Scheduling

OPERATING SYSTEMS

Sercan Külcü | Operating Systems | 16.04.2023

Contents

Chapter 5: Scheduling

1 Introduction

Welcome to the chapter on CPU scheduling in operating systems! In this chapter, we will discuss the fundamental concepts and goals of CPU scheduling.

Firstly, we will provide a definition of CPU scheduling and highlight its significance in modern operating systems. CPU scheduling is the process by which an operating system selects a process from a pool of processes to allocate the CPU to. This process is crucial as the CPU is the most valuable resource in a system, and efficient utilization of the CPU is essential for optimal system performance.

Next, we will explore the goals of CPU scheduling. These goals include maximizing CPU utilization, ensuring fair allocation of CPU time among processes, minimizing response time, and ensuring that processes are executed in a predictable manner. We will discuss how these goals are achieved and the various techniques used to accomplish them.

So, let's dive into the world of CPU scheduling and learn how it impacts the performance and efficiency of modern operating systems.

1.1 Definition of CPU scheduling

In a modern operating system, multiple processes compete for CPU time, which is a scarce and valuable resource. CPU scheduling is the process of determining which process should be allocated CPU time and for how

long. It is a crucial aspect of operating system design, as efficient scheduling algorithms can significantly improve system performance and user experience.

CPU scheduling involves maintaining a queue of ready processes and selecting which process to run next. The scheduler must make decisions quickly and efficiently, taking into account factors such as process priority, CPU utilization, and fairness.

The goal of CPU scheduling is to maximize CPU utilization while providing fair access to CPU resources for all processes. A good scheduling algorithm should balance the needs of different processes, preventing any single process from monopolizing the CPU and causing other processes to wait too long.

CPU scheduling algorithms can be preemptive or non-preemptive. In a preemptive algorithm, the scheduler can interrupt a running process to allocate CPU time to another process. In a non-preemptive algorithm, the running process must voluntarily give up the CPU before another process can run.

Overall, CPU scheduling is a critical component of operating system design. An effective scheduling algorithm can improve system performance, responsiveness, and fairness, while a poorly designed algorithm can result in slow and unresponsive systems.

1.2   Importance of CPU scheduling in operating systems

CPU scheduling is a crucial component of any operating system, responsible for managing the allocation of the CPU's resources to various processes and threads. It is a fundamental task that directly affects the performance, responsiveness, and fairness of an operating system.

The primary goal of CPU scheduling is to maximize CPU utilization while ensuring that processes and threads are executed in a fair and efficient manner. In a multi-tasking environment, where multiple processes and threads compete for the CPU's resources, effective CPU scheduling can significantly improve the overall system's performance.

By using CPU scheduling, an operating system can provide a fast and responsive user experience, allowing users to interact with the system while simultaneously running multiple applications in the background. Effective CPU scheduling can also improve the overall throughput of the system, enabling more work to be accomplished in less time.

Moreover, effective CPU scheduling can also ensure that high-priority processes and threads are executed first, ensuring that critical tasks are completed promptly. It can also help prevent processes and threads from monopolizing the CPU, allowing other processes and threads to run and use resources, which improves the overall system's fairness.

Overall, CPU scheduling is a vital component of an operating system, playing a critical role in ensuring that the system performs optimally, is responsive, efficient, and fair. In the following sections, we will discuss the various goals and methods of CPU scheduling, which will provide a deeper understanding of how operating systems allocate resources to processes and threads.

1.3 Overview of the goals of CPU scheduling

CPU scheduling is an essential component of an operating system that helps manage the allocation of CPU time among competing processes. The primary objective of CPU scheduling is to increase the system's overall efficiency by minimizing the CPU idle time and maximizing the CPU utilization while ensuring that the system remains responsive to user requests.

In addition to improving the system's performance, CPU scheduling has several other goals, including:

- Fairness: CPU scheduling should ensure that all processes receive a fair share of the CPU time and that no process is given an unfair advantage over others.
- Priority: Some processes may have higher priority than others, such as real-time processes, which require immediate attention. The scheduling algorithm must ensure that higher priority processes receive the necessary CPU time while not completely starving lower priority processes.
- Response time: The time between a user request and the system's response should be as short as possible. The scheduling algorithm should prioritize processes that are interactive or waiting for user input, ensuring that the system remains responsive.
- Throughput: The number of processes completed per unit time should be maximized. The scheduling algorithm should aim to complete as many processes as possible in a given time period.
- Predictability: The behavior of the scheduling algorithm should be predictable, and the scheduling decisions should be transparent to the user and the system.

Achieving these goals is not always easy and often requires a trade-off between them. For example, a scheduling algorithm that maximizes throughput may not provide the best response time or fairness. Thus, the selection of a scheduling algorithm depends on the system's characteristics, workload, and the specific goals to be achieved.

In the next chapters, we will discuss different scheduling algorithms and techniques used by modern operating systems to achieve these goals.

2 Scheduling Algorithms

In this chapter, we will explore one of the most important functions of operating systems - CPU scheduling. We'll begin with a definition of CPU scheduling and discuss its importance in modern operating systems. Then, we'll dive into the various scheduling algorithms used by operating systems, including First-Come-First-Serve (FCFS), Shortest-Job-First (SJF), Priority Scheduling, Round-Robin (RR), and Multilevel Feedback Queue (MLFQ). We'll explore the strengths and weaknesses of each algorithm, and discuss how they are implemented in practice. Finally, we'll conclude with a comparison of the different scheduling algorithms and provide recommendations on when to use each one. So, let's get started!

2.1 First-Come-First-Serve (FCFS)

In the world of operating systems, scheduling algorithms play a critical role in managing resources efficiently. The First-Come-First-Serve (FCFS) algorithm is the simplest scheduling algorithm, which is commonly used in operating systems. In this chapter, we will discuss FCFS scheduling in detail, including its definition, advantages, disadvantages, and how it works.

The FCFS scheduling algorithm is the simplest scheduling algorithm that works on a non-preemptive basis. In this algorithm, the process that arrives first is executed first. The FCFS algorithm is implemented using a queue data structure, where the arriving processes are added to the tail of the queue, and the processor executes the process that is at the front of the queue.

The main advantage of the FCFS scheduling algorithm is that it is simple to implement and understand. It is also a fair scheduling algorithm because it follows the principle of first-come-first-serve, which means

that the process that arrives first will get executed first. Additionally, it is suitable for batch processing systems where there is no need for interactivity between the user and the system.

The FCFS scheduling algorithm has several disadvantages. One of the significant drawbacks is that it does not take into account the CPU burst time of a process. If a long process arrives first, it will hold the CPU for an extended period, causing other processes to wait, which may lead to poor performance. This problem is known as the convoy effect. Additionally, the FCFS algorithm is not suitable for interactive systems because it does not provide good response times.

The FCFS scheduling algorithm works by implementing a queue data structure. When a process arrives, it is added to the tail of the queue. The processor executes the process that is at the front of the queue. The CPU remains busy until the process completes its execution, and the next process is dequeued from the queue.

If a new process arrives while the processor is busy, it is added to the tail of the queue. The FCFS algorithm does not interrupt the currently executing process, even if a higher priority process arrives.

The FCFS scheduling algorithm is a simple and fair scheduling algorithm that is widely used in operating systems. However, it has several disadvantages, such as poor performance due to the convoy effect and lack of responsiveness in interactive systems. Therefore, it is not suitable for real-time and interactive systems.

Example: Here's a pseudocode for the First-Come-First-Serve (FCFS) CPU scheduling algorithm:

```
// Initialize the ready queue with processes

ready_queue = [P1, P2, P3, ..., PN]


// Set the current process to the first one in the queue

current_process = ready_queue[0]
```

```
// Execute each process in order of arrival

for process in ready_queue:

    // Switch to the next process

    current_process = process


    // Execute the process

    execute(current_process)
```

In this pseudocode, we start by initializing the ready queue with all the processes that are ready to be executed. We set the current process to the first process in the queue.

Then, we loop through each process in the ready queue, and for each process, we switch to it as the current process and execute it. Since FCFS executes processes in the order of their arrival, this pseudocode ensures that each process is executed in the same order it arrived in the ready queue.

2.2 Shortest-Job-First (SJF)

In operating systems, scheduling algorithms are used to determine which process should be given the CPU time and for how long. One of the most commonly used scheduling algorithms is Shortest-Job-First (SJF) scheduling. The basic idea behind SJF scheduling is to prioritize the process with the shortest burst time to run first, allowing for quicker turnaround times and improved performance. In this chapter, we will take a detailed look at SJF scheduling, its advantages and disadvantages, and its implementation.

The SJF scheduling algorithm is based on the assumption that the process with the shortest burst time should be scheduled first. In other

words, the process that will take the least amount of time to execute should be given priority. When a process enters the ready queue, its burst time is calculated, and the process with the shortest burst time is selected for execution.

There are two variations of the SJF scheduling algorithm: non-preemptive SJF and preemptive SJF.

2.2.1 Non-Preemptive SJF Scheduling:

In non-preemptive SJF scheduling, once a process has been assigned the CPU, it will continue to run until its completion. This means that a process cannot be interrupted by another process with a shorter burst time. Non-preemptive SJF scheduling is also known as Shortest-Job-Next (SJN) or Non-Preemptive Priority Scheduling.

Example: The following pseudocode illustrates the implementation of non-preemptive SJF scheduling:

1. Sort the processes in the ready queue by their burst times (shortest to longest).

2. While the ready queue is not empty:

    a. Dequeue the first process in the queue.

    b. Assign the CPU to this process.

    c. Wait for the process to complete.

2.2.2 Preemptive SJF Scheduling:

In preemptive SJF scheduling, a running process can be preempted by a newly arrived process with a shorter burst time. This means that the process with the shortest remaining burst time will be given priority to execute, regardless of whether it is currently running or not. Preemptive SJF scheduling is also known as Shortest-Remaining-Time-First (SRTF).

Example: The following pseudocode illustrates the implementation of preemptive SJF scheduling:

```
1. Initialize the currently running process to null.

2. While the ready queue is not empty:

    a. Sort the processes in the ready queue by their remaining
burst times (shortest to longest).

    b. If the currently running process has a longer remaining
burst time than the first process in the queue:

        i. Preempt the currently running process.

        ii. Enqueue the preempted process back into the ready queue.

        iii. Dequeue the first process in the queue.

        iv. Assign the CPU to this process.

    c. Wait for the process to complete.
```

The SJF scheduling algorithm has several advantages, including:

- It reduces average waiting time, as processes with shorter burst times are executed first.
- It minimizes average turnaround time, as processes are executed in the order of their burst times.
- It improves system efficiency, as CPU time is allocated to the process that requires it the most.

Despite its advantages, the SJF scheduling algorithm also has some disadvantages, including:

- It is difficult to predict burst times accurately, which can lead to poor scheduling decisions.
- It can cause long waiting times for processes with long burst times, as they will be scheduled last.
- It can result in starvation of processes with longer burst times, as they may never get the opportunity to execute.

To implement SJF scheduling, the operating system must know the length of the next CPU burst for each process. One way to estimate this is to use the length of the previous CPU burst, although this method may not always be accurate. Another way is to use an exponential average of the previous burst lengths, which gives more weight to recent bursts.

Once the estimated burst lengths are known, the processes can be scheduled based on the shortest estimated burst length. If a new process arrives with a shorter estimated burst length than the currently running process, the currently running process is preempted and the new process is scheduled to run.

SJF scheduling can either be non-preemptive or preemptive. Non-preemptive SJF scheduling means that once a process starts running, it will continue to run until it completes its CPU burst. Preemptive SJF scheduling means that if a new process arrives with a shorter estimated burst length, the currently running process is preempted and the new process is scheduled to run.

Preemptive SJF scheduling can lead to starvation if a long process keeps being preempted by shorter processes, and therefore never completes. One way to mitigate this is to use a priority queue, where processes with shorter estimated burst lengths have higher priorities.

Overall, SJF scheduling is a good choice for systems where the length of CPU bursts is known or can be estimated accurately. It can lead to shorter average waiting times and turnaround times compared to FCFS scheduling, and is fairer in terms of allocating CPU time to processes with shorter burst lengths.

2.3 Priority Scheduling

Priority scheduling is a non-preemptive CPU scheduling algorithm in which each process is assigned a priority, and the process with the

highest priority is executed first. In priority scheduling, each process is assigned a priority based on its characteristics, such as the amount of CPU time it needs, its importance to the system, and the amount of I/O it requires. A process with a higher priority value will be executed before a process with a lower priority value.

Priority scheduling can be implemented in different ways. One common approach is to use static priorities, where the priority of a process is set at the time of its creation and remains constant throughout its execution. Another approach is to use dynamic priorities, where the priority of a process changes during its execution based on certain criteria.

There are various factors that can be used to assign priorities to processes. Some of the commonly used factors are:

- CPU Burst Time: The time that a process requires to complete its execution is an important factor in determining its priority. A process that requires a shorter CPU burst time is given a higher priority than a process that requires a longer CPU burst time.
- Deadline: If a process has a strict deadline by which it must complete its execution, it is given a higher priority than other processes.
- I/O Requirement: Processes that require more I/O operations are given a lower priority than processes that require less I/O operations.
- Memory Requirement: Processes that require more memory resources are given a lower priority than processes that require less memory resources.

In priority scheduling, the scheduler selects the process with the highest priority from the ready queue and assigns the CPU to it. If two processes have the same priority, they are executed in a First-Come-First-Serve (FCFS) manner.

One of the advantages of priority scheduling is that it allows the system to be more responsive to high-priority processes. For example, if a critical system process requires immediate attention, it can be assigned a higher priority, and the scheduler will ensure that it is executed before other processes. Another advantage is that it allows the system to be more efficient by maximizing the use of available resources. By executing high-priority processes first, priority scheduling can ensure that the system makes the most efficient use of CPU time.

However, priority scheduling also has some disadvantages. One potential problem is that lower-priority processes may suffer from starvation, which means that they may never get a chance to execute if there are always high-priority processes waiting in the ready queue. Another problem is that priority inversion may occur, where a low-priority process holds a resource that a high-priority process needs, causing the high-priority process to be blocked.

Overall, priority scheduling is a useful CPU scheduling algorithm that can be used to ensure that the system is responsive to high-priority processes and efficient in its use of resources. However, it is important to carefully assign priorities to processes and to take steps to avoid problems such as starvation and priority inversion.

Example: Here's a pseudocode for Priority Scheduling:

```
1. Initialize an empty ready queue for each priority level

2. for each process do the following:

3.    set priority of the process

4.    enqueue the process in the corresponding ready queue

5. while there are processes in the ready queues do the following:

6.    select the highest priority process from the non-empty ready
queue

7.    execute the selected process for a time slice
```

```
8.      if the process is still runnable, re-enqueue it in the
corresponding ready queue
```

In this pseudocode, we first initialize a separate ready queue for each priority level. Each process is then assigned a priority and enqueued in the corresponding ready queue. The scheduling algorithm then selects the highest priority process from the non-empty ready queues and executes it for a time slice. If the process is still runnable after the time slice, it is re-enqueued in the corresponding ready queue. The process repeats until there are no more processes in the ready queues.

2.4 Round-Robin (RR)

Round-Robin (RR) is a CPU scheduling algorithm in which each process is assigned a fixed time slice or time quantum. When a process arrives in the ready queue, it is assigned the CPU for a fixed time quantum, usually in the range of 10 to 100 milliseconds. If the process completes its execution before the time quantum expires, it voluntarily relinquishes the CPU. However, if the time quantum expires before the process completes its execution, the process is preempted, and the CPU is assigned to the next process in the ready queue. The preempted process is then placed at the end of the ready queue, where it waits for its turn to come again.

The RR algorithm is widely used in real-time systems, where it is essential to ensure that all processes get a fair share of CPU time, regardless of their priorities. It is also used in interactive systems, where it is important to provide a responsive user interface.

One of the advantages of the RR algorithm is that it provides fairness in the sense that all processes get an equal share of CPU time. This is achieved by giving each process a fixed time quantum, after which it is preempted and replaced by the next process in the ready queue. Another advantage of the RR algorithm is that it provides good response

time, as processes are executed in a round-robin fashion, with each process getting a chance to run for a fixed time quantum.

However, one of the disadvantages of the RR algorithm is that it may result in unnecessary context switches, as processes are preempted even if they do not require the entire time quantum to complete their execution. This can lead to a decrease in the overall system performance. To mitigate this issue, the time quantum must be chosen carefully, to balance between fairness and responsiveness.

Example: Pseudocode for Round-Robin (RR) Scheduling Algorithm:

```
1. Initialize the ready queue and set the time quantum (q).

2. While the ready queue is not empty:

    a. Dequeue the first process from the ready queue.

    b. If the process can complete its execution within the time
quantum (q):

        i. Execute the process for the required CPU time.

        ii. Update the process's state to completed.

    c. Else:

        i. Execute the process for the time quantum (q).

        ii. Update the process's state to ready.

        iii. Enqueue the process at the end of the ready queue.
```

In summary, the Round-Robin (RR) CPU scheduling algorithm provides fairness and good response time by giving each process a fixed time quantum to execute, after which it is preempted and replaced by the next process in the ready queue. It is widely used in real-time and interactive systems, but it may result in unnecessary context switches if the time quantum is not chosen carefully.

2.5 Multilevel Feedback Queue (MLFQ)

Multilevel Feedback Queue (MLFQ) scheduling is a dynamic scheduling algorithm that employs multiple priority queues to schedule processes. It is an extension of the priority scheduling algorithm, but with the added advantage of dynamically adjusting priorities based on process behavior.

The MLFQ scheduling algorithm works by maintaining a set of queues, each with a different priority level. Each queue has a different time quantum assigned to it, with higher priority queues having smaller time quanta. When a process enters the system, it is assigned to the highest priority queue. The process runs until its quantum expires, or it blocks for I/O. If the process uses up its entire quantum, it is demoted to the next lower priority queue. If a process blocks before its quantum expires, it is placed at the back of the same priority queue. This allows I/O-bound processes to move up in priority faster than CPU-bound processes.

The MLFQ scheduling algorithm attempts to provide the benefits of both short-term and long-term scheduling. Short-term scheduling is achieved by using smaller time quanta for higher priority processes, while long-term scheduling is achieved by periodically demoting processes to lower priority queues. This allows CPU-bound processes to complete without starving I/O-bound processes.

MLFQ scheduling also incorporates a feature known as aging, which increases the priority of a process that has been waiting in a lower priority queue for a long time. This ensures that processes that have been waiting for a long time are eventually given a chance to execute, preventing indefinite starvation.

MLFQ scheduling has been shown to perform well in most scenarios, but there are some cases where it can perform poorly. For example, if a process has a burst of CPU activity that is longer than the time quantum

of the highest priority queue, it will be demoted to a lower priority queue before it completes its burst. This can result in unnecessary context switching and decreased performance.

Example: Here is an example pseudocode implementation of the MLFQ scheduling algorithm:

```
initialize all queues

set time quantum for each queue

set priority of initial queue


while (true) {

  if (any queue is not empty) {

    select the highest priority non-empty queue

    remove the first process from the queue

    run the process for its time quantum

    if (process is complete) {

      remove the process from the system

    } else if (process blocked for I/O) {

      place the process at the back of the same queue

    } else if (process used up its quantum) {

      demote the process to the next lower priority queue

    }

  } else {

    wait for a process to arrive

  }


  check for aging of processes in lower priority queues
```

```
  adjust priorities of processes as necessary
}
```

In this pseudocode, the algorithm first initializes all the priority queues, sets the time quantum for each queue, and sets the initial queue priority. The algorithm then enters a loop that continues indefinitely, waiting for processes to arrive and scheduling them as necessary.

If any of the queues are non-empty, the algorithm selects the highest priority non-empty queue and removes the first process from the queue. The process is then run for its time quantum. If the process completes during its quantum, it is removed from the system. If the process blocks for I/O, it is placed at the back of the same queue. If the process uses up its quantum, it is demoted to the next lower priority queue.

If all the queues are empty, the algorithm waits for a process to arrive. Additionally, the algorithm checks for aging of processes in lower priority queues and adjusts their priorities as necessary. This ensures that processes that have been waiting for a long time are eventually given a chance

The above-mentioned factors make MLFQ scheduling a popular choice for operating systems. However, it is not perfect and has some potential drawbacks. One of the main issues with MLFQ scheduling is that it can lead to process starvation, where a low-priority process never gets a chance to execute if there are always high-priority processes in the system. Another issue is that the complexity of the algorithm can lead to higher overhead and longer response times.

Despite these potential drawbacks, MLFQ scheduling remains a popular choice for modern operating systems, particularly for systems that require high levels of concurrency and responsiveness. The ability to prioritize processes based on their behavior and requirements, combined with the flexibility of the algorithm, makes MLFQ scheduling an attractive option for many different types of systems.

Overall, MLFQ scheduling represents a significant advancement in the field of CPU scheduling, offering a flexible and effective way to manage system resources in complex and dynamic environments. As operating systems continue to evolve and become more complex, it is likely that MLFQ scheduling will remain a key part of their design and implementation.

2.6 Comparison of scheduling algorithms

In this chapter, we will compare and contrast the different CPU scheduling algorithms discussed in the previous chapters. We will evaluate them based on various criteria such as turnaround time, waiting time, response time, fairness, and throughput.

2.6.1 Turnaround Time

Turnaround time is the time taken to complete a process, from the moment it is submitted to the moment it is completed. A scheduling algorithm that minimizes the turnaround time is preferred. Among the algorithms discussed, SJF has the lowest average turnaround time since it schedules the shortest jobs first. FCFS has a high turnaround time, especially for long processes, as it schedules processes in the order they arrive.

2.6.2 Waiting Time

Waiting time is the time spent by a process waiting in the ready queue before it is scheduled to run. A scheduling algorithm that minimizes the waiting time is preferred. SJF also has the lowest average waiting time as it schedules shorter processes first. On the other hand, FCFS has a higher average waiting time, especially for long processes.

2.6.3 Response Time

Response time is the time taken for a process to start responding after it is submitted. A scheduling algorithm that minimizes the response time is preferred, especially for interactive systems. Round-robin has the lowest average response time as it schedules processes for short time slices, ensuring that each process gets a chance to run quickly. SJF has a high response time since it prioritizes short processes over long ones.

2.6.4 Fairness

Fairness refers to how evenly the CPU time is allocated among processes. A fair scheduling algorithm ensures that each process gets an equal share of the CPU time. Round-robin is the most fair algorithm as it schedules processes in a circular fashion, giving each process a fixed time slice. FCFS and SJF are not fair since they prioritize some processes over others.

2.6.5 Throughput

Throughput refers to the number of processes completed per unit time. A scheduling algorithm that maximizes the throughput is preferred. Round-robin has the highest throughput since it schedules processes for short time slices, ensuring that each process gets a chance to run quickly. SJF also has a high throughput since it schedules shorter processes first. FCFS has a lower throughput, especially for long processes.

Overall, the best scheduling algorithm depends on the specific requirements of the system. SJF is best suited for systems with short processes, while round-robin is best suited for interactive systems. Priority scheduling is useful in real-time systems where certain processes require priority over others. MLFQ is useful in systems with a mix of long and short processes.

In conclusion, the choice of scheduling algorithm depends on the specific requirements of the system. The scheduler should be designed to balance the competing goals of minimizing turnaround time, waiting time, and response time while also ensuring fairness and maximizing throughput.

3 Process and Thread Prioritization

In this chapter, we will be discussing the important topic of process and thread prioritization in CPU scheduling. We will begin by defining what process and thread priorities are and why they are important in the context of CPU scheduling. We will then explore the different methods of prioritization, including static priorities, dynamic priorities, and aging. By the end of this chapter, you will have a clear understanding of how prioritization plays a crucial role in ensuring optimal performance and resource utilization in modern operating systems. So let's dive in!

In operating systems, process and thread priorities are an essential aspect of CPU scheduling. They are used to determine which processes or threads should be given access to the CPU and in what order. The priority of a process or thread is a numerical value that indicates its relative importance compared to other processes or threads.

3.1 Process Priorities

Process priorities are set by the operating system and can be fixed or dynamic. Fixed priorities are assigned to processes when they are created and do not change during the lifetime of the process. Dynamic priorities can change based on the behavior of the process, the system load, or other factors.

3.2 Thread Priorities

Thread priorities are a more fine-grained form of prioritization, allowing the operating system to make scheduling decisions on a per-thread basis. Each thread within a process can be assigned its own priority level, which is used by the scheduler to determine when and for how long the thread will run.

3.3 Importance of Prioritization in CPU Scheduling

Prioritization is crucial in CPU scheduling because it allows the operating system to make intelligent decisions about which processes or threads should be given access to the CPU at any given time. Without prioritization, the system would be unable to distinguish between critical processes and less important ones, leading to inefficient use of system resources and potential performance issues.

3.4 Methods of Prioritization

There are several methods for setting process and thread priorities, including static priorities, dynamic priorities, and aging.

3.4.1 Static Priorities

Static priorities are fixed values that are assigned to processes or threads when they are created. They do not change during the lifetime of the process or thread and are typically set by the system administrator or the program developer. Static priorities are useful for ensuring that critical processes or threads always have access to the CPU, but they can also lead to inefficient use of system resources if not set correctly.

3.4.2 Dynamic Priorities

Dynamic priorities change over time based on the behavior of the process or thread, the system load, or other factors. Dynamic priorities allow the system to adapt to changing conditions and ensure that critical processes or threads receive the resources they need to complete their tasks efficiently.

3.4.3 Aging

Aging is a technique used in some scheduling algorithms to prevent processes or threads from being starved of resources. As a process or thread waits in a queue, its priority may increase over time, ensuring that it eventually receives the resources it needs to complete its task.

In conclusion, process and thread priorities are a critical component of CPU scheduling in operating systems. They allow the system to make intelligent decisions about which processes or threads should be given access to the CPU and when, ensuring that critical processes receive the resources they need to complete their tasks efficiently. By using methods such as static and dynamic priorities, and aging, the operating system can provide a fair and efficient scheduling environment for all processes and threads.

4 Scheduling in Multiprocessor and Multicore Systems

As computer hardware has continued to advance, we have seen a shift towards using multiple processors or cores within a single machine, which can greatly increase the amount of work that can be performed simultaneously. However, this also creates new challenges in terms of how to efficiently allocate and manage resources between different processes or threads.

In this chapter, we will discuss the various methods of scheduling in multiprocessor and multicore systems. This includes approaches such as load balancing, processor affinity, and gang scheduling. We will also examine the trade-offs involved in these approaches, including considerations such as communication overhead, cache locality, and fairness.

Overall, this chapter aims to provide a comprehensive overview of the key issues involved in scheduling in modern multiprocessor and multicore systems. By understanding these concepts, you will be better equipped to develop efficient and effective scheduling strategies for your own applications and systems.

4.1 Definition of multiprocessor and multicore systems

Multiprocessor and multicore systems are computing systems that contain more than one processor or core. A processor or core is a central processing unit (CPU) that can execute instructions and carry out computations. Traditional computers typically have a single processor or core, which means that they can only execute one task at a time. Multiprocessor and multicore systems, on the other hand, have the ability to execute multiple tasks simultaneously, leading to an increase in overall system performance.

Multiprocessor systems can be classified into two main categories: tightly coupled and loosely coupled systems. In tightly coupled systems, the processors share the same memory and communicate with each other through a bus or a switch. This allows them to work together on a single task, making them ideal for applications that require a lot of computing power, such as scientific simulations, financial modeling, and database management.

Loosely coupled systems, on the other hand, consist of multiple independent processors that communicate with each other through a

network. Each processor has its own memory and can execute its own set of instructions, making them ideal for applications that require high availability and fault tolerance, such as web servers and data centers.

Multicore systems, on the other hand, consist of a single processor that contains multiple cores. Each core is capable of executing instructions and carrying out computations independently, which allows for parallelism within a single processor. This parallelism results in an increase in performance without the need for additional hardware, making multicore systems ideal for desktop computers, laptops, and mobile devices.

Multiprocessor and multicore systems require specialized hardware and software to effectively utilize their capabilities. Operating systems need to be designed to take advantage of the multiple processors or cores, and software applications need to be written with parallelism in mind. Additionally, there may be issues with scalability, load balancing, and synchronization that need to be addressed to ensure optimal performance.

In conclusion, multiprocessor and multicore systems are computing systems that contain multiple processors or cores, allowing for increased performance and parallelism. Tightly coupled systems share memory and communicate through a bus or switch, while loosely coupled systems communicate through a network. Multicore systems contain multiple cores within a single processor. To effectively utilize these systems, specialized hardware and software are required, and issues with scalability, load balancing, and synchronization need to be addressed.

4.2 Importance of Scheduling in Multiprocessor and Multicore Systems

In a multiprocessor or multicore system, tasks can be executed in parallel on different processors or cores. However, the system must be able to manage the tasks efficiently, ensuring that each processor or core is kept busy and that tasks are completed in a timely manner. The scheduling algorithm must be designed to take into account the number of processors or cores available, the nature of the tasks to be executed, and the resources required by each task. Without efficient scheduling, the system will not be able to make full use of all available resources, leading to lower performance and reduced efficiency.

4.3 Methods of Scheduling in Multiprocessor and Multicore Systems

There are several methods of scheduling in multiprocessor and multicore systems. One approach is to use a centralized scheduler, where all scheduling decisions are made by a single entity. Another approach is to use a distributed scheduler, where scheduling decisions are made by each processor or core independently. The distributed scheduler can be further divided into two categories: homogeneous and heterogeneous. In a homogeneous system, all processors or cores have identical characteristics, while in a heterogeneous system, processors or cores have different characteristics such as processing speed, cache size, and memory access.

4.4 Challenges in Scheduling in Multiprocessor and Multicore Systems

Scheduling in multiprocessor and multicore systems presents several challenges. One of the main challenges is load balancing, which involves distributing tasks evenly across all processors or cores. If tasks are not distributed evenly, some processors or cores may be idle while others are overloaded, leading to reduced performance. Another challenge is synchronization, which involves coordinating the activities of multiple processors or cores to ensure that they do not interfere with each other.

Scheduling in multiprocessor and multicore systems is a critical component of operating system design. Efficient scheduling algorithms are necessary to ensure that all processors or cores are utilized optimally and that tasks are completed in a timely manner. With the continued growth in computing power, scheduling in multiprocessor and multicore systems will remain a vital area of research and development in the field of operating systems.

5 Case Study: CPU Scheduling in Linux Operating System

In modern operating systems, CPU scheduling is a critical component responsible for assigning processes and threads to available CPU resources. The efficient utilization of CPU resources is essential to achieve optimal system performance, responsiveness, and fairness. Therefore, operating system designers have implemented various scheduling algorithms and methods to achieve these goals.

In this chapter, we will discuss a case study of CPU scheduling in the Linux operating system. We will examine the features of the Linux scheduler, compare it with other operating systems, and evaluate its impact on system performance, responsiveness, and fairness.

First, we will provide an overview of CPU scheduling in general and its importance in operating systems. Then, we will discuss the methods and algorithms used for CPU scheduling, including First-Come-First-Serve (FCFS), Shortest-Job-First (SJF), Priority Scheduling, Round-Robin (RR), and Multilevel Feedback Queue (MLFQ). We will also examine process and thread prioritization, including static and dynamic priorities and aging.

Next, we will examine CPU scheduling in multiprocessor and multicore systems, which are becoming increasingly common in modern computing environments. We will discuss the methods used to schedule processes and threads across multiple processors and cores.

Finally, we will focus on the Linux operating system and examine its CPU scheduling features, including its Completely Fair Scheduler (CFS). We will compare the Linux scheduler with other operating systems and evaluate its impact on system performance, responsiveness, and fairness.

Overall, this chapter will provide an in-depth understanding of CPU scheduling, its methods and algorithms, and its critical role in achieving optimal system performance, responsiveness, and fairness. We will also gain insights into the Linux scheduler and its impact on system performance, making this chapter an essential read for anyone interested in operating system design and performance.

5.1 Overview of Linux CPU scheduling

Linux is one of the most popular operating systems that is widely used in various applications ranging from personal computers to data centers. The Linux kernel has evolved significantly over the years, and so has its CPU scheduling algorithm. The scheduling algorithm in Linux is responsible for determining which processes should run and for how long. The algorithm used in Linux is a combination of several scheduling policies that operate in a hierarchical fashion. In this chapter, we will

discuss the overview of the Linux CPU scheduling algorithm, its design principles, and the policies used in the algorithm.

The Linux kernel implements a preemptive, priority-based scheduling algorithm. This means that the scheduler is responsible for preempting a running process and allowing another process to run if it has a higher priority. The priority of a process is determined by several factors, including the process's nice value, which is a user-defined parameter that ranges from -20 to +19. A higher nice value indicates that the process is less important, whereas a lower nice value indicates that the process is more important.

The Linux scheduler uses a runqueue data structure to keep track of the processes that are ready to run. Each runqueue contains a set of processes that have the same priority. The scheduler selects the highest priority runqueue that is not empty and selects the process at the head of the queue to run. If there are multiple processes in the runqueue with the same priority, the scheduler uses a round-robin scheduling policy to ensure that each process gets a fair share of CPU time.

The Linux scheduler has several design principles that govern its operation. These include fairness, responsiveness, and scalability. Fairness means that each process should get a fair share of CPU time, regardless of its priority or the resources it is using. Responsiveness means that the scheduler should be able to quickly respond to changes in the system load or to user requests. Scalability means that the scheduler should be able to handle a large number of processes and threads efficiently.

5.2 Policies Used in the Linux CPU Scheduling Algorithm:

The Linux scheduler uses several policies to determine the priority of a process. These policies include the Completely Fair Scheduler (CFS), the

Round Robin Scheduler, the Real-time Scheduler, and the Idle Process Scheduler.

The CFS is the default scheduler in Linux and is designed to provide fairness and responsiveness. The CFS uses a red-black tree data structure to keep track of the processes in the system. Each node in the tree represents a process, and the nodes are sorted based on the process's virtual runtime, which is a measure of the CPU time the process has received. The process with the smallest virtual runtime is selected to run next.

The Round Robin Scheduler is used to provide fair sharing of the CPU among processes of the same priority. Each process is given a time slice, and the scheduler ensures that each process gets a fair share of CPU time by using a round-robin policy to switch between processes when their time slice is up.

The Real-time Scheduler is used to provide guaranteed response times for time-critical applications. Real-time processes are given a higher priority than other processes and are scheduled first. The Real-time Scheduler is divided into two classes: the SCHED_FIFO and SCHED_RR. SCHED_FIFO is a First-In-First-Out (FIFO) scheduling policy that is used for processes that need to run for a long time without being preempted. SCHED_RR is a Round-Robin scheduling policy that is used for processes that need to be preempted after a certain amount of time.

The Idle Process Scheduler is used to keep the CPU busy when there are no processes to run. The Idle Process Scheduler runs a special idle process that executes when there are no other processes to run. The idle process consumes very little CPU time and is used to keep the

The Completely Fair Scheduler (CFS) is another popular scheduling algorithm used in Linux. It aims to give each process a fair share of the CPU based on the amount of work it has to do. The CFS maintains a red-black tree of processes, sorted by their virtual runtime. The virtual runtime of a process is the amount of time it has spent running on the

CPU divided by its priority. This way, the CFS ensures that every process gets an equal share of the CPU, regardless of its priority.

Another interesting feature of the CFS is that it is not limited to a fixed time slice like Round-Robin scheduling. Instead, it dynamically adjusts the time slice of each process based on the number of runnable processes in the system. This ensures that the CPU time is used efficiently, and no process is left waiting for too long.

In addition to the CFS, Linux also supports other scheduling algorithms such as the Real-Time (RT) scheduler and the Completely Fair Queuing (CFQ) scheduler. The RT scheduler is designed for real-time applications that require a guaranteed amount of CPU time, while the CFQ scheduler is optimized for disk I/O performance.

Overall, Linux CPU scheduling is a complex and evolving field, with a wide range of algorithms and techniques to choose from. The Linux kernel developers continue to refine and improve the scheduling subsystem, in order to provide the best possible performance, responsiveness, and fairness for all users and processes.

5.3 Comparison with CPU scheduling in other operating systems

As we have seen, Linux CPU scheduling is a complex and sophisticated system that balances multiple factors to provide efficient and fair CPU allocation. But how does it compare with CPU scheduling in other operating systems?

Let's start with the most well-known operating system, Microsoft Windows. The CPU scheduling algorithm in Windows is also based on priority levels, but it uses a feedback mechanism to adjust the priority of a process based on its recent behavior. This means that if a process has been using the CPU heavily, its priority will be reduced to prevent it

from monopolizing the CPU for too long. The Windows scheduler also allows for real-time priority levels, which can be used for critical tasks that require immediate attention.

In macOS, the CPU scheduling algorithm is similar to that of Linux in that it uses a multi-level feedback queue, but it places a greater emphasis on interactivity. This means that macOS prioritizes processes that are likely to generate user-visible output, such as a keystroke or a mouse click. macOS also uses a technique called thread throttling to limit the CPU usage of background processes and prevent them from slowing down foreground processes.

In the realm of real-time operating systems, such as those used in embedded systems and robotics, CPU scheduling takes on an even greater level of importance. Real-time operating systems require precise timing and predictable response times, and as such, they often use a fixed-priority scheduling algorithm. This means that each task is assigned a priority level, and the scheduler always selects the task with the highest priority to execute next.

Overall, while the specific details of CPU scheduling algorithms may differ between operating systems, the goal is always the same: to provide efficient and fair allocation of the CPU's processing power. Each operating system has its own unique approach to achieving this goal, based on its particular requirements and design philosophy.

6 Conclusion

In conclusion, CPU scheduling is a crucial aspect of modern operating systems, allowing for efficient utilization of system resources and enabling concurrency and parallelism. The different scheduling algorithms, process and thread prioritization methods, and scheduling techniques for multiprocessor and multicore systems offer a range of options for achieving the goals of CPU scheduling.

While each approach has its advantages and disadvantages, the selection of the most appropriate scheduling technique depends on the specific system requirements and workload characteristics. Additionally, case studies like Linux CPU scheduling provide valuable insights into the design decisions and performance trade-offs involved in real-world implementations of CPU scheduling.

Overall, a thorough understanding of CPU scheduling is essential for developing efficient and responsive operating systems that can handle the diverse computing needs of today's applications.