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.

1.4 Process Behavior

One of the primary characteristics of process behavior is that most processes alternate bursts of computing with I/O requests. Typically, the CPU runs for a period, then a system call is made to read from or write to a file. Once the system call completes, the CPU resumes computing until more data is needed or there is more data to write. It's important to note that certain I/O activities are still considered computing, such as updating the screen with video RAM because the CPU is still in use.

I/O operations in this context refer to the process entering the blocked state while waiting for an external device to complete its work. This can include waiting for a file to load or saving a file to disk. During this time, the process is not actively running, and the CPU can work on other tasks.

Understanding process behavior is essential for optimizing system performance. The operating system can use this knowledge to schedule processes efficiently, prioritizing those that are actively computing and delaying those that are in a blocked state. By doing so, the system can maximize its utilization of resources, ensuring that every process receives the necessary resources to complete its work.

1.5 Scheduling decisions

Scheduling decisions are an integral part of operating systems. When to schedule a process is one of the most critical questions an operating system needs to answer. In this chapter, we will explore the four key situations when scheduling decisions need to be made.

The first situation occurs when a new process is created. The scheduler must decide whether to run the parent process or the child process. Both processes are in the ready state, and the scheduler can legitimately choose either process to run next.

The second situation is when a process exits. Since the process can no longer run, the scheduler must select another process from the set of ready processes. If no process is ready, a system-supplied idle process is run.

The third situation arises when a process blocks on I/O, a semaphore, or some other reason. Another process must be selected to run. Sometimes the reason for blocking may play a role in the selection process. For example, if an important process is waiting for another process to exit its critical region, letting that process run next will enable the important process to continue.

The fourth situation occurs when an I/O interrupt happens. If the interrupt came from an I/O device that has completed its work, a process that was blocked waiting for the I/O may now be ready to run. The scheduler must decide whether to run the newly ready process, the process that was running at the time of the interrupt, or some third process.

The scheduler must make these decisions promptly and efficiently to ensure the system operates optimally. To achieve this, scheduling algorithms employ various strategies, such as round-robin, priority-based scheduling, and lottery scheduling. In conclusion, scheduling decisions are critical to the smooth operation of an operating system, and the timing of such decisions is influenced by various factors.

1.6 Categories of Scheduling

When designing a scheduling algorithm, the environment in which it will operate must be taken into account. There are three main categories of environments: batch, interactive, and real-time.

In a batch environment, jobs are submitted in advance and then executed without user interaction. The goal of the scheduler in this environment is to maximize throughput, or the number of jobs

completed per unit of time. A typical algorithm used in batch environments is the First-Come, First-Served (FCFS) algorithm, in which jobs are executed in the order they are received.

In an interactive environment, the goal of the scheduler is to minimize response time, or the time between when a user submits a request and when a response is returned. In this environment, users are directly interacting with the system, so a fast response time is essential for a good user experience. Interactive scheduling algorithms often use time-sharing, where each user is allocated a slice of time in which they can interact with the system. One common algorithm used in interactive environments is the Round-Robin algorithm, where each user is given a fixed time slice in which to execute their jobs.

In a real-time environment, the goal of the scheduler is to ensure that critical tasks are completed within their deadlines. In this environment, there are hard deadlines that must be met, such as controlling a physical process like an assembly line or a nuclear power plant. Real-time scheduling algorithms must take into account the importance of meeting these deadlines and ensure that critical tasks are given priority over non-critical tasks. One common algorithm used in real-time environments is the Earliest Deadline First (EDF) algorithm, in which the job with the earliest deadline is given priority.

It is important to note that these categories are not mutually exclusive, and many systems operate in a mixed environment. For example, a web server may have both batch jobs running in the background and interactive requests from users. In this case, the scheduler must balance the needs of both environments to ensure the system operates efficiently and responsively.

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.

Example:

Input:

Process 1: Arrival Time = 0, Burst Time = 4

Process 2: Arrival Time = 2, Burst Time = 2

Process 3: Arrival Time = 4, Burst Time = 3

Process 4: Arrival Time = 6, Burst Time = 1

Output:

Process 1: Waiting Time = 0, Turnaround Time = 4

Process 2: Waiting Time = 2, Turnaround Time = 4

Process 3: Waiting Time = 4, Turnaround Time = 7

Process 4: Waiting Time = 7, Turnaround Time = 8

Explanation:

Process 1 arrives at time 0 and executes for 4 units of time.

Process 2 arrives at time 2 but has to wait for 2 units of time (until process 1 completes) before executing for 2 units of time.

Process 3 arrives at time 4 but has to wait for 4 units of time (until process 2 completes) before executing for 3 units of time.

Process 4 arrives at time 6 but has to wait for 7 units of time (until process 3 completes) before executing for 1 unit of time.

Waiting time for each process is calculated as the time spent waiting in the ready queue before executing, while turnaround time is the total time spent by a process from arrival to completion (i.e., waiting time + burst time).

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.
```

Example: Here's an example input and output for the SJF (Shortest Job First) scheduling algorithm:

Input:

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	2
P4	3	4

Output:

Process	Arrival	Burst	Completion	Turnaround	Waiting
P1	0	5	5	5	0
P3	2	2	7	5	3
P2	1	3	10	9	6
P4	3	4	14	11	7

Explanation:

The SJF algorithm schedules processes based on their burst time, with the shortest job being scheduled first. In this example, the arrival time and burst time for each process are provided in the input table.

Initially, there are no processes in the ready queue, and P1 arrives at time 0. P1 is the only process in the queue and starts executing immediately. At time 1, P2 arrives and its burst time is shorter than P1's remaining burst time, so P2 is scheduled next. At time 2, P3 arrives and its burst time is shorter than P1's remaining burst time, so P3 is scheduled next. At time 3, P4 arrives and its burst time is shorter than P1's remaining burst time, but longer than P3's remaining burst time, so P1 continues executing.

After P1 completes, P3 is the shortest job in the ready queue and starts executing. P2 is scheduled next as its burst time is shorter than P4's remaining burst time. Finally, P4 completes the execution.

The output table shows the completion time, turnaround time, and waiting time for each process. The completion time is the time when the process finishes execution. The turnaround time is the difference between the completion time and the arrival time, which represents the time a process spends in the system. The waiting time is the difference between the turnaround time and the burst time, which represents the time a process spends waiting in the ready queue.

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.
```

Example: Here's an example input and output for the preemptive SJF (Shortest Job First) scheduling algorithm:

Input:

Process	Arrival Time	Burst Time
P1	0	5
P2	2	3
P3	4	4

P4	6	2

Output:

Time	Process	Remaining Burst Time
0	P1	5
1	P1	4
2	P2	3
3	P2	2
4	P2	1
5	P1	3
6	P4	2
7	P4	1
8	P3	4
9	P3	3
10	P3	2
11	P3	1
12	P1	2
13	P1	1

In this scenario, the pre-emptive sjf algorithm selects the process with the shortest remaining burst time. At time 0, process P1 arrives and starts executing. At time 2, process P2 arrives, but since P1 has a shorter remaining burst time, the scheduler pre-empts P1 and allows P2 to execute. At time 4, process P3 arrives, but P2 still has a shorter remaining burst time, so P3 is not selected. At time 5, P1 is selected again since it has the shortest remaining burst time. At time 6, process P4 arrives and has a shorter remaining burst time than P1, so P1 is pre-

empted and P4 starts executing. At time 8, process P3 finally gets selected since it has the shortest remaining burst time. The remaining burst times for each process are shown in the output table, and the algorithm ends at time 13 when all processes have completed.

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 Shortest Remaining Time Next (SRTN)

Shortest Remaining Time Next (SRTN) is a preemptive version of the shortest job first scheduling algorithm. It is a CPU scheduling algorithm that is used in operating systems to minimize the average waiting time for processes. The idea behind this algorithm is to always select the process that has the shortest remaining burst time. Burst time is the amount of time a process needs to complete its execution.

In SRTN, the scheduler keeps track of the remaining time for each process in the ready queue. Whenever a new process arrives or the running process becomes blocked, the scheduler selects the process with the shortest remaining time to execute. If a process with a shorter

burst time arrives while another process is running, the running process is preempted, and the new process is executed.

SRTN is an optimal algorithm because it reduces the average waiting time for processes. However, it requires knowledge of the total execution time for each process in advance, which is not always available. In addition, it suffers from the same problem as SJF where long-running processes may suffer from starvation.

SRTN can be implemented using priority queues, where processes with shorter remaining times have higher priority. This ensures that shorter processes are always executed first, regardless of the order in which they arrive.

Overall, SRTN is a powerful scheduling algorithm that can improve the performance of the system, especially for processes with short burst times. However, it requires accurate estimation of the remaining execution time of each process, which can be difficult to obtain in practice.

2.4 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.

2.5 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.

Example: Here's an example input and output for the Round Robin scheduling algorithm:

Input:

Process	Arrival Time	Burst Time
P1	0	10
P2	1	4
P3	2	3
P4	3	5

Time Quantum: 2

Output:

Time	Process	Remaining Time
0	P1	8
2	P2	2
4	P3	1
6	P4	3
8	P1	6
10	P2	0
12	P3	0
13	P4	1
15	P1	4
17	P4	0
18	P1	2
19	P1	0

In this example, there are four processes arriving at different times with different burst times. The time quantum is set to 2 units.

At time 0, the first process P1 is scheduled and given the full burst time of 10 units, since it is the only process present.

At time 2, P2 arrives and is scheduled, but is only given 2 units of CPU time, as that is the time quantum. At the end of its time quantum, P2's remaining time is 2 units.

At time 4, P3 arrives and is scheduled for 2 units, P3's remaining time is 1 unit.

At time 6, P4 arrives and is scheduled for 2 units, P4's remaining time is 3 units.

At time 8, P1 is scheduled again P1's remaining time, which is now 6 units.

At time 10, P2 is scheduled again and completes its execution with 0 remaining time.

At time 12, P3 is scheduled again and completes its execution with 0 remaining time.

At time 13, P4 is scheduled again, P4's remaining time is now 1 units.

At time 15, P1 is scheduled again P1's remaining time, which is now 4 units.

At time 17, P4 is scheduled again and completes its execution with 0 remaining time.

At time 18, P1 is scheduled again P1's remaining time, which is now 2 units.

At time 19, P1 completes its execution with 0 remaining time, resulting in all processes being completed.

2.6 Multiple queues

Multiple queues are a common approach for scheduling processes in many modern operating systems. The basic idea is to have several separate queues of processes waiting to be executed. The processes are grouped into the different queues based on their priority level, and the scheduler selects processes from each queue in turn. This approach allows the scheduler to give higher priority to certain processes while still ensuring that all processes get some CPU time.

One of the earliest examples of a priority scheduler using multiple queues was in the CTSS operating system, which ran on the IBM 7094 computer. CTSS had a unique problem in that process switching was slow due to the limited memory capacity of the 7094. Thus, the

designers of CTSS came up with a clever solution to optimize process scheduling. They assigned different priority classes to the processes, where higher-priority processes were given a larger quantum to run than lower-priority processes. The highest-priority processes were run for one quantum, while lower-priority processes were given longer quanta. When a process used up all the quanta allocated to it, it was moved down one class.

Modern operating systems use a similar approach with multiple priority queues. Processes are assigned to a specific queue based on their priority level, and the scheduler selects processes from each queue in turn. Some operating systems, such as Linux, use a round-robin approach, where each process in a queue is given a fixed amount of CPU time before the scheduler moves on to the next process. Other operating systems, such as Windows, use a priority-based approach, where processes in higher-priority queues are given more CPU time than those in lower-priority queues.

The advantage of using multiple queues is that it allows the scheduler to give higher priority to certain processes while still ensuring that all processes get some CPU time. For example, in a real-time system, processes with hard deadlines may be assigned to a higher-priority queue than processes with soft deadlines. Similarly, in a desktop environment, interactive processes, such as the user's mouse and keyboard input, may be assigned to a higher-priority queue than background processes, such as file backups.

In conclusion, multiple queues are an effective and efficient approach for scheduling processes in modern operating systems. They allow the scheduler to give higher priority to certain processes while still ensuring that all processes get some CPU time. The use of multiple queues has been a standard approach in operating systems since the early days of computing, and it remains a key component of modern operating system design.

Example: Here's an example Java code that demonstrates multiple queues scheduling algorithm:

```
import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;


public class MultipleQueuesScheduler {


    // Define constants for the number of priority queues and quantum values

    private static final int NUM_QUEUES = 3;

    private static final int[] QUANTUM_VALUES = {10, 20, 40};


    // Create an array of queues to represent the multiple priority queues

    private Queue<Process>[] queues = new Queue[NUM_QUEUES];


    // Constructor to initialize the queues

    public MultipleQueuesScheduler() {

        for (int i = 0; i < NUM_QUEUES; i++) {

            queues[i] = new LinkedList<>();

        }

    }


    // Method to add a process to the appropriate queue based on its priority

    public void addProcess(Process process) {
```

```
        int priority = process.getPriority();
        if (priority < NUM_QUEUES) {
            queues[priority].add(process);
        } else {
            queues[NUM_QUEUES - 1].add(process);
        }
    }

    // Method to run the scheduler and execute the processes
    public void run() {
        for (int i = 0; i < NUM_QUEUES; i++) {
            int quantum = QUANTUM_VALUES[i];
            Queue<Process> currentQueue = queues[i];
            while (!currentQueue.isEmpty()) {
                Process currentProcess = currentQueue.remove();
                int remainingTime =
currentProcess.getRemainingTime();
                if (remainingTime <= quantum) {
                    currentProcess.execute(remainingTime);
                } else {
                    currentProcess.execute(quantum);
                    currentProcess.setRemainingTime(remainingTime
- quantum);
                    currentQueue.add(currentProcess);
                }
            }
```

```
    }
}

// Inner class to represent a process
private static class Process {
    private int priority;
    private int remainingTime;

    public Process(int priority, int remainingTime) {
        this.priority = priority;
        this.remainingTime = remainingTime;
    }

    public int getPriority() {
        return priority;
    }

    public int getRemainingTime() {
        return remainingTime;
    }

    public void setRemainingTime(int remainingTime) {
        this.remainingTime = remainingTime;
    }
```

```
        public void execute(int time) {
            System.out.println("Executing process with priority " + priority + " for " + time + " time units.");
        }
    }

    // Main method to test the scheduler
    public static void main(String[] args) {
        MultipleQueuesScheduler scheduler = new MultipleQueuesScheduler();
        scheduler.addProcess(new Process(0, 30));
        scheduler.addProcess(new Process(2, 60));
        scheduler.addProcess(new Process(1, 20));
        scheduler.addProcess(new Process(3, 50));
        scheduler.run();
    }
}
```

In this example, the MultipleQueuesScheduler class represents the scheduler that uses multiple priority queues with different quantum values. The Process inner class represents a process with a priority level and a remaining execution time. The addProcess method adds a process to the appropriate priority queue based on its priority level, and the run method executes the processes in each priority queue using the corresponding quantum value.

In the main method, we create a MultipleQueuesScheduler instance and add four processes with different priority levels and remaining execution times. Finally, we call the run method to execute the processes according to their priority levels and the quantum values of the priority queues.

2.7 Shortest process next (SPN)

Shortest process next (SPN) is a scheduling algorithm that is similar to shortest job first (SJF). The difference is that in SPN, the scheduler selects the process with the shortest expected processing time instead of the shortest actual processing time.

The expected processing time is calculated based on the process's previous execution history. The algorithm works well for interactive systems, where processes typically execute a series of short tasks.

When a new process arrives, the scheduler calculates the expected processing time of the process based on its previous execution history. The process with the shortest expected processing time is selected to run next. If a new process arrives with a shorter expected processing time than the currently running process, the scheduler preempts the running process and starts the new process.

One of the main advantages of SPN is that it provides a good balance between short response times and high throughput. It ensures that short processes are executed first, thereby reducing response time. At the same time, it does not ignore longer processes entirely, ensuring that they get executed too.

One disadvantage of SPN is that it requires an accurate estimate of the expected processing time. If the estimate is inaccurate, the algorithm may select the wrong process, leading to poor performance.

Example: Here's an example Java code that demonstrates the shortest process next (SPN) scheduling algorithm:

```
import java.util.*;


public class SPNScheduler {
```

```
    public static void main(String[] args) {


        // Create a list of processes with their arrival times and
burst times

        int[][] processes = {{1, 0, 6}, {2, 1, 8}, {3, 2, 7}, {4,
3, 3}, {5, 4, 4}};


        // Sort the processes by their arrival times

        Arrays.sort(processes,    Comparator.comparingInt(a    ->
a[1]));


        // Initialize variables for the current time and total
waiting time

        int currentTime = 0;

        int totalWaitingTime = 0;


        // Create a priority queue to store the processes by their
burst times

        PriorityQueue<int[]>          queue          =          new
PriorityQueue<>(Comparator.comparingInt(a -> a[2]));


        // Loop through each process

        for (int i = 0; i < processes.length; i++) {

            int[] process = processes[i];


            // If the process has not arrived yet, skip it

            if (process[1] > currentTime) {

                i--;
```

```
                currentTime++;
                continue;
            }

            // Add the process to the queue
            queue.add(process);

            // Get the shortest process from the queue
            int[] shortestProcess = queue.poll();

            // Calculate the waiting time for the process
            int waitingTime = currentTime - shortestProcess[1];

            // Add the waiting time to the total waiting time
            totalWaitingTime += waitingTime;

            // Increment the current time by the process's burst time
            currentTime += shortestProcess[2];
        }

        // Calculate the average waiting time
        double averageWaitingTime = (double) totalWaitingTime / processes.length;

        // Print the average waiting time
```

```
        System.out.println("Average waiting time: " +
averageWaitingTime);
    }


}
```

This code uses a two-dimensional array to represent the processes, with each row containing the process ID, arrival time, and burst time. It sorts the processes by their arrival times, and then loops through each process, adding it to a priority queue sorted by its burst time. It then gets the shortest process from the queue and calculates the waiting time for that process. Finally, it increments the current time by the process's burst time and repeats the process until all processes have been executed. At the end, it calculates the average waiting time and prints it to the console.

2.8 Guaranteed scheduling

Guaranteed scheduling is a unique approach to scheduling that makes actual promises to users about their computer system's performance. This approach aims to provide a guarantee of a specific level of system performance for each user or process.

One of the most realistic and straightforward guarantees that can be made is that each user or process will receive a specific portion of the CPU power. This is based on the idea that if n users are logged in, then each user will receive approximately 1/n of the CPU power. Similarly, if n processes are running, then all things being equal, each process should get 1/n of the CPU cycles.

The goal of guaranteed scheduling is to ensure that users or processes have predictable and consistent performance, which is particularly important in time-critical environments. For example, in real-time systems, where the system must respond to external events within a

specified time frame, guaranteed scheduling can provide the assurance that processes will be executed in a timely and predictable manner.

One of the challenges of guaranteed scheduling is ensuring that the guarantees can be met. In a system with multiple users or processes, it can be challenging to allocate CPU resources fairly and ensure that each user or process receives their guaranteed share. In addition, the actual amount of CPU power required by each user or process can vary over time, which can make it difficult to maintain the promised level of performance.

To implement guaranteed scheduling, an operating system may use a variety of techniques, such as priority-based scheduling and time-slicing. For example, in a priority-based scheduling approach, each user or process is assigned a priority level, which determines how much CPU power they will receive. In a time-slicing approach, the CPU is divided into time slices, and each user or process is allocated a specific amount of CPU time within each time slice.

In summary, guaranteed scheduling is an approach to scheduling that aims to provide users or processes with a specific level of performance. By making real promises to users about their system's performance, guaranteed scheduling can provide predictable and consistent performance, which is particularly important in time-critical environments. However, implementing guaranteed scheduling can be challenging, and requires careful management of CPU resources to ensure that promises are kept.

Example: The guaranteed scheduling algorithm is not something that can be implemented directly in Java, as it requires low-level operating system support to guarantee resource allocation. However, we can simulate the behavior of this algorithm in Java by using a simple algorithm that allocates CPU time equally to all running processes.

Here is an example Java code that demonstrates the basic idea of guaranteed scheduling:

```
import java.util.ArrayList;


public class GuaranteedScheduling {

    // Define a simple Process class that represents a running
process

    static class Process {

        private String name;

        private int cpuTime;


        public Process(String name, int cpuTime) {

            this.name = name;

            this.cpuTime = cpuTime;

        }


        public String getName() {

            return name;

        }


        public int getCpuTime() {

            return cpuTime;

        }


        public void setCpuTime(int cpuTime) {

            this.cpuTime = cpuTime;

        }

    }
```

```
    public static void main(String[] args) {
        // Create a list of processes to run
        ArrayList<Process> processes = new ArrayList<>();
        processes.add(new Process("Process 1", 5));
        processes.add(new Process("Process 2", 2));
        processes.add(new Process("Process 3", 4));

        // Allocate CPU time equally to all processes
        int timeQuantum = 1;
        int totalTime = processes.size();
        int time = 0;
        while (!processes.isEmpty()) {
            Process p = processes.remove(0);
            System.out.println("Running " + p.getName() + " (CPU
time left: " + p.getCpuTime() + ")");
            p.setCpuTime(p.getCpuTime() - timeQuantum);
            time += timeQuantum;
            if (p.getCpuTime() > 0) {
                processes.add(p);
            } else {
                System.out.println(p.getName() + " completed at
time " + time);
            }
        }
    }
```

```
}
```

In this example, we create a list of three processes, each with a different amount of CPU time required. We then simulate guaranteed scheduling by allocating CPU time equally to each process, with a time quantum of 1. We run each process in turn until it has completed, and then move on to the next process in the list.

Note that this code is just a simple example, and does not actually guarantee equal allocation of CPU time in a real-world operating system. However, it should give you an idea of how the guaranteed scheduling algorithm might work in practice.

2.9 Lottery scheduling

One of the most elegant and innovative scheduling algorithms that exists is lottery scheduling. Its use of randomness is one of its key features, and it offers at least three significant advantages over more traditional approaches.

Firstly, randomness avoids strange corner-case behaviors that a more traditional algorithm may struggle to handle. For instance, consider the LRU (least recently used) replacement policy, which is examined in greater depth in a later chapter on virtual memory. Although LRU is often a good replacement algorithm, it can achieve worst-case performance for certain cyclic-sequential workloads. Random, on the other hand, has no such worst-case scenario.

Secondly, random is lightweight, requiring little state to track alternatives. In a traditional fair-share scheduling algorithm, monitoring how much CPU each process has received necessitates per-process accounting, which must be updated after each process runs. Random, on the other hand, only needs minimal per-process state (e.g., the number of tickets each process has).

Finally, random can be very quick. As long as generating a random number is rapid, making the decision is equally fast, and thus random can be utilized in several scenarios where speed is required. Of course, the quicker the need, the more random tends towards pseudo-random.

The use of lottery scheduling is a powerful technique that can be employed in various scenarios. This method assigns each process a certain number of tickets, and the scheduler then selects a winning ticket at random. The process with the matching ticket is then allocated the CPU. The higher the number of tickets a process has, the more likely it is to win the lottery and get access to the CPU. The process that wins the lottery can then execute for a set amount of time or until it blocks on I/O or some other event.

Lottery scheduling is a versatile technique that can be adapted to fit a wide range of situations. It has been used in operating systems for purposes such as load balancing, where it helps to ensure that the resources of a system are distributed fairly among processes. Additionally, it can be utilized for scheduling jobs on large-scale computer systems, which require the distribution of workloads across multiple nodes.

Example: Here's a pseudocode implementation of lottery scheduling:

```
// Each process is assigned a number of "lottery tickets" based on its priority

// The total number of tickets is fixed and can be adjusted as needed

// A process with more tickets is more likely to be selected for execution


struct Process {

  int pid;

  int tickets;
```

```
}

// Initialize the list of processes and their tickets
List<Process> processes;
processes.add(new Process(1, 10));
processes.add(new Process(2, 5));
processes.add(new Process(3, 3));

// Set the total number of tickets
int total_tickets = 18;

// Main loop for scheduling
while (true) {
  // Randomly select a winning ticket number
  int winner = random(1, total_tickets);

  // Iterate over the list of processes and check if the winner's
ticket matches
  for (Process process : processes) {
    if (winner <= process.tickets) {
      // Found the winning process, execute it
      execute(process.pid);
      break;
    }
    else {
```

```
      // Subtract the number of tickets for the current process
and continue
      winner -= process.tickets;
    }
  }
}
```

Note that this is a simplified pseudocode implementation for demonstration purposes and may not include all necessary features such as handling I/O requests, priority adjustments, and synchronization.

Example:

Input:

P1 with 10 tickets

P2 with 20 tickets

P3 with 30 tickets

P4 with 40 tickets

Output:

Total tickets: 100

Winning ticket: 36

Process P3 wins the lottery and is executed.

Winning ticket: 71

Process P4 wins the lottery and is executed.

Winning ticket: 16

Process P2 wins the lottery and is executed.

Winning ticket: 92

Process P4 wins the lottery and is executed.

Winning ticket: 6

Process P1 wins the lottery and is executed.

2.10Fair-share scheduling

Fair-share scheduling is a type of scheduling algorithm that takes into account the ownership of processes while scheduling them. In other words, it ensures that each user gets a fair share of the CPU time, irrespective of the number of processes they have running on the system.

The main idea behind fair-share scheduling is to allocate a portion of the CPU time to each user or group of users, based on the resources they are entitled to. For example, if two users are promised equal CPU time, say 50% each, then the system will ensure that they each get that amount of CPU time, regardless of the number of processes they have running.

One way to implement fair-share scheduling is to use a feedback control algorithm. This algorithm uses feedback from the system to adjust the amount of CPU time allocated to each user. The feedback can be in the form of CPU usage statistics, which are used to compute the relative shares of CPU time for each user. These shares are then used to determine how much CPU time each user should be allocated.

Another way to implement fair-share scheduling is to use a time-sharing algorithm, such as round-robin scheduling, in combination with a resource allocation mechanism. The resource allocation mechanism is used to allocate resources to each user, based on their entitlements. Once the resources are allocated, the time-sharing algorithm is used to schedule processes among the users.

Example: Here is an example Java code that demonstrates the fair share scheduling algorithm:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class FairShareScheduling {
    // Process class to store process details
    static class Process {
        String name;
        int time;
        String owner;

        Process(String name, int time, String owner) {
            this.name = name;
            this.time = time;
            this.owner = owner;
        }
    }

    // User class to store user details
    static class User {
        String name;
        double share;

        User(String name, double share) {
```

```
            this.name = name;
            this.share = share;
        }
    }

    // Scheduler method to assign CPU time to processes based on user shares
    public static void scheduler(List<Process> processes, List<User> users) {
        // Calculate total share of CPU time allocated to all users
        double totalShare = 0;
        for (User user : users) {
            totalShare += user.share;
        }

        // Calculate share of CPU time allocated to each user
        for (User user : users) {
            user.share = user.share / totalShare;
        }

        // Sort processes by owner to group processes by user
        Collections.sort(processes, Comparator.comparing(p -> p.owner));

        // Assign CPU time to processes based on user shares
        double[] shares = new double[users.size()];
        int[] counts = new int[users.size()];
```

```
        int idx = 0;
        for (Process process : processes) {
            while (!process.owner.equals(users.get(idx).name)) {
                idx++;
            }
            shares[idx] += users.get(idx).share;
            counts[idx]++;
            process.time -= (int) Math.ceil(users.get(idx).share *
process.time);
        }

        // Print CPU time assigned to each process
        for (int i = 0; i < processes.size(); i++) {
            Process process = processes.get(i);
            System.out.println("Process  "  +  process.name  +  "
assigned " + counts[i] * shares[i] * 100 + "% of CPU time.");
        }
    }

    public static void main(String[] args) {
        // Create processes
        List<Process> processes = new ArrayList<>();
        processes.add(new Process("P1", 20, "User1"));
        processes.add(new Process("P2", 30, "User2"));
        processes.add(new Process("P3", 40, "User1"));
        processes.add(new Process("P4", 10, "User3"));
```

```
        // Create users and assign share of CPU time
        List<User> users = new ArrayList<>();
        users.add(new User("User1", 2));
        users.add(new User("User2", 1));
        users.add(new User("User3", 1));

        // Call scheduler method to assign CPU time to processes based on user shares
        scheduler(processes, users);
    }
}
```

In this code, we first define a Process class to store the name, execution time, and owner of each process. We also define a User class to store the name and share of CPU time allocated to each user. We then define a scheduler method to assign CPU time to processes based on user shares.

In the scheduler method, we first calculate the total share of CPU time allocated to all users and then calculate the share of CPU time allocated to each user. We then sort the processes by owner to group processes by user. We assign CPU time to each process based on its owner's share of CPU time and subtract the assigned time from the process's execution time.

2.11 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.

2.12 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.

2.12.1 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.

2.12.2 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.

2.12.3 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.

2.12.4 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.

2.12.5 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.

2.13 Incorporating I/O

A scheduler plays a crucial role in determining which processes to run on a CPU at any given time. However, its job becomes even more challenging when a process initiates an I/O request. During this time, the process is blocked and waiting for I/O completion, which means the CPU remains idle. Therefore, the scheduler needs to make a decision to schedule another job on the CPU.

In addition, the scheduler must also decide what to do when the I/O completes. When the I/O operation completes, an interrupt is raised, and the OS moves the process that initiated the I/O from the blocked state back to the ready state. The scheduler then has to decide whether to continue running the currently-executing process or switch to the newly-ready process.

The OS should consider various factors while making scheduling decisions. For example, it could choose to prioritize processes with short I/O operations to minimize the wait time. The scheduler could also prioritize CPU-bound processes to maximize CPU utilization during I/O operations. Additionally, the OS could use priority-based scheduling, where higher priority processes are given preference over lower priority ones.

One approach that operating systems often use is the concept of priority-based scheduling with round-robin. In this approach, each process is assigned a priority level, and the scheduler runs the highest priority process first. If a process with a higher priority enters the ready state while a lower priority process is running, the scheduler preempts the lower priority process and switches to the higher priority process.

Furthermore, in the round-robin scheduling, the scheduler allocates a fixed time slice to each process, and if the process completes its time slice before its execution finishes, the process is moved to the back of

the ready queue. The process is then run again when it becomes the head of the ready queue.

Overall, the scheduler must make decisions based on a variety of factors, such as process priority, CPU utilization, and I/O wait times. By using efficient algorithms and techniques, the scheduler can maximize system throughput and minimize the waiting time for processes.

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.