Operating System Project

Submitted by:

Md Ahmed Raza Sofi (USN-3PD23CS065)

Hisham Khuram (USN-3PD23CS050)

Mohammed Abdul Maaz Hanan (USN-3PD23CS070)

Course Name:

Operating System

Instructor:

Sujata Terdal[HOD]

Submission Date:

30 December 2024

Understanding File Allocation Basics

Introduction to File Allocation in Operating Systems

File allocation is a fundamental concept in operating systems that deals with how data files are stored, organized, and managed on storage devices. Effective file management is crucial for system performance, ensuring that files can be created, accessed, modified, and deleted efficiently. At its core, file allocation involves determining how and where files are stored on disk, which directly impacts the speed and accessibility of those files.

One of the primary methods for managing file allocation is through dynamic memory allocation techniques. Linked lists, as demonstrated in the provided C code, serve as an excellent example of this approach. A linked list is a data structure where each element, known as a "node," contains data and a pointer to the next node in the sequence. This structure allows for efficient memory usage and dynamic re-sizing, as nodes can be added or removed without reallocating the entire list.

In the context of file allocation, linked lists can be used to represent the blocks of data within a file. Each block of data can be stored in a node of the linked list, with pointers connecting these nodes to form a chain. This design enables files to grow dynamically as more data is added, without requiring a contiguous block of memory. It also facilitates the management of free and used memory, as the operating system can easily traverse the linked list to find available space or to release memory when files are deleted.

The use of linked lists in file allocation not only enhances the flexibility of memory management but also simplifies the process of accessing file data. By following the pointers from one block to the next, the operating system can efficiently read or write data, making it a vital technique in the design of modern file systems. This dynamic allocation is particularly essential in environments where file sizes can vary significantly, allowing for optimal utilization of storage resources.

Explanation of the Provided C Code Structure

The provided C code implements a basic file system using three primary structures: Block, File, and FileSystem. Each structure serves a specific purpose and collectively facilitates the management of files and their contents.

The Block structure is fundamental for storing data within a file. It includes a character array data that can hold up to 256 characters, representing the actual content of the file. Additionally, it contains a pointer next that links to the next block in the sequence. This design allows for the creation of files that can grow dynamically, as new blocks can be added without needing a contiguous memory allocation. The linked nature of blocks ensures that any amount of data can be represented, limited only by the number of blocks created.

The File structure represents an individual file in the file system. It contains a name field to store the file's name and a pointer start_block that points to the first block of data in that file. This pointer is crucial, as it acts as the entry point for accessing the contents of the file. When data is written to a file, new blocks are created and linked together through their next pointers, allowing for seamless traversal when reading the file's contents.

The FileSystem structure acts as a container for managing multiple files. It holds an array of pointers to File structures, enabling the system to keep track of all the files created. The file_count variable maintains a count of the total number of files in the system, ensuring that operations like file creation and display can reference the correct number of files. The FileSystem structure also facilitates file management operations such as checking for existing files, creating new files, writing data, and reading data, thus serving as the backbone for the file management system.

Together, these structures interact to form a simple yet effective file management system that can create, store, and retrieve files and their associated data. This design highlights the importance of linked data structures in managing dynamic memory allocation, particularly in scenarios where the size and number of files can vary significantly.

Functions for Managing the File System

The provided C code includes several key functions that serve to manage the file system effectively. Each function plays a pivotal role in facilitating various operations, ensuring that users can create, modify, and access files seamlessly within the system.

init_file_system

The init_file_system function is responsible for initializing the file system. It sets the file_count attribute of the FileSystem structure to zero, indicating that no files have been created yet. This function is essential as it prepares the file system for subsequent operations, ensuring that the data structure is clean and ready for use.

file_exists

The file_exists function checks whether a file with a specified name already exists in the file system. It iterates through the array of file pointers in the FileSystem structure, comparing each file’s name with the given filename. If a match is found, the function returns 1 (true), indicating that the file exists. If no match is found after checking all files, it returns 0 (false). This function is crucial for maintaining the uniqueness of file names, preventing the accidental overwriting of existing files.

create_file

The create_file function facilitates the creation of a new file. It first checks if a file with the provided name already exists by calling file_exists. If the file exists, it prompts the user to enter a different name. Once a unique name is confirmed, the function allocates memory for a new File structure, initializes its fields, and adds it to the FileSystem. This function is fundamental for file management, allowing users to create files dynamically while ensuring there is no duplication.

write_to_file

The write_to_file function allows users to add data to an existing file. It searches for the file by name within the FileSystem and, once found, creates a new block to store the data. If the file is empty, the new block becomes the first block; otherwise, the function traverses to the last block and appends the new block. This function is essential for modifying file content, enabling users to store data in a structured manner.

read_from_file

The read_from_file function retrieves and displays the data stored in a specified file. It searches for the file within the FileSystem, and if found, it traverses through the linked blocks, printing each block's data sequentially. This function is critical for data retrieval, allowing users to access and view the contents of their files effortlessly.

show_directory

The show_directory function displays the names of all files currently stored in the file system. It iterates through the file array and prints each file name, providing users with a quick overview of the files they have created. This function enhances usability by allowing users to see their file collection at a glance, aiding in file management.

Data Handling and User Interactions

In the provided C code, data handling revolves around user interactions that enable the creation, writing, and reading of files. The user interaction loop is designed to facilitate an intuitive experience, allowing users to manage files effectively based on their inputs. At the core of this loop is a series of functions that interact with the FileSystem, ensuring that operations are conducted smoothly while maintaining data integrity.

When a user initiates the application, they are prompted to specify how many files they wish to create. This input is captured using the scanf function, which reads the number of files from the standard input. Following this, a loop iterates for the specified number of files, allowing the user to enter a unique filename for each file. The create_file function is invoked to handle the creation process. Before a file is created, the program checks for existing files with the same name using the file_exists function. This serves as a safeguard against duplication, prompting the user to suggest a different name if a conflict arises.

Once a file is created, users can add data to it. The program prompts for the number of data blocks to write, and for each block, it captures the user's input using fgets. This input is then processed by the write_to_file function, which dynamically allocates memory for new data blocks and links them to the existing blocks of the file. This method allows for flexible data storage, as users can add multiple blocks of data without worrying about memory constraints.

Reading from files is equally user-friendly. The application iterates through the files in the FileSystem, allowing the user to view the contents of each file sequentially. The read_from_file function traverses the linked blocks, printing the stored data to the console. This interaction not only enhances user engagement but also provides an efficient way to verify the contents of the files created.

Overall, the design of the user interaction loop in this file management system emphasizes robustness and flexibility, allowing users to create, populate, and access files with ease while ensuring that data is handled correctly throughout the process.

Conclusion

The current implementation of the file management system showcases a fundamental approach to file allocation using linked lists in C. This design effectively handles file creation, data writing, and reading while ensuring dynamic memory allocation. One of the primary strengths of this implementation is its flexibility; files can grow dynamically as more data is added, and the linked list structure allows for efficient memory utilization. Additionally, the user-friendly interface supports straightforward interactions, making it accessible for users to manage their files.

However, the system does have limitations. For instance, error handling is minimal and could be significantly improved. Currently, the program relies on basic checks for file existence and allocation failures, but it does not adequately handle memory allocation errors, which could lead to crashes or undefined behavior if the system runs out of memory. Implementing more robust error handling would enhance the reliability of the system, providing users with clearer feedback in case of issues.

Moreover, memory management could be optimized further. Although the linked list structure is suitable for dynamic memory allocation, it lacks mechanisms for freeing unused memory blocks when files are deleted or when data blocks are no longer needed. This could lead to memory leaks over time, especially in long-running applications. Incorporating a function to deallocate memory for blocks and files would improve memory efficiency.

Expanding system capabilities is another avenue for improvement. For example, adding support for file metadata, such as timestamps or file sizes, could provide users with more context about their files. Furthermore, implementing additional features such as file searching, sorting, and support for different file types would enhance the functionality of the system, allowing for more complex file management tasks.

Incorporating these potential improvements would not only enhance the robustness and efficiency of the file management system but also enrich the user experience, making it a more versatile tool for managing files in various applications.

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Define a Block structure that holds data and points to the next block

typedef struct Block {

    char data[256];            // Data stored in the block (limit of 256 characters)

    struct Block* next;        // Pointer to the next block (linked list)

} Block;


// Define a File structure that has a name and points to the first block

typedef struct File {

    char name[50];             // File name

    Block* start_block;        // Pointer to the first block of the file

} File;
```

```
// Define a FileSystem structure to hold the list of files
typedef struct FileSystem {
    File* files[100];         // Array of pointers to File structures (directory)
    int file_count;           // Counter to track the number of files
} FileSystem;

// Function to initialize a FileSystem
void init_file_system(FileSystem* fs) {
    fs->file_count = 0;
}

// Function to check if a file with the given name already exists
int file_exists(FileSystem* fs, const char* filename) {
    for (int i = 0; i < fs->file_count; i++) {
        if (strcmp(fs->files[i]->name, filename) == 0) {
            return 1;  // File with the same name exists
        }
    }
    return 0;  // File does not exist
}

// Function to create a new file with unique name
void create_file(FileSystem* fs, char* filename) {
    while (file_exists(fs, filename)) {
        printf("Error: File '%s' already exists.\nPlease choose a different name: ",
filename);
        fgets(filename, 50, stdin);
        filename[strcspn(filename, "\n")] = '\0';  // Remove newline character from
filename
```

```
    }
    if (fs->file_count >= 100) {
        printf("File system full! Cannot create more files.\n");
        return;
    }

    // Create a new file
    File* new_file = (File*)malloc(sizeof(File));
    strcpy(new_file->name, filename);
    new_file->start_block = NULL;  // No data in the file initially

    fs->files[fs->file_count] = new_file;
    fs->file_count++;
    printf("File '%s' created successfully.\n", filename);
}

// Function to add data to a file
void write_to_file(FileSystem* fs, const char* filename, const char* data) {
    for (int i = 0; i < fs->file_count; i++) {
        File* file = fs->files[i];

        if (strcmp(file->name, filename) == 0) {
            // Create a new block to store the data
            Block* new_block = (Block*)malloc(sizeof(Block));
            strcpy(new_block->data, data);
            new_block->next = NULL;  // Last block points to NULL

            if (file->start_block == NULL) {
                file->start_block = new_block;  // First block of the file
            } else {
                // Traverse to the last block and append the new block
                Block* current = file->start_block;
                while (current->next != NULL) {
                    current = current->next;
                }
                current->next = new_block;
            }
            printf("Data written to file '%s'.\n", filename);
            return;
        }
    }
    printf("File '%s' not found.\n", filename);
}

// Function to read data from a file
void read_from_file(FileSystem* fs, const char* filename) {
    for (int i = 0; i < fs->file_count; i++) {
        File* file = fs->files[i];
        if (strcmp(file->name, filename) == 0) {
            Block* current = file->start_block;
            printf("Data from file '%s':\n", filename);
            while (current != NULL) {
                printf("%s", current->data);
                current = current->next;
            }
            printf("\n");
            return;
        }
    }
    printf("File '%s' not found.\n", filename);
}

// Function to display all files in the directory
void show_directory(FileSystem* fs) {
    printf("Directory contents:\n");
    for (int i = 0; i < fs->file_count; i++) {
        printf("- %s\n", fs->files[i]->name);
    }
}

int main() {
    FileSystem fs;
    init_file_system(&fs);

    int num_files;
    char filename[50];
    char data[256];

    // Ask the user how many files they want to create
    printf("Enter the number of files you want to create: ");
    scanf("%d", &num_files);
    getchar();  // Consume newline character left by scanf

    // Create files and add data to them
```

```
   for (int i = 0; i < num_files; i++) {
       // Input file name
       printf("\n");
       printf("\nEnter the name of file %d: ", i + 1);
       fgets(filename, sizeof(filename), stdin);
       filename[strcspn(filename, "\n")] = '\0';  // Remove newline character from
filename

       // Create the file
       create_file(&fs, filename);

       // Ask for the number of blocks of data to write
       int num_blocks;
       printf("Enter the number of blocks of data you want to write to '%s': ",
filename);
       scanf("%d", &num_blocks);
       getchar();  // Consume newline character left by scanf

       // Input data for each block and write to file
       for (int j = 0; j < num_blocks; j++) {
           printf("Enter data for block %d of '%s': ", j + 1, filename);
           fgets(data, sizeof(data), stdin);
           data[strcspn(data, "\n")] = '\0';  // Remove newline character from data

           write_to_file(&fs, filename, data);
       }
   }

   // Show the directory contents
```

```
    printf("\n");
    show_directory(&fs);

    // Read data from files
    printf("\n");
    for (int i = 0; i < fs.file_count; i++) {
        read_from_file(&fs, fs.files[i]->name);
        printf("\n");
    }
    return 0;
}
```

Synchronization in C: Producer-Consumer Problem

Overview
This C program demonstrates the Producer-Consumer problem using synchronization primitives like mutex locks and condition variables. The producer adds items to a shared buffer, while the consumer removes items from it. Synchronization ensures that the producer does not add items when the buffer is full, and the consumer does not remove items when the buffer is empty.

Producer-Consumer Problem
The Producer-Consumer problem is a classical synchronization problem where two threads, producer and consumer, share a common buffer. Proper synchronization is necessary to avoid scenarios like:
1. Buffer overflow: The producer adds items to an already full buffer.
2. Buffer underflow: The consumer tries to remove items from an empty buffer.

Code Explanation
The program uses a fixed-size buffer, mutex locks, and condition variables to synchronize the producer and consumer threads. Key elements of the program include:

1. Buffer and Count:
   - A shared array ‘buffer’ stores the items.
   - \`count\` keeps track of the number of items in the buffer.

2. Mutex Lock:
   - ‘pthread_mutex_t’ ensures mutual exclusion, preventing simultaneous access to the shared buffer.

3. Condition Variables:
   - ‘pthread_cond_t cond_produce’ signals when the producer can add items.
   - ‘pthread_cond_t cond_consume’ signals when the consumer can remove items.

Producer Thread
The producer generates items and adds them to the buffer. It waits when the buffer is full and resumes adding items when the consumer removes items.
Steps:
1. Lock the mutex to ensure exclusive access to the buffer.
2. Wait on ‘cond_produce’ if the buffer is full.
3. Add an item to the buffer and increment the count.
4. Signal ‘cond_consume’ to notify the consumer.
5. Unlock the mutex.

Consumer Thread

The consumer removes items from the buffer. It waits when the buffer is empty and resumes consumption when the producer adds items.
Steps:
1. Lock the mutex to ensure exclusive access to the buffer.
2. Wait on ‘cond_consume’ if the buffer is empty.
3. Remove an item from the buffer and decrement the count.
4. Signal ‘cond_produce’ to notify the producer.
5. Unlock the mutex.

Advantages and Limitations

Advantages:
1. Prevents race conditions using mutex locks.
2. Condition variables provide efficient thread communication.
3. Implements a real-world synchronization scenario.

Limitations:
1. Hardcoded buffer size limits flexibility.
2. Lack of error handling for thread creation and joining.
3. Fixed sleep time simulates delays but isn't precise.

Conclusion

This program effectively illustrates synchronization concepts in multithreading. Proper use of mutex locks and condition variables ensures safe interaction between threads, preventing data corruption and ensuring logical flow. This implementation is a foundation for understanding more complex synchronization mechanisms.

Detailed Analysis

Critical Section Problem:
The Producer-Consumer problem is a classic example of the critical section problem, where multiple threads need access to shared resources (in this case, the buffer) without interfering with each other. The use of mutex locks ensures that only one thread can modify the buffer at a time, thus preventing race conditions.

Mutex Locks and Their Role:
Mutex locks are essential for ensuring mutual exclusion. In this program, ‘pthread_mutex_lock’ and ‘pthread_mutex_unlock’ are used to lock and unlock the critical section. This prevents simultaneous access to the buffer by the producer and consumer threads, ensuring data integrity.

Condition Variables and Thread Coordination:
Condition variables provide a mechanism for threads to signal each other when a particular condition is met. For example, the producer signals the consumer using ‘pthread_cond_signal’ when an item is added to the buffer. Conversely, the

consumer signals the producer when an item is removed, allowing the producer to add more items.

Sample Output

The following is an example of the program's output:

```
Producer: produced item 1
Consumer: consumed item 1
Producer: produced item 2
Producer: produced item 3
Consumer: consumed item 2
Consumer: consumed item 3
...
```

Real-World Applications

Applications of the Producer-Consumer Model:
1. Operating Systems: Managing processes that produce and consume data, such as input-output operations.
2. Networking: Data packets being sent (producer) and received (consumer) over a network.
3. Database Systems: Query execution where one thread fetches data, and another processes it.
4. Multimedia Systems: Streaming services where video/audio data is buffered and consumed.

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>



#define BUFFER_SIZE 5



int buffer[BUFFER_SIZE];

int count = 0;



pthread_mutex_t mutex;

pthread_cond_t cond_produce;
pthread_cond_t cond_consume;


// Function for the producer thread
void* producer(void* arg) {
   int item = 0;
   while (item != 6) {
      item++;

      pthread_mutex_lock(&mutex);

      // Wait if the buffer is full
      while (count == BUFFER_SIZE) {
         pthread_cond_wait(&cond_produce, &mutex);
      }

      // Produce an item
      buffer[count] = item;
      count++;
      printf("Producer: produced item %d\n", item);

      // Signal the consumer that an item is available
      pthread_cond_signal(&cond_consume);

      pthread_mutex_unlock(&mutex);

      sleep(1); // Simulate production time
   }
```

```
    return NULL;
}

// Function for the consumer thread
void* consumer(void* arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        // Wait if the buffer is empty
        while (count == 0) {
            pthread_cond_wait(&cond_consume, &mutex);
        }

        // Consume an item
        int item = buffer[count - 1];
        count--;
        printf("Consumer: consumed item %d\n", item);

        // Signal the producer that space is available
        pthread_cond_signal(&cond_produce);

        pthread_mutex_unlock(&mutex);

        sleep(1); // Simulate consumption time
        if(item == 6)
            break;
    }
    return NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initialize mutex and condition variables
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_produce, NULL);
    pthread_cond_init(&cond_consume, NULL);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for the threads to finish (they won't in this example)
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Clean up
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_produce);
    pthread_cond_destroy(&cond_consume);

    return 0;
}
```

Understanding CPU Scheduling Algorithms

Introduction

CPU scheduling is a critical function in operating systems that determines the order in which processes are executed by the CPU. Efficient scheduling ensures that CPU resources are utilized effectively, leading to improved system performance and responsiveness. The importance of CPU scheduling cannot be overstated, as it directly affects various performance metrics such as waiting time, turnaround time, and overall system throughput.

As systems become more complex and the demand for multitasking increases, the need for robust scheduling algorithms becomes evident. Different scheduling techniques are designed to meet varying requirements based on the specific workload and user expectations. Analyzing these algorithms allows system designers and users to understand their strengths and weaknesses, enabling them to make informed decisions about which approach to implement in their environments.

This document focuses on three popular CPU scheduling algorithms: First-Come-First-Serve (FCFS), Shortest Job First (SJF), and Round Robin (RR). Each of these algorithms has distinct characteristics and operational methods that influence how well they perform under different conditions. By examining these algorithms, we aim to provide insight into their functionality and effectiveness, ultimately aiding in the selection of the most appropriate scheduling method for various scenarios.

Overview of Scheduling Algorithms

CPU scheduling algorithms are fundamental components of operating systems that manage the execution order of processes, ensuring efficient use of CPU resources. These algorithms play a vital role in process management, influencing system responsiveness and overall performance. By determining which processes to execute and in what order, scheduling algorithms significantly affect key performance metrics, including waiting time, turnaround time, and throughput.

Among the various scheduling algorithms, three of the most widely recognized are First-Come-First-Serve (FCFS), Shortest Job First (SJF), and Round Robin (RR). Each algorithm has unique features that make it suitable for different types of workloads and user demands.

First-Come-First-Serve (FCFS) is the simplest scheduling algorithm, where processes are executed in the order they arrive in the ready queue. While easy to implement, FCFS can lead to inefficiencies, particularly the "convoy effect", where shorter processes are delayed by longer ones. This makes FCFS more appropriate for batch processing systems where simplicity is prioritized over responsiveness.

Shortest Job First (SJF) focuses on minimizing waiting time by selecting the process with the smallest burst time for execution next. This algorithm can significantly reduce average waiting time compared to FCFS. However, it requires accurate knowledge of burst times, which can lead to the problem of starvation for longer processes. SJF is commonly used in environments where job lengths can be predicted reliably.

Round Robin (RR) introduces a preemptive approach, allowing processes to share CPU time in a cyclical manner. Each process is allocated a fixed time slice, or quantum, ensuring that all processes receive a fair share of CPU time. This algorithm is particularly well-suited for time-sharing systems, where user interaction and responsiveness are critical. While RR promotes fairness and better response times for short processes, it can incur higher overhead due to frequent context switching, especially with smaller quantum values.

Understanding these scheduling algorithms allows system designers to select the most appropriate method for their specific requirements, balancing efficiency, fairness, and responsiveness in process management.

Detailed Algorithm Descriptions

First-Come-First-Serve (FCFS)

Definition:
First-Come-First-Serve (FCFS) is the simplest CPU scheduling algorithm that processes jobs in the order they arrive in the ready queue.

Characteristics:

- Non-preemptive: Once a process starts executing, it runs to completion.
- Simple and straightforward to implement.
- Can lead to the "convoy effect", where short processes are delayed by long ones.

Advantages:

- Easy to understand and implement, making it suitable for batch systems.
- Predictable, as processes are executed in the order they arrive.

Disadvantages:

- Poor CPU utilization, especially if a long process arrives first.
- High average waiting time in scenarios where long processes block shorter ones.

Implementation in C Code:
In the provided C code, the processes are sorted by arrival time. The completion time, turnaround time, and waiting time for each process are calculated sequentially after sorting, ensuring that each process runs until completion.

Shortest Job First (SJF)

Definition:
Shortest Job First (SJF) schedules the process with the smallest burst time next, aiming to minimize waiting time.

Characteristics:

- Can be implemented in both preemptive and non-preemptive forms. The provided implementation is non-preemptive.
- Generally reduces average waiting time compared to FCFS.

Advantages:

- Efficient in minimizing the average turnaround and waiting times, making it suitable for environments with predictable job lengths.

Disadvantages:

- Possibility of starvation, where longer processes may wait indefinitely if shorter processes continually arrive.
- Requires accurate knowledge of burst times, which may not be available or easy to predict in real-world scenarios.

Implementation in C Code:
The sjf function sorts processes first by arrival time and then by burst time. It calculates performance metrics after scheduling, ensuring that the shortest jobs are prioritized.

Round Robin (RR)

Definition:
Round Robin (RR) allocates a fixed time quantum to each process in a circular manner, allowing for preemptive scheduling.

Characteristics:

- Preemptive: If a process does not complete in its allocated time quantum, it is placed back in the queue.
- Promotes fairness by ensuring all processes receive CPU time.

Advantages:

- Fair to all processes, preventing starvation and ensuring a responsive system.

- Particularly suitable for interactive systems where user response time is crucial.

Disadvantages:

- Increased overhead due to frequent context switching, especially with smaller quantum values.
- Average waiting time can increase with high variability in burst times, as shorter tasks may wait longer if many longer tasks are present.

Implementation in C Code:
The roundRobin function employs a queue to manage process execution. Each process executes for the specified quantum or until it completes. If time remains, the process is re-added to the queue, maintaining a fair execution order.

By understanding these scheduling algorithms in detail, we can appreciate their operational principles and the implications of their implementation in various computing environments.

Code Implementation and Key Functions

The C program implementing the CPU scheduling algorithms is structured to encapsulate the logic of each algorithm within dedicated functions. This modular design enhances clarity and allows for easy debugging and maintenance. The major components of the code are the functions for each scheduling algorithm—fcfs, sjf, and roundRobin—alongside utility functions like calculateMetrics and printMetrics to process and display the results.

Key Functions Overview

1. **fcfs Function**:
   This function handles the First-Come-First-Serve scheduling algorithm. It starts by sorting the list of processes based on their arrival times, ensuring that the scheduling happens in the correct order. After sorting, the function calculates the completion time for each process sequentially. From the completion time, it derives the turnaround time and waiting time using the predefined formulas. The simplicity of this function reflects the straightforward nature of the FCFS algorithm, making it easy to follow.

2. **sjf Function**:
   Similar to the fcfs function, the sjf function begins by sorting the processes, but in this case, it sorts by both arrival time and burst time. Once sorted, it calculates the performance metrics, including the completion, turnaround, and waiting times. The focus on minimizing waiting time is evident, as this function prioritizes shorter processes, thereby optimizing overall performance.

3. **roundRobin Function**:
   The roundRobin function implements the Round Robin scheduling

technique using a queue to manage process execution. Each process gets executed for a specified time quantum. If a process does not finish within its time slice, it is placed back in the queue for another turn. This approach ensures fairness among processes, allowing for a balanced distribution of CPU time. The function carefully manages the queue to keep track of processes still needing execution.

4. **calculateMetrics Function**:
   This utility function computes essential performance metrics such as completion time, turnaround time, and waiting time for the scheduled processes. It takes into account the results generated by the scheduling algorithms and uses them to provide meaningful insights into the performance of the system under different scheduling strategies.

5. **printMetrics Function**:
   Once the metrics are calculated, the printMetrics function is responsible for displaying the detailed information about each process, including average waiting time, turnaround time, and throughput. This function formats and presents the data in a user-friendly manner, allowing users to easily interpret the outcomes of their selected scheduling algorithm.

Each of these functions plays a crucial role in the overall functioning of the program, effectively managing the scheduling of incoming tasks and calculating the corresponding performance metrics. The careful design and implementation ensure that users can easily select and visualize the impact of different CPU scheduling strategies.

Performance Metrics Explanation

When evaluating CPU scheduling algorithms, three primary performance metrics are utilized: Waiting Time (WT), Turnaround Time (TAT), and Throughput. Each of these metrics provides critical insights into the efficiency and effectiveness of a scheduling strategy, influencing system performance significantly.

Waiting Time (WT)

Definition:
Waiting Time is the total time a process spends waiting in the ready queue before it gets executed by the CPU. It is calculated using the formula:

WT = Turnaround Time - Burst Time

Significance:
A lower waiting time indicates that processes are being executed promptly, which improves system responsiveness. High waiting times can lead to user dissatisfaction, especially in interactive systems. In scheduling algorithms like FCFS, processes can experience increased waiting times due to the "convoy effect", where short processes wait for longer ones to complete.

Turnaround Time (TAT)

Definition:
Turnaround Time is the total time taken from the arrival of a process to its completion. It is computed as follows:

TAT = Completion Time - Arrival Time

Significance:
Turnaround time encompasses the entire lifecycle of a process and reflects the efficiency of the scheduling algorithm. A minimal turnaround time is desirable, as it indicates that processes are being completed quickly. Algorithms like Shortest Job First (SJF) aim to reduce turnaround time by prioritizing processes with shorter burst times, thus optimizing overall system throughput.

Throughput

Definition:
Throughput refers to the number of processes completed per unit of time. It can be calculated using the formula:

Throughput = Total Processes / Total Completion Time

Significance:
Throughput is a vital indicator of system performance, as it measures the efficiency of the CPU in handling processes. Higher throughput indicates that the system can handle a larger workload efficiently. Round Robin (RR) scheduling, while ensuring fairness, may experience lower throughput if the time quantum is not optimally set, leading to excessive context switching.

Implications on System Performance

Each of these metrics—waiting time, turnaround time, and throughput — interplays with one another, influencing the overall performance of the CPU scheduling algorithms. A balance among these metrics is essential for designing responsive, efficient systems. For instance, while a scheduling algorithm may minimize waiting time, it could adversely affect throughput or turnaround time. Therefore, understanding these performance metrics is crucial for selecting the most appropriate scheduling strategy based on the specific requirements of the system and its workload.

Sample Execution Flow

The execution flow of the CPU scheduling program involves several steps that guide the user from inputting process information to interpreting the output metrics. Below is an illustration of a typical run of the program, including sample inputs and the expected outputs.

Step 1: Input Process Information

When the program starts, it prompts the user to enter the number of processes. For example, the user inputs 3 to indicate that three processes will be scheduled. Next, the program requests the arrival and burst times for each process. The user might enter the following details:

- Process 1: Arrival Time = 0, Burst Time = 5
- Process 2: Arrival Time = 1, Burst Time = 3
- Process 3: Arrival Time = 2, Burst Time = 8

Step 2: Select Scheduling Algorithm

After inputting the process details, the program presents a menu of scheduling algorithms. The user is prompted to choose from the following options:

1. First-Come-First-Serve (FCFS)
2. Shortest Job First (SJF)
3. Round Robin (RR)

For this example, let's assume the user selects 2 for Shortest Job First (SJF).

Step 3: Specify Quantum (if applicable)

If the user had selected Round Robin, the program would then ask for a time quantum. For instance, if the user input 4, the program would use this value in its scheduling calculations. However, since SJF was selected in this case, this step is skipped.

Step 4: Calculate and Display Output Metrics

Once the scheduling algorithm is selected, the program processes the input data according to SJF. It sorts the processes by burst time and calculates performance metrics. The output might look like this:

Process ID	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	5	5	0
2	1	3	8	7	4
3	2	8	16	14	6

Average Waiting Time: 3.33
Average Turnaround Time: 8.67
Throughput: 0.18

Step 5: Interpret the Output

In the output, the user can see the detailed performance metrics for each process, including completion time, turnaround time, and waiting time. The averages of waiting time and turnaround time show how efficiently the selected algorithm has performed. The throughput value indicates the efficiency of the CPU in processing the provided jobs.

This step-by-step flow effectively illustrates how the CPU scheduling program operates, providing users with critical insights into the performance of their selected scheduling algorithm based on the entered process details.

Comparative Analysis

The comparative analysis of CPU scheduling algorithms—First-Come-First-Serve (FCFS), Shortest Job First (SJF), and Round Robin (RR)—highlights their respective strengths and weaknesses based on average waiting time, average turnaround time, and throughput. These performance metrics are essential for assessing the efficiency and effectiveness of each algorithm in various computing scenarios.

Algorithm	Average Waiting Time (WT)	Average Turnaround Time (TAT)	Throughput	Remarks
FCFS	Moderate	Moderate	Low	Simple but can lead to high waiting times due to the convoy effect.
SJF	Low	Low	Moderate	Efficient for predictable workloads but may cause starvation for longer processes.
Round Robin	High (if quantum is small)	High	High	Fair to all processes, but context switching can reduce efficiency with small quanta.

FCFS (First-Come-First-Serve)

FCFS is the simplest scheduling algorithm, processing jobs in the order they arrive. It is easy to implement and understand, making it suitable for batch processing systems. However, FCFS can lead to significant waiting times, especially when shorter tasks are queued behind longer ones, resulting in the "convoy effect." This makes FCFS less effective in environments requiring quick responsiveness, such as interactive systems.

SJF (Shortest Job First)

SJF minimizes waiting time by prioritizing processes with the shortest burst times. This algorithm generally results in lower average waiting and turnaround times compared to FCFS, making it efficient for environments with predictable job lengths. However, SJF can lead to starvation for longer processes if shorter tasks continuously arrive. It is preferable in scenarios where burst times can be accurately predicted, such as processing similar tasks in a controlled environment.

RR (Round Robin)

Round Robin scheduling allocates a fixed time slice to each process, ensuring fairness and responsiveness. This makes it particularly suitable for time-sharing systems, where user interaction is critical. While RR promotes fairness, it can incur higher average waiting times if the time quantum is too small, leading to frequent context switches. In scenarios with high variability in burst times, RR may be less efficient. It is best suited for interactive systems where response time is more important than raw throughput.

In conclusion, the choice of scheduling algorithm should be based on the specific requirements of the system. Factors such as the nature of the workload, the need for responsiveness, and the potential for starvation must be considered when selecting the most appropriate scheduling method.

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX_PROCESSES 100



// Structure to represent a process

typedef struct {
    int pid;        // Process ID
    int arrival;    // Arrival time
    int burst;      // Burst time
    int waiting;    // Waiting time
    int turnaround; // Turnaround time
    int completion; // Completion time
    int remaining;  // Remaining burst time for Round Robin
    int processed;  // Flag to check if the process has been added to the queue (for
Round Robin)
} Process;


// Function prototypes
void fcfs(Process processes[], int n);
void sjf(Process processes[], int n);
void roundRobin(Process processes[], int n, int quantum);
void calculateMetrics(Process processes[], int n);
void printMetrics(Process processes[], int n, const char *algorithm);


int main() {
    system("cls");
    int n, choice, quantum;
    int i;
    Process processes[MAX_PROCESSES];


    printf("Enter the number of processes: ");
    scanf("%d", &n);


    for (i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time for process P%d: ", processes[i].pid);
        scanf("%d", &processes[i].arrival);
        printf("Enter burst time for process P%d: ", processes[i].pid);
        scanf("%d", &processes[i].burst);
        processes[i].remaining = processes[i].burst; // Set remaining burst time
        processes[i].processed = 0; // Initialize the processed flag
    }

    while (1) {
        printf("\nCPU Scheduling Algorithms:\n");
        printf("1. First-Come-First-Serve (FCFS)\n");
        printf("2. Shortest Job First (SJF)\n");
        printf("3. Round Robin (RR)\n");
        printf("4. Exit\n");
        printf("Choose an option: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                fcfs(processes, n);
                break;
            case 2:
                sjf(processes, n);
                break;
            case 3:
                printf("Enter time quantum for Round Robin: ");
                scanf("%d", &quantum);
                roundRobin(processes, n, quantum);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}

// First-Come-First-Serve Scheduling
void fcfs(Process processes[], int n) {
    Process temp[MAX_PROCESSES];
    int i, j;
    for (i = 0; i < n; i++) {
        temp[i] = processes[i];
    }

    // Sort by arrival time
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (temp[j].arrival > temp[j + 1].arrival) {
                Process t = temp[j];
                temp[j] = temp[j + 1];
                temp[j + 1] = t;
            }
        }
    }

    calculateMetrics(temp, n);
    printMetrics(temp, n, "First-Come-First-Serve (FCFS)");
}

// Shortest Job First Scheduling
void sjf(Process processes[], int n) {
    Process temp[MAX_PROCESSES];
    int i, j;
    for (i = 0; i < n; i++) {
        temp[i] = processes[i];
    }

    // Sort by arrival time, then by burst time
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (temp[j].arrival > temp[j + 1].arrival ||
                (temp[j].arrival == temp[j + 1].arrival && temp[j].burst > temp[j +
1].burst)) {
                Process t = temp[j];
                temp[j] = temp[j + 1];
                temp[j + 1] = t;
            }
        }
    }

    calculateMetrics(temp, n);
    printMetrics(temp, n, "Shortest Job First (SJF)");
```

```
}

// Round Robin Scheduling
void roundRobin(Process processes[], int n, int quantum) {
    Process temp[MAX_PROCESSES];
    int i;
    for (i = 0; i < n; i++) {
        temp[i] = processes[i];
        temp[i].remaining = processes[i].burst; // Reset remaining burst time for RR
        temp[i].processed = 0; // Reset the processed flag
    }

    int currentTime = 0;
    int remainingProcesses = n;
    int queue[MAX_PROCESSES];
    int front = 0, rear = 0;

    // Add processes to the queue that are ready to execute
    for (i = 0; i < n; i++) {
        if (temp[i].arrival <= currentTime) {
            queue[rear++] = i;
            temp[i].processed = 1; // Mark as processed
        }
    }

    // Process queue in round robin fashion
    while (remainingProcesses > 0) {
        int idx = queue[front++];
        if (temp[idx].remaining > quantum) {
            temp[idx].remaining -= quantum;
            currentTime += quantum;
        } else {
            currentTime += temp[idx].remaining;
            temp[idx].remaining = 0;
            remainingProcesses--;
        }

        // Check if any new process has arrived while processing
        for (i = 0; i < n; i++) {
            if (temp[i].arrival <= currentTime && !temp[i].processed) {
                queue[rear++] = i;
                temp[i].processed = 1; // Mark as processed
            }
        }

        // Add back the process to queue if it still has remaining burst time
        if (temp[idx].remaining > 0) {
            queue[rear++] = idx;
        }
    }

    // Calculate metrics after Round Robin scheduling
    calculateMetrics(temp, n);
    printMetrics(temp, n, "Round Robin (RR)");
}
```

```
// Calculate metrics: waiting time, turnaround time, and completion time
void calculateMetrics(Process processes[], int n) {
    int currentTime = 0;
    int i;

    for (i = 0; i < n; i++) {
        if (currentTime < processes[i].arrival) {
            currentTime = processes[i].arrival;
        }
        processes[i].completion = currentTime + processes[i].burst;
        processes[i].turnaround = processes[i].completion - processes[i].arrival;
        processes[i].waiting = processes[i].turnaround - processes[i].burst;
        currentTime = processes[i].completion;
    }
}

// Print metrics and performance
void printMetrics(Process processes[], int n, const char *algorithm) {
    int totalWaiting = 0, totalTurnaround = 0;
    double throughput = (double)n / processes[n - 1].completion;
    int i;

    printf("\n%s:\n", algorithm);
    printf("\nP\tArrival\t\tBurst\t\tCompletion\tTurnaround\tWaiting\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
               processes[i].pid,
               processes[i].arrival,
            processes[i].burst,
            processes[i].completion,
            processes[i].turnaround,
            processes[i].waiting);

        totalWaiting += processes[i].waiting;
        totalTurnaround += processes[i].turnaround;
    }

    printf("\nAverage Waiting Time: %.2f\n", (double)totalWaiting / n);
    printf("Average Turnaround Time: %.2f\n", (double)totalTurnaround / n);
    printf("Throughput: %.2f processes/unit time\n", throughput);
}
```