A Project Report On

RESOURCE MANAGEMENT : THE FULL HOST DISPATCHER

Course: Operating System

Submitted by:

Shaik. Ershad Ali - AP22110010941
Harshitha Challa - AP22110010942
Harsha Vardhan Reddy Vanga - AP22110010943
Sai Gopala Krishna Konjeti - AP22110010945
Praveen Kumar - AP22110010950
Rohith Sai Gudibandla - AP22110010954

SRM University-AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh - 522 240

November,2024

CONTENT

I. Project description
II. Abstract
III. Introduction
IV. Objective
V. System Overview
VI. Implementation details
VII. Simulation Workflow
VIII. Code
IX. Output Sample
X. Conclurion
XI. Contribution

PROBLEM:

Resource Management

The Real-Time processes need no i/o resources, so that resource allocation need only be implemented for the Lo-Priority processes.

In addition, you know in advance what resources are going to be required by the processes so they can all be allocated before the process is admitted to the feedback queue(s) and marked as READY to run.

The check, allocation and freeing of resources can be done at the same time as the check, etc for memory allocation, so it makes sense to have analogous function entry points (e.g. **rsrcChk**, **rsrcAlloc**, and **rsrcFree**).

The Full HOST Dispatcher

We should now be ready to put all the parts of the HOST dispatcher together:

1. Initialize dispatcher queues (input queue, real time queue, user job queue, and feedback queues);
2. Initialise memory and resource allocation structures;
3. Fill input queue from dispatch list file;
4. Start dispatcher timer (**dispatcher timer = 0**);
5. While there's anything in any of the queues or there is a currently running process:
    i. Unload pending processes from the input queue:
    While (**head-of-input-queue.arrival-time <= dispatcher timer**)
    dequeue process from input queue and enqueue on either:
        a. Real-time queue or
        b. User job queue;
    ii. Unload pending processes from the user job queue:
    While (**head-of-user-job-queue.mbytes** can be allocated and resources are available)
        a. dequeue process from user job queue,
        b. allocate memory to the process,
        c. allocate i/o resources to process, and
        d. enqueue on appropriate priority feedback queue;
    iii. If a process is currently running:
        a. Decrement process **remaining cpu time**;
        b. If times up:
            A. Send **SIGINT** to the process to terminate it;
            B. Free memory and resources allocated to the process (user processes only);
            C. Free up process structure memory;
        c. else if it is a user process and other processes are waiting in any of the queues:
            A. Send **SIGTSTP** to suspend it;
            B. Reduce the priority of the process (if possible) and enqueue it on the appropriate feedback queue

iv. If no process currently running && real time queue and feedback queue are not all empty:
    a. Dequeue a process from the highest priority queue that is not empty
    b. If already started but suspended, restart it (send **SIGCONT** to it) else start it (**fork** & **exec**)
    c. Set it as currently running process;
v. **sleep** for one second;
vi. Increment dispatcher timer;
vii. Go back to 5.

6. Exit.

Try this with the following example job list:

0, 1, 2, 128, 1, 0, 0, 1
1, 0, 1, 64, 0, 0, 0, 0
1, 1, 1, 128, 0, 0, 0, 0
1, 1, 2, 256, 1, 0, 0, 0
2, 1, 3, 256, 0, 0, 0, 2
3, 1, 2, 770, 1, 0, 0, 0

The order of execution will be:

T = 0 1 2 3 4 5 6 7 8 9 10 11
P0: PFR S S S R |
P1: QR |
P2: PF R |
P3: PF F R S R |
P4: P P P P FR R R |
P5: P P P P P P FR R |

where:

P: in pending queue waiting for resources
Q: queued in Real Time queue - not started
F: queued in feedback queue - not started
R: process running
S: process suspended in feedback queue
|: process terminated

Abstract

This project focuses on simulating the process scheduling and resource management techniques typically used in operating systems. The primary goal is to demonstrate how processes transition between different states, including Pending, Running, Suspended, and Terminated, while managing system resources effectively. The project implements a dispatcher to handle process execution, utilizing a combination of feedback queues and resource allocation methods. By simulating process states, memory allocation, and error handling, the system ensures that real-time and user processes are scheduled efficiently. Additionally, the project accounts for error scenarios, allowing for error detection and correction. This simulation provides valuable insights into how operating systems manage multitasking, resource allocation, and process transitions, laying the groundwork for more advanced research in operating system design and optimization.

Introduction

This project simulates **process scheduling and resource management** in an operating system. It mimics the allocation of memory, CPU time, and resources to processes based on priorities and availability. Using queues for real-time and user processes, the system dynamically schedules tasks, manages resource constraints, and transitions processes between states.

It handles process scheduling based on priority and arrival time, manages process states (Pending, Running, Suspended, Terminated), and uses a dispatcher to control transitions between these states, ensuring optimal execution based on process priorities and available resources.

The program demonstrates fundamental operating system concepts such as process management, resource allocation, and scheduling using a multi-level feedback queue system.

Objective

- **Simulating the Lifecycle of Processes in an Operating System:** The project focuses on replicating the key stages of a process lifecycle, including **pending**, **running**, **suspended**, and **terminated** states. Each process transitions between these states based on its priority, availability of resources, and execution progress. This helps in understanding how an operating system handles process management under dynamic conditions.

- **Implementing Dynamic Resource Allocation and Deallocation:**
  Resources such as memory and hardware are allocated to processes dynamically based on their requirements and availability. The project includes mechanisms to check resource constraints before starting a process and to release these resources back to the system once the process terminates. This ensures efficient utilization of limited resources, reflecting the way real-world operating systems operate.

- **Managing Processes Using Real-Time and Feedback Queue Scheduling:**
  The project employs a priority-based scheduling mechanism where:
  - **Real-time processes** (priority 0) are given immediate attention, ensuring critical tasks are executed without delay.
  - **User processes** are managed through a multi-level feedback queue system, which adapts the priority of processes dynamically. Lower-priority tasks are gradually escalated for fairness, ensuring no process is starved of resources.

- **Visualizing How Operating Systems Prioritize and Handle Multiple Processes:**
  The project provides a step-by-step simulation of how processes are prioritized, scheduled, and executed. At each time step, the states of all processes are displayed, giving a clear visualization of the system's operations. This helps in understanding how operating systems make complex decisions to manage competing demands from multiple processes.

System Overview

The system is based on a dispatcher that:

- Processes input: Reads processes from a file and initializes their attributes.
- Manages queues: Sorts processes into real-time, user job, and feedback queues based on priority.
- Schedules execution: Allocates resources to processes, manages states, and executes tasks while adhering to resource constraints.

Key Components:

1. Processes: Defined by attributes such as priority, memory, and resource requirements.
2. Queues: Used to manage processes efficiently:
    - Input queue for new processes.
    - Real-time queue for priority 0 tasks.
    - User job queue and feedback queues for other processes.
3. Resource Management: Tracks available memory and hardware resources, ensuring processes run without exceeding limits.

Implementation Details

Input Details

The program takes input from a file named dispatch_list.txt. Each line represents a process with the following attributes:

1. **Arrival Time**: When the process enters the system.
2. **Priority**: Task importance (0 for real-time; higher numbers for lower priority).
3. **Memory**: Amount of memory the process requires.
4. **CPU Time**: Execution time required by the process.
5. **Resources**: Additional hardware resources needed by the process.

Input Example

```
void initialize_processes() {
    // Direct initialization of process attributes
    int arrivals[MAX_PROCESSES] = {0, 1, 2, 3, 4, 5};
    int priorities[MAX_PROCESSES] = {0, 1, 1, 2, 2, 3};
    int memories[MAX_PROCESSES] = {100, 200, 150, 300, 250, 400};
    int cpu_times[MAX_PROCESSES] = {5, 3, 6, 2, 4, 7};
    int resources[MAX_PROCESSES] = {2, 3, 1, 2, 1, 3};

    for (int i = 0; i < MAX_PROCESSES; i++) {
        processes[i].id = i;
        processes[i].arrival_time = arrivals[i];
        processes[i].priority = priorities[i];
        processes[i].memory = memories[i];
        processes[i].cpu_time = cpu_times[i];
        processes[i].resources = resources[i];
        processes[i].state = 0; // Pending
        processes[i].remaining_time = processes[i].cpu_time;
        enqueue(&input_queue, &processes[i]);
    }
}
```

Process Attributes

Each process has the following fields, as defined in the Process structure:

- **ID**: Unique identifier.
- **Arrival Time**: Time the process arrives.
- **Priority**: Determines scheduling order.
- **Memory**: Memory required by the process.
- **CPU Time**: Time required to complete.
- **Resources**: Number of hardware resources needed.
- **State**: Current state (Pending, Running, Suspended, Terminated).
- **Remaining Time**: CPU time left for execution.

Queue Management

The system uses multiple queues to organize processes:

1. **Input Queue**: Stores all processes initially.
2. **Real-Time Queue**: Processes with priority 0 are directly added here.
3. **User Job Queue**: Non-real-time processes enter here after resource checks.
4. **Feedback Queues**: Multi-level priority queues for user processes.

Key queue operations:

- **Enqueue**: Adds a process to the queue if space permits.
- **Dequeue**: Removes and returns the next process in the queue.

Simulation Workflow

1. **Initialization**:
    - Input is directly entered in the program itself.
    - Queues (Input, Real-Time, User Job, Feedback Queues) are initialized.
2. **Dispatcher Timer**:
    - At each time step (T), the dispatcher processes tasks based on their priority, resource needs, and current states.
3. **State Transitions**:
    - **Pending**: Processes wait in the input queue until their arrival time.
    - **Running**: The highest-priority process that meets resource constraints starts executing.
    - **Suspended**: Processes are paused if preempted or if resources become unavailable.
    - **Terminated**: Completed processes release memory and resources.
4. **Scheduling**:
    - **Real-Time Processes**: Priority 0 tasks are executed immediately if resources are available.
    - **User Processes**: Managed using feedback queues, starting with the highest priority.
5. **Resource Allocation**:
    - Memory and hardware resources are allocated before a process starts and freed upon termination.
6. **Output Generation**:
    - At each time step, the system logs the state of all processes.

CODE:

```
#include <bits/stdc++.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_PROCESSES 6
#define MAX_MEMORY 1024
#define MAX_RESOURCES 10
#define NUM_FEEDBACK_QUEUES 3

// Process structure
typedef struct {
   int id;
   int arrival_time;
   int priority;
   int memory;
   int cpu_time;
   int resources;
   int state; // 0: pending, 1: running, 2: suspended, 3: terminated
   int remaining_time;
} Process;

// Queue structure
typedef struct {
   Process* processes[MAX_PROCESSES];
   int front, rear, count;
} Queue;

// Global variables
Process processes[MAX_PROCESSES];
Queue input_queue, real_time_queue, user_job_queue,
```

```
feedback_queues[NUM_FEEDBACK_QUEUES];
int available_memory = MAX_MEMORY;
int available_resources = MAX_RESOURCES;
int dispatcher_time = 0;
int running_process_id = -1;

// Function prototypes
void initialize_queue(Queue* queue);
bool is_queue_empty(const Queue* queue);
bool is_queue_full(const Queue* queue);
void enqueue(Queue* queue, Process* process);
Process* dequeue(Queue* queue);
void initialize_processes();
bool can_allocate_resources(int memory, int resources);
void allocate_resources(Process* process);
void release_resources(Process* process);
void manage_dispatch();

void initialize_queue(Queue* queue) {
   queue->front = 0;
   queue->rear = 0;
   queue->count = 0;
}

bool is_queue_empty(const Queue* queue) {
   return queue->count == 0;
}

bool is_queue_full(const Queue* queue) {
   return queue->count == MAX_PROCESSES;
}

void enqueue(Queue* queue, Process* process) {
   if (is_queue_full(queue)) return;
   queue->processes[queue->rear] = process;
```

```
    queue->rear = (queue->rear + 1) % MAX_PROCESSES;
    queue->count++;
}

Process* dequeue(Queue* queue) {
    if (is_queue_empty(queue)) return NULL;
    Process* process = queue->processes[queue->front];
    queue->front = (queue->front + 1) % MAX_PROCESSES;
    queue->count--;
    return process;
}

void initialize_processes() {
    // Direct initialization of process attributes
    int arrivals[MAX_PROCESSES] = {0, 1, 2, 3, 4, 5};
    int priorities[MAX_PROCESSES] = {0, 1, 1, 2, 2, 3};
    int memories[MAX_PROCESSES] = {100, 200, 150, 300, 250, 400};
    int cpu_times[MAX_PROCESSES] = {5, 3, 6, 2, 4, 7};
    int resources[MAX_PROCESSES] = {2, 3, 1, 2, 1, 3};

    for (int i = 0; i < MAX_PROCESSES; i++) {
        processes[i].id = i;
        processes[i].arrival_time = arrivals[i];
        processes[i].priority = priorities[i];
        processes[i].memory = memories[i];
        processes[i].cpu_time = cpu_times[i];
        processes[i].resources = resources[i];
        processes[i].state = 0; // Pending
        processes[i].remaining_time = processes[i].cpu_time;
        enqueue(&input_queue, &processes[i]);
    }
}

bool can_allocate_resources(int memory, int resources) {
    return memory <= available_memory && resources <=
```

```
available_resources;
}

void allocate_resources(Process* process) {
   available_memory -= process->memory;
   available_resources -= process->resources;
}

void release_resources(Process* process) {
   available_memory += process->memory;
   available_resources += process->resources;
}

void manage_dispatch() {
   while (true) {
      bool all_terminated = true;

      // Move pending processes to appropriate queues
      while (!is_queue_empty(&input_queue)) {
         Process* process = input_queue.processes[input_queue.front];
         if (process->arrival_time > dispatcher_time) break;
         dequeue(&input_queue);
         if (process->priority == 0) {
            enqueue(&real_time_queue, process);
         } else {
            enqueue(&user_job_queue, process);
         }
      }

      // Process user job queue for allocation
      while (!is_queue_empty(&user_job_queue)) {
         Process* process =
user_job_queue.processes[user_job_queue.front];
         if (!can_allocate_resources(process->memory, process-
>resources)) break;
```

```
            dequeue(&user_job_queue);
            allocate_resources(process);
            enqueue(&feedback_queues[0], process);
        }

        // Manage the running process
        if (running_process_id != -1) {
            Process* running_process = &processes[running_process_id];
            running_process->remaining_time--;

            if (running_process->remaining_time == 0) {
                running_process->state = 3; // Terminated
                release_resources(running_process);
                running_process_id = -1;
            } else if (running_process->priority > 0 &&
!is_queue_empty(&real_time_queue)) {
                running_process->state = 2; // Suspended
                enqueue(&feedback_queues[running_process->priority - 1],
running_process);
                running_process_id = -1;
            }
        }

        // Assign a new process to run if none is running
        if (running_process_id == -1) {
            Process* next_process = NULL;

            if (!is_queue_empty(&real_time_queue)) {
                next_process = dequeue(&real_time_queue);
            } else {
                for (int i = 0; i < NUM_FEEDBACK_QUEUES; i++) {
                    if (!is_queue_empty(&feedback_queues[i])) {
                        next_process = dequeue(&feedback_queues[i]);
                        break;
                    }
```

```
            }
        }

        if (next_process) {
            next_process->state = 1; // Running
            running_process_id = next_process->id;
        }
    }

    // Check termination status
    for (int i = 0; i < MAX_PROCESSES; i++) {
        if (processes[i].state != 3) {
            all_terminated = false;
            break;
        }
    }

    // Display system state
    printf("Time = %d: ", dispatcher_time);
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d:", i);
        switch (processes[i].state) {
            case 0: printf(" Pending "); break;
            case 1: printf(" Running "); break;
            case 2: printf(" Suspended "); break;
            case 3: printf(" Terminated "); break;
        }
    }
    printf("\n");

    if (all_terminated) break;
    dispatcher_time++;
  }
}
```

```
int main() {
    initialize_queue(&input_queue);
    initialize_queue(&real_time_queue);
    initialize_queue(&user_job_queue);
    for (int i = 0; i < NUM_FEEDBACK_QUEUES; i++) {
        initialize_queue(&feedback_queues[i]);
    }

    initialize_processes();
    manage_dispatch();

    return 0;
}
```

Output for the input sample:

```
PS C:\Users\prave> cd "c:\Users\prave\OneDrive\Desktop\OS\" ; if ($?) { g++ os.cpp -o os } ; if ($?) { .\os }
Time = 0: P0: Running P1: Pending P2: Pending P3: Pending P4: Pending P5: Pending
Time = 1: P0: Running P1: Pending P2: Pending P3: Pending P4: Pending P5: Pending
Time = 2: P0: Running P1: Pending P2: Pending P3: Pending P4: Pending P5: Pending
Time = 3: P0: Running P1: Pending P2: Pending P3: Pending P4: Pending P5: Pending
Time = 4: P0: Running P1: Pending P2: Pending P3: Pending P4: Pending P5: Pending
Time = 5: P0: Terminated P1: Running P2: Pending P3: Pending P4: Pending P5: Pending
Time = 6: P0: Terminated P1: Running P2: Pending P3: Pending P4: Pending P5: Pending
Time = 7: P0: Terminated P1: Running P2: Pending P3: Pending P4: Pending P5: Pending
Time = 8: P0: Terminated P1: Terminated P2: Running P3: Pending P4: Pending P5: Pending
Time = 9: P0: Terminated P1: Terminated P2: Running P3: Pending P4: Pending P5: Pending
Time = 10: P0: Terminated P1: Terminated P2: Running P3: Pending P4: Pending P5: Pending
Time = 11: P0: Terminated P1: Terminated P2: Running P3: Pending P4: Pending P5: Pending
Time = 12: P0: Terminated P1: Terminated P2: Running P3: Pending P4: Pending P5: Pending
Time = 13: P0: Terminated P1: Terminated P2: Running P3: Pending P4: Pending P5: Pending
Time = 14: P0: Terminated P1: Terminated P2: Terminated P3: Running P4: Pending P5: Pending
Time = 15: P0: Terminated P1: Terminated P2: Terminated P3: Running P4: Pending P5: Pending
Time = 16: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Running P5: Pending
Time = 17: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Running P5: Pending
Time = 18: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Running P5: Pending
Time = 19: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Running P5: Pending
Time = 20: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 21: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 22: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 23: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 24: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 25: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 26: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Running
Time = 27: P0: Terminated P1: Terminated P2: Terminated P3: Terminated P4: Terminated P5: Terminated
PS C:\Users\prave\OneDrive\Desktop\OS>
```

Key Observations:

1. Real-time processes are prioritized over user processes.
2. Feedback queues allow fair execution of lower-priority processes.
3. Resource constraints dynamically affect execution order and delays.

Conclusion:

This project successfully demonstrates the functionalities of process and resource management in an operating system through a simulation. It provides an understanding of how operating systems handle multiple processes by allocating limited resources dynamically and prioritizing tasks based on their urgency and requirements.

The use of real-time and feedback queue scheduling ensures that critical tasks are executed promptly while maintaining fairness for lower-priority processes. By implementing features such as dynamic resource allocation and process state transitions, the simulation closely mirrors real-world operating system behavior.

Additionally, the project highlights the complexities involved in process lifecycle management, such as handling resource constraints, preempting processes, and ensuring efficient queue management. The clear and step-by-step visualization of process states at each time unit provides a practical insight into the intricate decisions made by operating systems.

Overall, this project serves as a foundational learning tool for understanding the key principles of operating system design, offering a hands-on approach to studying process scheduling, resource management, and multi-level priority handling.

Challenges Faced

1. Dynamic Resource Allocation: Implementing checks for memory and resources without halting the system.
2. Queue Management: Efficiently transitioning processes between different queues.
3. Debugging State Changes: Ensuring correct updates to process states during preemption or suspension.

Contribution:

- Worked on the coding part of the project
  - Shaik. Ershad Ali - AP22110010941

- Prepared presentation for project and done research on the Topic
  - By Harshitha Challa - AP22110010942

- Done research on the Topic and Explained that to his teammates
  - Harsha Vardhan Reddy Vanga - AP22110010943

- Worked on the coding part of the project and helped in making report.
  - Sai Gopala Krishna Konjeti - AP22110010945

- Prepared report and suggested changes required to full fill the project requirement.
  - Praveen Kumar - AP22110010950

- Worked on the coding aspect of the project and Done some research regarding project.
  - Rohith Sai Gudibandla - AP22110010954