-bound processes
- Analyze scheduling strategies for different use cases

Welcome to this unit about the theory of process management in operating systems. Today, we begin the last part of this course, where you will learn about some basics of how operating systems work internally. The first topic will be Process Management in a Unix-like operating system. With this unit, you will unriddle the concept of processes and understand their functioning in Linux. Processes are at the heart of any operating system, and we will explore how they are managed. We will identify their varied states from their creation until their termination. As we go on, we will discuss the state transition in detail, which is one of the most important points you must remember. Our discussion will then extend to scheduling strategies — which play a critical role in maintaining system performance. An important aspect of process management is the distinct categories of CPU-bound and IO-bound processes — two contrasting types that affect the performance and efficiency of our systems. Furthermore, in an on-site meeting, we'll discuss different use cases and analyze relevant process scheduling strategies.

3 Process Model

Process Model

- A process is a program currently executed
- The program's source code is in the main memory

- Starting a program twice leads to two processes
- However, the code is not stored a second time

- The process model contains the main memory content, the program counter, and all registers

- **Each process has its virtual CPU**

Let's start by defining what a process is. A process, essentially, is a program that is currently being executed. The source code of the program resides in the main memory. Notably, if you start the same program twice, it creates two processes, yet the code is not stored twice in the main memory. A process model is maintained for every running process. This model contains the main memory content, the program counter, and all other registers of the Central Processing Unit. Each process, in essence, has its virtual CPU.

4 Processes

Executing Processes

- CPU is used only some of the time by one program
- Programs have to wait for I/O devices, e.g. HDD
- Sequential processing leads to a bad workload

- Computers run multiple programs quasi-parallel

- Multi-Programming / Multi-Tasking
- Multi-User

Understanding process management begins with recognizing that a single program does not always use a Central Processing Unit or CPU. Programs often wait for input or output devices like a hard disk drive. The question is whether the processor can do something more meaningful than nothing. Very strict sequential processing, or running tasks one after the other, leads to inefficient workloads because of these waiting times. Hence, it allows for more than one process to work in parallel. An improvement that was already introduced at the beginning of the 1960s in the Fortran Monitor System for the IBM 7090 mainframe computers was an overlapped execution model, where one process uses the CPU while another process waits for Input/Output devices. This very first and simple implementation was named "spooling." Later, this overlapping execution became more sophisticated in the sense that two or more processes share one physical CPU. This is referred to as multi-programming or multi-tasking, providing multi-user functionalities. Both concepts go back to the late 1960s when, for example, the IBM OS/360 operating system was developed.

5 User Space vs. Kernel Space

Kernel Mode vs. User Mode

- **Kernel Mode** = runs in privileged mode on hardware
- Kernel operations: scheduling, memory management

- **User Mode** = all processes outside of the OS kernel
- Processes are strictly separated in user mode

Before we continue and go deeper into this topic of how the operating system realizes this overlapped execution, we need to discuss a concept that modern operating systems have for security purposes. When executing processes, it is distinguished between the kernel mode and the user mode. The core functions of the operating system run in kernel mode, i.e., it has full access to the hardware or resources of the computer. Regarding the CPU, it means that all CPU statements can be executed, including direct access to memory, modifications of the interrupt table, and setting and deleting interrupts. In contrast, all other programs started by normal users are executed in user mode. This is the most important point here; all processes are strictly separated from each other. One process can't access the memory of another process. Processes in user mode cannot modify interrupt handling, and so on.

6 Interleaved Execution of Many Processes

Overlapped Execution of Many Processes

- Process execution is interrupted after a time interval
- The time between interrupts is called *quantum*

Now, let's continue with the question of how an operating system can execute processes in an overlapping way. In the picture beside me, four processes are executed. The x-axis is time. Process 1 starts and runs for some time before it is interrupted by the operating system because then Process 2 has the right to use the CPU for some time. After the last Process 4, the turn returns to Process 1, and the cycle repeats. Why does the operating system switch execution from process 1 to process 2? There are two reasons: the first is that process 1 has nothing to do because it waits for Input/Output. The second reason is that the operating system interrupts a process after a specific time. This time is named a quantum, and the value differs from operating system to operating system. We will go into details now.

7 Small Quantum

Small Quantum

- Processes are switched often
- Shorter response times
- Scheduling is triggered often

What are the consequences of a small quantum? Clearly, a small quantum leads to more frequent switches between processes. On a switch, the operating system must select the next process to execute. This decision is made by the so-called scheduler, and we will talk about the scheduler later today. Another aspect is how fast the process can react to user input. When you press a key on your computer, the hardware informs the operating system about the keystroke. An interrupt on the kernel level is activated, and the operating system kernel stores the input data in a buffer. One process is waiting for user input, which might be selected for execution next. If many processes might be waiting for Input/Output, the small quantum guarantees that all of them have the chance to handle new data soon.

8 Big Quantum

Big Quantum

- Processes are switched rarely
- Processes well-use the CPU
- Longer response times

The opposite case is a big quantum, leading to fewer switches between the processes. Each process can use the CPU for a longer time without interrupts, which will lead to better utilization. One drawback is that it will take longer for a process to handle user input. You can imagine this by a delay between typing in a letter on the keyboard until an effect is shown on the screen.

9 What is the Duration of a Quantum?

What is the Duration of one Quantum?

- 1 second?
- 100 milliseconds?
- 10 milliseconds?
- 1 millisecond?

- Different quantum lengths for different purposes!

You might wonder how long a quantum is in a modern Linux operating system. The answer is difficult because today's operating systems no longer work with fixed length. The size depends on the process's priority and the system's overall load. The quantum can be between, let's say, 1 ms to 5 ms in a desktop system or a laptop. In a real-time operating system, the quantum is longer. The process must complete a computation sooner without many interruptions. A possible value is, for example, 100 ms.

10 Process Management

Process Management

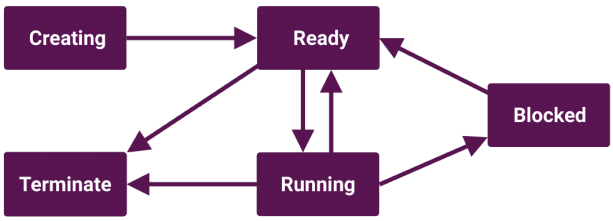
- The OS keeps a table with information on processes
- One table entry: **Process Control Block** (PCB)

- Process identifier
- Process state
- Programm counter (PC)
- Content of registers
- Stack pointer (SP)
- Segment pointer
- Priority
- User identifier
- Group identifier
- Working directory
- Root directory
- File handler

Next, we discuss how the operating system manages processes internally. Let's start with the data structures before continuing with the functions. The operating system keeps track of all processes in a table, each entry of which is termed a Process Control Block, or PCB. A PCB contains information about the process, such as process identifier, process state, program counter, content of registers, and so on. You can see the list beside me. The Linux kernel keeps one PCB for each process and stores this information in the memory area that belongs to the kernel; we call this the kernel space. The kernel space is strictly separated from the user processes. So, we can make this clear again: all data /of/ the process, variables, stack values, and so on are located in the user space. The metadata/about/ processes is stored in the kernel space.

11 Process States

Process States



Let's focus on the process states in Linux, which are fundamental to understanding process management. Every process in a system exists in one of these states at any given time. Let's examine what each state represents. First, we have the Creating state. This is where a process begins its lifecycle. A program is launched, and the operating system allocates the necessary resources, such as memory and process control structures, to initialize the process. At this stage, the process is not yet ready to execute. Next is the Ready state. A process enters this state once it is fully initialized and ready to run, but it is waiting for CPU time. Many processes can be in the Ready state simultaneously, and the scheduler decides which one gets to execute next. Once a process is scheduled, it moves to the Running state. This is where the process actively executes instructions on the CPU. However, a process cannot remain in the Running state indefinitely; it will eventually pause because it has completed its execution, requires I/O, or another higher-priority process needs the CPU. Processes performing I/O or waiting for external events enter the Blocked state. In this state, the process can only proceed once the requested operation is complete or an external event occurs. Once it's ready again, it transitions back to Ready. Finally, the Terminated state marks the end of a process's lifecycle. The process has either been completed successfully or has been explicitly stopped. All allocated resources are reclaimed, and the process ceases to exist. Understanding these states is crucial, as they form the basis for the scheduling and resource management decisions we'll discuss next.

12 Process Creation

Process Creation

- Create a process copy by system call `fork`
- `fork` creates a parent-child relation
- Child process can start a program by *execve*
- These two steps are required to pass through file handlers, default I/O, and errors

We now discuss some more details about process states. Let's consider an example to understand the starting point of a process state in Linux. A system call called 'fork' creates a process copy. The 'fork' command also creates a parent-child relationship between the original process and the copy. Why is this parental bond important? It allows the child process to inherit certain crucial attributes from the parent, such as user and group IDs, environment variables, and open file descriptors. Now that we have a child process, what happens next? Here, another command, 'execve,' is used to start an entirely new program. Why are these two steps - 'fork' and 'execve'- required together? For several reasons. First, they allow us to preserve the context of the current process, making software development less complex. Then, they ensure the child process can pass through file handlers, manage default input/output settings, and handle errors effectively.

13 Transition: From Running to ...

Transition: From Running to ...

- Timer interrupt, next state: *Ready*
- Invoke I/O system call, next state: *Blocked*

- Scheduler picks process in state *Ready* for execution

This slide focuses on the transition of a running Linux process currently in the state of Running. What are the possible next states? We'll cover two key scenarios here. First, we have a Timer interrupt. Timer interrupts serve as a crucial element to manage time-sharing among processes in Linux. When a running process in Linux gets interrupted by a Timer, it moves from the 'Running' state to a 'Ready' state. The process is no longer being executed but remains prepared to continue running when CPU time is granted. Next, let's consider an I/O system call scenario. When a running process initiates a required Input/Output operation, the system call functions as a gateway into the privileged mode for executing special I/O instructions. This results in the process shifting to being 'Blocked.' Being in this 'Blocked' state means that the process cannot continue its execution until the I/O call is fully resolved. Lastly, we take a look at the role of the scheduler. Scheduling is an integral part of the process management in Linux. It chooses among the 'Ready' processes and picks one for execution. Understand that scheduling is dictated by algorithms and policies aimed at efficient and fair CPU time division. We will come back to this later.

14 Transistion: From Blocked to Ready

Transistion: From Blocked to Ready

- A blocked process is waiting for data (disk, network)
- Incoming data is indicated via interrupts

- When all data is available, the process switches to *Ready*

Transitioning from a blocked state to a ready state is a foundational aspect of process management in Linux. Imagine a process that is blocked. It waits for data from the hard disk or the network. It waits because it relies on this data to proceed. The arrival of incoming data is signaled using interrupts. An interrupt is a notification that prevents the processor from following the default sequence of instructions, allowing it to handle this incoming data. Note that the interrupt is managed by the operating system, not the process itself. The operating system then checks if all the necessary data has arrived. Then, the process is no longer in the waiting state. With all its data dependencies fulfilled, it's ready to run. This is when it transitions to the 'Ready' state.

15 Process Termination

Process Termination

- Regular termination because the program exits
- Termination due to errors (e.g., segmentation fault)
- Operating system terminates the process

A process can further change into the final state named “terminate,” and we distinguish three reasons for this. The regular termination is a planned exit, where the program concludes its tasks and terminates by itself. Alternatively, termination can also be caused by errors. For instance, a segmentation fault is a specific error caused by accessing memory that does not belong to it. This error forces the process to terminate immediately. Moreover, the operating system has the authority to terminate a process. It might do this to manage resources, respond to a user request, or handle an error. Remember, these transitions are integral to the smooth functioning of any operating system.

16 Scheduler and Dispatcher

Scheduler and Dispatcher

- **Scheduler** selects the next process for execution
- **Dispatcher** gives control to the next process
- The dispatcher saves the state of the current process and loads the last state of the next process

The responsibility of deciding which next process gets to run lies with the Scheduler. The Scheduler is an intrinsic part of the Operating System that employs certain algorithms to make this selection, maximizing efficiency and fairness. The Dispatcher then steps in once the Scheduler has selected the succeeding process. Its task is to switch the CPU's control from the current process to the next. For this clean handover, the Dispatcher saves the presently running process's state and fetches the next process's last state.

17 Process Scheduling is Expensive

Process Scheduling is Expensive

- Halting the current process and storing the state
- Selecting the next process using some strategy
- Loading the state of the next process
- Executing the next process

- Current CPU cache gets invalid
- Trade-off: **high responsiveness** vs. **low waste** of time

Diving deeper into process management, we now focus on an essential aspect known as Process Scheduling. One particular trait of this is that it can be quite costly regarding system resources. This is maintained because of a chain of activities triggered each time a process switch occurs. Firstly, the current active process is halted. The system must save its status - all the information, such as its CPU registers, pending inputs and outputs, and its progress so far. All this data is stored away for the next time the process is called upon. Next, the choice must be made regarding which method is to run next. This is done based on a specific strategy - this could be round-robin scheduling, for instance, or it may employ a more complex approach, such as priority scheduling or shortest job next. Once the next process is selected, its state needs to be loaded. This means bringing its previously stored state back into active memory, ready for the CPU to start work on it again. This involves moving lots of data around, which utilizes both time and computing resources. Following the state loading, the next process is then executed. However, there's a catch. The current CPU cache will logically become invalid, corresponding to the halting process and not the new one. Clearing and reloading cache data is another overhead, making process scheduling more expensive. This is a balancing act, a trade-off, if you will, between high responsiveness and low waste of time. High responsiveness brings user satisfaction and improved system usability but significantly reduces system overload. In contrast, reducing system overload leads to low waste of time but may reduce system responsiveness.

18 CPU-bound vs. IO-bound Processes

CPU-bound vs. IO-bound Processes

- CPU-bound: needs CPU much and needs less IO
- IO-bound: do a lot of IO and don't need the CPU much

- **Scheduling IO-bound process is more important**

We will explore two key process types in Linux: CPU-bound and IO-bound. A process can be categorized as CPU-bound or IO-bound, depending on its primary resource requirement. A CPU-bound process requires the CPU's processing power. This means it requires a lot of CPU time as it primarily involves calculations and does not rely heavily on input-output operations. Mathematical computations, for instance, can make a process CPU-bound. In contrast, an IO-bound process involves more input-output operations, such as reading from or writing to a disk. Such a process does not necessarily need much CPU time, as it involves waiting for data to be fetched or written. It is critical to note that scheduling IO-bound processes tends to be a higher priority. The rationale is that these processes, once given the necessary resources, are completed quickly, allowing for optimal utilization of system resources.

19 Scheduling: Non-preemptive Strategy

Scheduling: Non-preemptive Strategy

- **The process releases the CPU voluntarily**
- The process changes into the state *Blocked*
- The system freezes when a process keeps the CPU
- Examples: Windows before 95 or MacOS before 2001

We come to the last topic of today's unit. We want to discuss scheduling strategies. Let's start with two major categories. The first one is called non-preemptive, and this class of scheduling strategies can be considered outdated. The key idea to remember is that should a single process use the CPU without releasing it, the entire system grinds to a halt. This might seem unusual now, but it was commonplace in earlier operating systems, such as Windows versions before 95 or MacOS before 2001. Such situations underline the importance of process management and scheduling, ensuring smooth and optimized system performance.

20 Scheduling: Preemptive Strategy

Scheduling: Preemptive Strategy

- **A process is interrupted by the operating system**
- A timer interrupt starts the scheduler
- The scheduler checks if processes can be switched

- Examples: Windows since 95 or MacOS X or Linux

In a preemptive scheduling strategy, the operating system can interrupt a currently running process. This robust strategy allows a more flexible allocation of system resources, but why does this interruption occur? A timer interrupt initiates this mechanism. The timer could be likened to a clock, which regularly signals the operating system to check the status of running processes. Having received this timer interrupt signal, the scheduler comes into action. It performs the critical task of deciding which process should be given execution priority at any given moment. We have already discussed that before. This preemptive strategy forms the backbone of many modern operating systems. For instance, Windows has used it since the release of Windows 95, Linux uses it, and Mac OS X manages its processes preemptively.

21 Goals of Scheduling Strategies

Goals of Scheduling Strategies

- **Different uses-cases require different strategies**

- Batch Processing: patient users, long quantum
- Desktop: high responsiveness, short quantum

As we delve further into Process Management in Linux, let's focus on the goals of scheduling strategies. Firstly, we have Batch Processing. This strategy suits scenarios where users can afford to be patient. Processes can work with longer quantum periods. Here, the emphasis is maximizing throughput and minimizing the need for interaction during processing. Secondly, there's the Desktop strategy. This swings to the opposite end of the spectrum, with high responsiveness as the primary goal. For a rich interactive user experience, this strategy employs short scheduling quantum values.

22 Throughput and Turnaround-time

Throughput and Turnaround-time

Throughput

- Number of jobs done per time interval
- More is better

Turnaround-time

- Difference between end and start time
- Less is better

One crucial factor across all strategies is the number of jobs completed within a given time frame. The general rule of thumb, all else being equal, is that more is better. However, while we seek to maximize the job completion rate, we must also pay attention to turnaround times - the difference between the end and start times of processes. In this case, less is preferable, which means processes are being completed more rapidly. These different strategies inform the scheduler's decisions, guiding it as it manages a system's resources. By understanding them, we better understand how Linux optimizes different environments for peak performance.

23 Overview of Scheduling Strategies

Overview of Scheduling Strategies

Batch Processing

- First-Come-First-Served
- Shortest-Job-First
- Shortest-Remaining-Time

Desktop

- Round-Robin
- Priorities
- Shortest Process
- Lottery
- Fair-Share

On this slide, we'll look at an overview of scheduling strategies in Linux. You can see the two classes of strategies we mentioned earlier: For batch processing vs. desktop systems. I will go into details here, but I want to give you an outlook on the next on-site meeting. All the strategies mentioned here are described in detail in the given literature. In the meeting, we will do a group exercise where you prepare a poster about one of these strategies and teach the other students about your strategy.

24 Summary

Summary

- The operating system manages processes
- Scheduling strategies ensure efficiency and fairness

- Preemptive scheduling allows the OS to interrupt
- Non-preemptive relies on the voluntary releases

- Categories of scheduling adapt to different needs
- Throughput and turnaround measure effectiveness

That's it for today. You learned that managing processes is one of the main tasks of an operating system. Our primary interest has been in understanding how the operating system schedules these processes. Different scheduling strategies are employed to maintain efficiency and fairness, allowing all processes to use system resources. We're looking to balance maximum system use with equitable resource allocation. As we've seen, different circumstances warrant different categories of scheduling. Priorities can be adjusted based on requirements such as real-time processing, process dependency, or user preference. In conclusion, process management in Linux is a complex and multifaceted topic.