Kathmandu University

Dhulikhel, Kavre

Lab Report

COMP-307

Submitted by:

Bibhushan Saakha

Roll no: 41

Computer Engineering

3rd Year/ 1st Semester

Submitted to :

Mr. Sameer Tamrakar

Department of Computer Science and Engineering

CPU Scheduling Algorithms

CPU Scheduling Algorithms are methods employed by operating systems to manage the execution order of processes in a computer system. These algorithms determine which process gets access to the CPU at a given time, aiming to optimize resource utilization and system performance.

In this report, we will look into three CPU scheduling algorithms:

1. First-Come, First-Served (FCFS)
2. Shortest Job First (SJF)
3. Banker's Algorithm

These algorithms have been implemented using C++. These algorithms govern the order and allocation of processes, impacting system efficiency. Our exploration aims to unveil the practical application of these algorithms, emphasizing their execution using the C++ programming language. The report aims to shed light on how these algorithms contribute to effective resource management in computer systems.

1. First-Come-First-Serve (FCFS):

First-Come-First-Serve (FCFS) is a CPU scheduling algorithm that arranges processes in the order of their arrival. In this system, the first process to arrive is the first to be executed. While it is easy to understand and implement, FCFS may lead to inefficiencies, particularly in scenarios where short processes arrive later, resulting in longer waiting times and potentially compromising overall system performance.

Each function/ important parts of the implementation of FCFS program in C++ are explained below:

1.1 Process Structure:

The Process structure includes details such as an ID, how long the task takes, and when it arrives. This structure helps organize and store information about each task. In the main part of the program, we use this structure to create a list of tasks, making things neat and tidy.

```cpp
#include <iostream>
using namespace std;

// Define a structure to represent a process with its
attributes
struct Process
{
    int processId;      // Unique identifier for the process
    int burstTime;      // Time required by the process for
execution
    int arrivalTime;    // Time at which the process arrives in
the system
};
```

1.2 CalculateTimes Function:

The calculateTimes function handles processes and their times. It goes through each process, figures out how long they wait, when they finish, and how much time they take altogether. After doing the math, it shows a table with all this info for each process.

```
// Function to calculate completion time, turnaround time, and
waiting time for each process
void calculateTimes(Process processes[], int n, int wt[], int
ct[], int tat[])
{
    // Calculate values for the first process
    wt[0] = 0;
    ct[0] = processes[0].burstTime;
    tat[0] = ct[0] - processes[0].arrivalTime;

    // Calculate values for the rest of the processes
    for (int i = 1; i < n; i++)
    {
        wt[i] = ct[i - 1];
        ct[i] = wt[i] + processes[i].burstTime;
        tat[i] = ct[i] - processes[i].arrivalTime;
    }

    // Display a table with the calculated values
    cout << "\nProcess\t| Arrival Time\t| Burst Time\t|
Completion Time\t| Turnaround Time\t| Waiting Time\n";
    for (int i = 0; i < n; i++)
        cout << processes[i].processId << "\t| " <<
processes[i].arrivalTime << "\t\t| " << processes[i].burstTime
<< "\t\t| "
             << ct[i] << "\t\t\t| " << tat[i] << "\t\t\t| " <<
wt[i] << endl;
}
```

1.3 main Function:

The calculateTimes function handles processes and their times. It goes through each process, figures out how long they wait, when they finish, and how much time they take altogether. After doing the math, it shows a table with all this info for each process.

```
// Function to calculate completion time, turnaround time, and
waiting time for each process
void calculateTimes(Process processes[], int n, int wt[], int
ct[], int tat[])
{
    // Calculate values for the first process
    wt[0] = 0;
    ct[0] = processes[0].burstTime;
    tat[0] = ct[0] - processes[0].arrivalTime;

    // Calculate values for the rest of the processes
    for (int i = 1; i < n; i++)
    {
        wt[i] = ct[i - 1];
        ct[i] = wt[i] + processes[i].burstTime;
        tat[i] = ct[i] - processes[i].arrivalTime;
    }

    // Display a table with the calculated values
    cout << "\nProcess\t| Arrival Time\t| Burst Time\t|
Completion Time\t| Turnaround Time\t| Waiting Time\n";
    for (int i = 0; i < n; i++)
        cout << processes[i].processId << "\t| " <<
processes[i].arrivalTime << "\t\t| " << processes[i].burstTime
<< "\t\t| "
             << ct[i] << "\t\t\t| " << tat[i] << "\t\t\t| " <<
wt[i] << endl;
}
```

1.4 Output:

The output displays the results of a CPU scheduling simulation for five processes. Each process is identified by a unique ID, and the user inputs their arrival times and burst times.

The table provides a detailed breakdown of each process, including the arrival time, burst time, completion time, turnaround time, and waiting time.

Here, Process 1, with an arrival time of 2 and a burst time of 4, completes at time 4, resulting in a turnaround time of 2 and no waiting time. Similarly, process 2, 3, 4 and 5 has their respective arrival time and burst time input as following table.

The table comprehensively outlines the performance metrics for all processes, offering insights into their execution characteristics under the specified CPU scheduling algorithm.

```
bibhushansaakha@MacBook-Air-520 CPU-Scheduling-Algorithms % ./a.out
Enter Arrival Time and Burst Time for 5 processes:
Process 1 Arrival Time: 2
Process 1 Burst Time: 4
Process 2 Arrival Time: 3
Process 2 Burst Time: 5
Process 3 Arrival Time: 2
Process 3 Burst Time: 3
Process 4 Arrival Time: 5
Process 4 Burst Time: 2
Process 5 Arrival Time: 1
Process 5 Burst Time: 3

Process | Arrival Time  | Burst Time    | Completion Time       | Turnaround Time       | Waiting Time
1       | 2             | 4             | 4                     | 2                     | 0
2       | 3             | 5             | 9                     | 6                     | 4
3       | 2             | 3             | 12                    | 10                    | 9
4       | 5             | 2             | 14                    | 9                     | 12
5       | 1             | 3             | 17                    | 16                    | 14
bibhushansaakha@MacBook-Air-520 CPU-Scheduling-Algorithms %
```

2. Shortest Job First (SJF):

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time for execution next. It prioritizes tasks based on their expected processing time, aiming to minimize the overall completion time. SJF is intuitive, efficient for minimizing waiting times, and can lead to optimal results in certain scenarios. However, predicting the exact burst time poses a challenge, and inaccurate estimations may lead to unexpected interruptions.

Each function/ important parts of the implementation of FCFS program in C++ are explained below:

2.1 Process Structure:

The Process structure includes details such as an ID, how long the task takes, and when it arrives. This structure helps organize and store information about each task. In the main part of the program, we use this structure to create a list of tasks, making things neat and tidy.

```cpp
#include<iostream>
using namespace std;

// Structure to represent a process
struct Process {
    int processId;      // Unique identifier for the process
    int burstTime;      // Time required by the process for
execution
    int arrivalTime;    // Time at which the process arrives in
the system
};
```

2.2 bubbleSort Function:

The bubbleSort function is in charge of arranging processes based on their burst times in ascending order. It uses a simple sorting algorithm called bubble sort, which compares adjacent processes and swaps them if they are not in the correct order. This ensures that processes with shorter burst times come first in the list.

```
// Function to implement the Bubble Sort algorithm for sorting
processes by burst time and arrival time
void bubbleSort(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // Compare arrival times
            if (processes[j].arrivalTime > processes[j +
1].arrivalTime) {
                // Swap processes[j] and processes[j+1] if they
are out of order based on arrival time
                swap(processes[j], processes[j + 1]);
            }
            // If arrival times are equal, compare burst times
            else if (processes[j].arrivalTime == processes[j +
1].arrivalTime &&
                     processes[j].burstTime > processes[j +
1].burstTime) {
                // Swap processes[j] and processes[j+1] if they
are out of order based on burst time
                swap(processes[j], processes[j + 1]);
            }
        }
    }
}
```

2.3 calculateTimes Function:

The calculateTimes function employs the sorted processes obtained from bubbleSort to compute waiting time, completion time, and turnaround time for each process. It starts by updating the current time, considering process arrival times. Then, it calculates and displays the required metrics in a table. Additionally, the function calculates and shows the average burst time for all processes.

```
// Function to calculate waiting time, completion time,
turnaround time, and display the results
void calculateTimes(Process processes[], int n, int wt[], int
ct[], int tat[]) {
    // Sort processes based on burst time using the Bubble Sort
algorithm
    bubbleSort(processes, n);

    int currentTime = 0;
    for (int i = 0; i < n; i++) {
        // Check if the current time is less than the arrival
time of the current process
        if (currentTime < processes[i].arrivalTime) {
            // If so, update the current time to the arrival
time of the current process
            currentTime = processes[i].arrivalTime;
        }

        // Calculate waiting time, completion time, and
turnaround time for the current process
        wt[i] = currentTime - processes[i].arrivalTime;
        currentTime += processes[i].burstTime;
        ct[i] = currentTime;
        tat[i] = ct[i] - processes[i].arrivalTime;
    }
```

```
    // Display a table with the calculated metrics for each
process
    cout << "\nProcess\tArrival Time\tBurst Time\tCompletion
Time\tTurnaround Time\tWaiting Time\n";
    for (int i = 0; i < n; i++)
        cout << processes[i].processId << "\t" <<
processes[i].arrivalTime << "\t\t" << processes[i].burstTime <<
"\t\t"
             << ct[i] << "\t\t" << tat[i] << "\t\t" << wt[i] <<
endl;

    // Calculate and display the average burst time for all
processes
    double averageBurstTime = 0;
    for (int i = 0; i < n; i++)
        averageBurstTime += processes[i].burstTime;

    averageBurstTime /= n;

    cout << "\nAverage Burst Time: " << averageBurstTime <<
endl;
}
```

2.4 main Function:

In the main function, arrays and constants are set up to store process details and times. Users are prompted to input arrival and burst times for each of the five processes. The calculateTimes function is then called to process the input, showcase the computed metrics for each process, and display the average burst time for all processes. The SJF scheduling policy is implemented through the sorting mechanism to prioritize processes with shorter burst times.

```
// Main function where the program starts execution
int main() {
    const int n = 5;
    Process processes[n];
    int waitingTime[n], completionTime[n], turnaroundTime[n];

    // Prompt the user to input arrival and burst times for
each process
    cout << "Enter Arrival Time and Burst Time for 5
processes:\n";
    for (int i = 0; i < n; i++) {
        processes[i].processId = i + 1;
        cout << "Process " << processes[i].processId << "
Arrival Time: ";
        cin >> processes[i].arrivalTime;
        cout << "Process " << processes[i].processId << " Burst
Time: ";
        cin >> processes[i].burstTime;
    }

    // Calculate and display waiting time, completion time,
turnaround time, and average burst time
    calculateTimes(processes, n, waitingTime, completionTime,
turnaroundTime);

    return 0;
}
```

2.4 Output:

The output represents the results of a Shortest Job First (SJF) CPU scheduling algorithm simulation for five processes. Each process is characterized by a distinct ID, and the user inputs their arrival times and burst times.

The table displays a detailed breakdown of each process, including arrival time, burst time, completion time, turnaround time, and waiting time. The processes are ordered based on their burst times, with the shortest job being executed first.

The table provides a comprehensive overview of the execution characteristics under the SJF algorithm, emphasizing the order of execution based on the shortest burst time. Additionally, the average burst time for all processes is calculated and displayed as 3.2.

```
bibhushansaakha@MacBook-Air-520 CPU-Scheduling-Algorithms % ./a.out
Enter Arrival Time and Burst Time for 5 processes:
Process 1 Arrival Time: 1
Process 1 Burst Time: 2
Process 2 Arrival Time: 3
Process 2 Burst Time: 4
Process 3 Arrival Time: 2
Process 3 Burst Time: 4
Process 4 Arrival Time: 3
Process 4 Burst Time: 5
Process 5 Arrival Time: 3
Process 5 Burst Time: 6

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       1               2               3               2               0
3       2               4               7               5               1
2       3               4               11              8               4
4       3               5               16              13              8
5       3               6               22              19              13

Average Burst Time: 4.2
```

3. Banker’s Algorithm:

Banker's Algorithm is a resource allocation and deadlock avoidance method in operating systems. It assesses process resource needs and system availability to predict safe execution. The algorithm aims to prevent deadlocks by granting resources only if safety conditions are met. It efficiently allocates resources while ensuring system safety.

Each function/ important parts of the implementation of Banker’s Algorithm program in C++ are explained below:

3.1 Initialization:

The program begins by setting the maximum number of processes and resource types. It initializes arrays representing available resources, maximum claims by each process, and resources currently allocated to each process. The initial system state is printed using the printState function.

```cpp
#include <iostream>
using namespace std;

const int MAX_PROCESSES = 5; // Maximum number of processes
const int MAX_RESOURCES = 3; // Maximum number of resource
types

// Available resources
int available[MAX_RESOURCES] = {10, 5, 7};

// Maximum claim by each process
int maximum[MAX_PROCESSES][MAX_RESOURCES] = {{7, 5, 3}, {3, 2,
2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};

// Resources currently allocated to each process
int allocation[MAX_PROCESSES][MAX_RESOURCES] = {{0, 1, 0}, {2,
0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};
```

3.2 Print State Function:

The printState function serves to present a detailed overview of the current system state. It starts by displaying the availability of each resource type. Subsequently, it constructs a table detailing each process, showing their respective maximum resource claims and the resources currently allocated to them. This tabular representation aids in understanding the distribution of resources across processes and provides a visual snapshot of the system's resource utilization.

```
// Function to print the current system state
void printState()
{
    cout << "Available Resources: ";
    for (int i = 0; i < MAX_RESOURCES; i++)
    {
        cout << available[i] << " ";
    }
    cout << endl;

    cout << "Process:   Max Resources:\tAlloc Resources:\n";
    for (int i = 0; i < MAX_PROCESSES; i++)
    {
        cout << i << ": \t\t";
        for (int j = 0; j < MAX_RESOURCES; j++)
        {
            cout << maximum[i][j] << " ";
        }
        cout << "\t\t";
        for (int j = 0; j < MAX_RESOURCES; j++)
        {
            cout << allocation[i][j] << " ";
        }
        cout << endl;
    }
}
```

3.3 Safety Check Function (isSafe):

The isSafe function assesses whether the system is in a safe state after a resource request. It checks if the requested resources are available and employs the Banker's algorithm to ensure that granting the request won't lead to an unsafe state. The function returns a boolean indicating whether the system is safe.

```
// Function to check if the system is in a safe state after a
resource request
bool isSafe(int process, int request[MAX_RESOURCES])
{
    // Check if the request is less than or equal to the
available resources
    for (int i = 0; i < MAX_RESOURCES; i++)
    {
        if (request[i] > available[i])
        {
            return false;
        }
    }

    // Assume the allocation and available resources to check
safety
    int tempAvailable[MAX_RESOURCES];
    int tempAllocation[MAX_PROCESSES][MAX_RESOURCES];

    for (int i = 0; i < MAX_RESOURCES; i++)
    {
        tempAvailable[i] = available[i] - request[i];
        tempAllocation[process][i] = allocation[process][i] +
request[i];
    }

    int need[MAX_PROCESSES][MAX_RESOURCES];
    for (int i = 0; i < MAX_PROCESSES; i++)
    {
        for (int j = 0; j < MAX_RESOURCES; j++)
        {
            need[i][j] = maximum[i][j] - tempAllocation[i][j];
        }
    }
```

```
// Check for safety using Banker's algorithm
    bool finish[MAX_PROCESSES] = {false};
    int count = 0;

    while (count < MAX_PROCESSES)
    {
        bool found = false;

        for (int i = 0; i < MAX_PROCESSES; i++)
        {
            if (!finish[i])
            {
                int j;
                for (j = 0; j < MAX_RESOURCES; j++)
                {
                    if (need[i][j] > tempAvailable[j])
                    {
                        break;
                    }
                }

                if (j == MAX_RESOURCES)
                {
                    for (int k = 0; k < MAX_RESOURCES; k++)
                    {
                        tempAvailable[k] += tempAllocation[i]
[k];
                    }

                    finish[i] = true;
                    found = true;
                    count++;
                }
            }
        }

        if (!found)
        {
            break;
        }
    }

    // Check if all processes are finished
    return count == MAX_PROCESSES;
}
```

3.4 Resource Request Processing (requestResources):

The requestResources function processes a resource request for a specific process. It uses the isSafe function to check if granting the request maintains a safe state. If safe, it updates the available and allocated resources accordingly. The function then prints the system state, indicating whether the request was granted or denied.

```
// Function to process a resource request
void requestResources(int process, int request[MAX_RESOURCES])
{
    if (isSafe(process, request))
    {
        // Grant the request
        for (int i = 0; i < MAX_RESOURCES; i++)
        {
            available[i] -= request[i];
            allocation[process][i] += request[i];
        }

        cout << "Request granted. System is in safe state.\n";
        printState();
    }
    else
    {
        cout << "Request denied. Granting the request may lead
to an unsafe state.\n";
    }
}
```

3.5 Main Function::

In the main function, the initial system state is displayed. The user is prompted to input a process number and a resource request. The program then checks the validity of the input and processes the resource request using the requestResources function, updating and displaying the system state accordingly.

```
int main()
{
    int process;
    int request[MAX_RESOURCES];

    // Print the initial state
    cout << "Initial System State:\n";
    printState();

    // Input request
    cout << "\nEnter process number (0 to 4): ";
    cin >> process;

    cout << "Enter resource request (e.g., 0 1 0): ";
    for (int i = 0; i < MAX_RESOURCES; i++)
    {
        cin >> request[i];
    }

    // Check and process the request
    if (process >= 0 && process < MAX_PROCESSES)
    {
        requestResources(process, request);
    }
    else
    {
        cout << "Invalid process number.\n";
    }

    return 0;
}
```

3.6 Output:

The output reflects the results of a Banker's algorithm simulation for resource allocation. The initial system state displays available resources, maximum resources each process can claim, and the current allocation.

The initial system state reveals available resources (10, 5, 7), maximum resources each process can claim, and the current resource allocation. In this example, the user inputs a request for Process 2, specifying the resources as 8, 5, and 9.

The output displays a denial message, explaining that granting the request could result in an unsafe state. This highlights the Banker's algorithm's preventive nature, ensuring that resource allocations maintain a safe and stable system state.