Operating Systems - Final Project

Process & Memory Manager

Salman Ahmed (SP22-BCT-042)

Hadia Naeem (SP22-BCT-015)

Course Instructor: Muhammad Mustafa Khattak

Date: 14/05/2024

INTRODUCTION

TITLE:

PROCESS AND MEMORY MANAGER

PURPOSE:

This project seeks to refine process management by employing inverted page tables instead of conventional ones. By leveraging this approach, it aims to streamline memory allocation, focusing on efficient utilization of real and virtual memory. The system enables creating, deleting, and displaying process status while monitoring memory usage and fragmentation. Integration of an Inverted Page Table optimizes resource allocation, enhancing overall system performance.

BACKGROUND:

In modern computer systems, efficient memory management is crucial for optimal performance. Traditional memory management techniques utilize page tables to map virtual addresses to physical memory locations. However, inverted page tables offer a more streamlined approach by indexing physical memory frames rather than virtual addresses. This inversion reduces memory overhead and lookup times, enhancing system efficiency. By utilizing inverted page tables, the system can manage memory allocation more effectively, ensuring that processes are allocated to the most appropriate memory locations. This approach optimizes resource utilization and reduces fragmentation, ultimately improving overall system performance.

PROBLEM STATEMENT:

The existing process management system operates without a dedicated memory management module, leading to inefficient memory utilization and performance bottlenecks. This project endeavors to address this gap by integrating a specialized memory management module into the system architecture. Without such a module, the system lacks the capability to efficiently allocate and manage memory resources, resulting in increased overhead and fragmentation issues. By introducing a dedicated memory manager, the system aims to optimize memory utilization, enhance performance, and improve overall system stability and reliability.

PROBLEM SOLUTION:

APPLICATION OF PROCESS & MEMORY MANAGER:

To tackle the memory management inefficiencies within the existing process manager, a comprehensive solution is proposed. This entails the integration of an inverted page table mechanism, complemented by the 2nd chance page replacement algorithm alongside the existing LRU algorithm. By incorporating these advanced techniques, the system can effectively manage memory allocation, reducing fragmentation and enhancing performance. This integrated approach ensures that processes are allocated to appropriate memory locations based on their priority and usage patterns, optimizing resource utilization. The seamless integration of the inverted page table and 2nd chance page replacement algorithm with the process manager offers a robust solution to the memory management challenges, resulting in improved system stability and efficiency.

OBJECTIVES:

1. **Implement Inverted Page Table:** Develop and integrate an inverted page table mechanism into the existing process manager to facilitate efficient mapping of virtual addresses to physical memory frames.
2. **Integrate 2nd Chance Page Replacement Algorithm:** Incorporate the 2nd chance page replacement algorithm alongside the existing LRU algorithm to optimize memory usage and minimize page faults.
3. **Enable Memory Allocation:** Enable the system to dynamically allocate and deallocate memory for processes based on their memory requirements, ensuring optimal resource utilization.
4. **Provide Memory Status Monitoring:** Implement features to display real-time status updates of both virtual and physical memory, allowing users to monitor memory usage, allocation, and fragmentation levels.
5. **Enhance Process Management:** Improve the overall process management capabilities of the system by integrating advanced memory management techniques, resulting in enhanced system performance, stability, and reliability.

RELATED SYSTEM ANALYSIS:

In comparison to related solutions addressing memory management within process management systems, this project offers a more comprehensive approach by integrating both an inverted page table mechanism and the 2nd chance page replacement algorithm. While some existing systems may utilize conventional page tables and basic page replacement algorithms, such as FIFO or LRU, they often encounter limitations in memory utilization and performance optimization. By incorporating an inverted page table, this project reduces memory overhead and lookup times, thus improving efficiency. Furthermore, the integration of the 2nd chance page replacement algorithm provides additional flexibility in managing page faults and optimizing memory usage. Together, these advanced techniques offer a more robust solution for memory management within process management systems, resulting in enhanced performance, stability, and resource utilization.

SCOPE:

This project aims to develop a comprehensive process management system with integrated memory management capabilities. It includes features such as process scheduling, suspension, and termination. Memory allocation is facilitated through an inverted page table mechanism, with support for page fault handling. The system employs optimization strategies for resource allocation, enhancing efficiency. Additionally, it supports various scheduling algorithms for process execution, ensuring optimal performance. Overall, this project provides a robust solution for managing processes and memory within a computing environment.

MODULES:

1. **Main Menu:** Manages core functionalities such as loading processes, viewing process status, and accessing sub-menus for process dispatching, addition, suspension, and termination. Additionally, it provides options to display memory-related information such as the inverted page table and internal fragmentation.
2. **Process Management:** Encompasses functions for adding, killing, suspending, resuming, and viewing processes. These functionalities ensure dynamic process management within the system.

3. **Memory Management:** Includes functions for handling memory-related operations such as allocating and deallocating memory, displaying available virtual memory, and monitoring internal fragmentation. These functions contribute to efficient memory utilization and optimization.
4. **Dispatch Menu:** Facilitates process dispatching and mode switching. Users can choose to dispatch available processes for execution or switch to auto mode for automated process execution. This module enhances process scheduling flexibility and automation.
5. **IPT Display:** Provides functionality to display the inverted page table, allowing users to visualize the mapping of virtual addresses to physical memory frames. This module aids in understanding memory allocation and utilization within the system.
6. **Process Execution:** Handles the execution of processes, including CPU bursts and scheduling decisions. Functions within this module determine the execution sequence of processes based on scheduling algorithms such as shortest time remaining first (STRF) and manage process execution until completion or suspension.

SYSTEM LIMITATIONS:

1. **Simulated CPU and I/O Bursts:** As the system simulates CPU and I/O bursts, the accuracy of process execution times may be limited by the simulation model's fidelity. While efforts are made to emulate real-world behavior, variations between simulated and actual execution times may occur, potentially impacting system performance predictions and behavior under different workloads.
2. **Simulated Virtual Memory:** The virtual memory system is simulated, implying that it may not fully replicate the intricacies and complexities of real-world memory management. While the simulation aims to provide a representative model of virtual memory operations, including page allocation, swapping, and page fault handling, certain nuances of real-world memory systems may not be captured accurately. Consequently, the system's behavior and performance in scenarios with heavy memory usage or dynamic memory demands may deviate from real-world counterparts. Users should interpret results and observations from the simulated virtual memory system with these limitations in mind.

ALGORITHMS:

addProcess()

```
function addProcess():
        repeat:
        display "Enter size of your process"
        input size
        until size >= 1

        repeat:
        display "Enter CPU bursts of your process"
        input cpu_bursts
        until cpu_bursts >= 1

        display "Enter priority of your process"
        input priority

        repeat:
        display "Does your process need any I/O? (0 for No, 1 for Yes)"
        input needs_io
        until needs_io is 0 or 1

        if needs_io is 1:
        for i from 1 to 3:
        repeat:
                display "Enter location of ith IO burst (0 for no IO burst)"
                input io_location_i
        until io_location_i >= 0 and io_location_i <= cpu_bursts

        repeat:
                display "Enter nature of ith IO burst (1 for Disk, 2 for Internet, 3 for
Printer)"
                input io_nature_i
        until io_nature_i >= 1 and io_nature_i <= 3

        repeat:
                display "Enter time of ith IO burst (in seconds)"
                input io_time_i
        until io_time_i >= 0

        create IO object i with io_location_i, io_nature_i, io_time_i
        create PCB object p with cpu_bursts, size, array of IO objects, and priority
        if processCanFit(p):
        allocateMemoryToProcess(p)
        insert p into RQ0
        else:
        display "Cannot allocate memory to process!"
        else:
        create PCB object p with cpu_bursts, size, and priority
        if processCanFit(p):
        allocateMemoryToProcess(p)
        insert p into RQ0
        else:
        display "Cannot allocate memory to process!"

        display "Added process"
```

allocateMemoryToProcess()

```
function allocateMemoryToProcess(p):
        size_in_int = ceil(p.size)
        fragmentation = modf(p.size / 3.0)
        TOTAL_INTERNAL_FRAGMENTATION += fragmentation
        FRAGMENTATION_RECORD.push(fragmentation)
        allocation = size_in_int
        frames_required = ceil(p.size / 3)
        real_allocation = frames_required
        VIRTUAL_MEMORY_AVAILABLE -= size_in_int

        for each frame in REAL_MEMORY:
        if frame is available and real_allocation is not 0:
        mark frame as unavailable
        assign frame number to IPT
        assign process ID to IPT
        decrement real_allocation by 1
```

deallocateMemory()

```
function deallocateMemory(p):
        for i from 0 to REAL_MEMORY_SIZE - 1:
        if IPT[i].pid equals p.pid:
        mark IPT[i] as available
        set page number of IPT[i] to 0
        set process ID of IPT[i] to 0

        subtract = FRAGMENTATION_RECORD.top()
        FRAGMENTATION_RECORD.pop()
        TOTAL_INTERNAL_FRAGMENTATION -= subtract
        VIRTUAL_MEMORY_AVAILABLE += ceil(p.size)
```

handlePageFault()

```
function handlePageFault(pid):
        display "Page fault occurred for process with PID: " + pid

        victimIndex = -1
        minReferenceTime = INT_MAX

        for i from 0 to 19:
        if IPT[i].pid equals pid and not IPT[i].reference and IPT[i].referenceTime <
minReferenceTime:
        victimIndex = i
        minReferenceTime = IPT[i].referenceTime

        if victimIndex is -1:
        for i from 0 to 19:
        if IPT[i].pid equals pid and IPT[i].referenceTime < minReferenceTime:
                victimIndex = i
                minReferenceTime = IPT[i].referenceTime

        if victimIndex is not -1:
        newSize = 0 // Placeholder for the size of the process
        newPageNo = 1
        found = false
        while not found and newPageNo < newSize:
        found = true
        for i from 0 to 19:
                if IPT[i].pid equals pid and IPT[i].page_no equals newPageNo:
                found = false
                break
        if not found:
                newPageNo++
        if found:
        IPT[victimIndex].pid = pid
        IPT[victimIndex].page_no = newPageNo
        IPT[victimIndex].modified = false
        IPT[victimIndex].reference = false
        IPT[victimIndex].referenceTime = currentTime

        display "Replaced page " + IPT[victimIndex].page_no + " in IPT"
        else:
        display "Error: Unable to find a suitable page for replacement"
        else:
        display "Error: Unable to find a suitable page for replacement"
```

executeBurst()

```
function executeBurst(p):
    if p.remaining_cpu_bursts equals 0:
    display "Process Finished"
    p.completion_history.completion_time = getCurrentTime()
    p.completion_history.completion_nature = 0
    call deallocateMemory(p)
    TQ.insertEnd(p)
    call appendToHistory(p)
    else:
    p.remaining_cpu_bursts--
    display "Remaining CPU Bursts: " + p.remaining_cpu_bursts + " "
    p.program_counter++
    display "Program Counter: " + p.program_counter

    random_value = random number between 0 and 99

    if random_value > 30:
    call handlePageFault(p.pid)

    for i from 0 to 2:
    if p.program_counter equals p.io_bursts[i].location:
        if p.io_bursts[i].nature equals 1:
        display "Servicing Disk I/O Request"
        else if p.io_bursts[i].nature equals 2:
        display "Servicing Internet I/O Request"
        else if p.io_bursts[i].nature equals 3:
        display "Servicing Printer I/O Request"

        sleep for p.io_bursts[i].time seconds
```

```
                    display "Serviced request!"
```

killProcess()

```
function killProcess():
          read pid from user

          found = ""
          res = NULL

          res = RQ0.searchNode(pid)
          if res is NULL:
          res = RQ1.searchNode(pid)
          if res is NULL:
          res = WQPrinter.searchNode(pid)
          if res is NULL:
                    res = WQInternet.searchNode(pid)
                    if res is NULL:
                    res = WQDisk.searchNode(pid)
                    if res is NULL:
                    res = SQ.searchNode(pid)
                    if res is not NULL:
                              found = "SQ"
                    else:
                    found = "WQDisk"
                    else:
                    found = "WQInternet"
          else:
                    found = "WQPrinter"
          else:
          found = "RQ1"
          else:
          found = "RQ0"

          if res is NULL:
          display "Process not found!"
          return
          else:
          res.process.state = 4
          res.process.completion_history.completion_time = getCurrentTime()
          res.process.completion_history.completion_nature = 1

          TQ.insertEnd(res.process)
          appendToHistory(res.process)
```

```
        if found equals "RQ0":
        RQ0.deleteSpecific(res.process.pid)
        else if found equals "RQ1":
        RQ1.deleteSpecific(res.process.pid)
        else if found equals "WQPrinter":
        WQPrinter.deleteSpecific(res.process.pid)
        else if found equals "WQInternet":
        WQInternet.deleteSpecific(res.process.pid)
        else if found equals "WQDisk":
        WQDisk.deleteSpecific(res.process.pid)
        else if found equals "SQ":
        SQ.deleteSpecific(res.process.pid)

        deallocateMemory(res.process)

        display "Killed Process"
```

suspendProcess()

```
function suspendProcess():
        read pid from user
        suspended_from = ""
        p = NULL

        // Search for the process in different queues
        res = RQ0.searchNode(pid)
        if res is NULL:
        res = RQ1.searchNode(pid)
        if res is NULL:
        res = RQ2.searchNode(pid)
        if res is NULL:
                res = WQPrinter.searchNode(pid)
                if res is NULL:
                res = WQInternet.searchNode(pid)
                if res is NULL:
                res = WQDisk.searchNode(pid)
                if res is NULL:
                        res = SQ.searchNode(pid)
                        if res is not NULL:
                        suspended_from = "SQ"
                        p = res.process
                        SQ.deleteSpecific(pid)
                else:
                        suspended_from = "WQDisk"
                        p = res.process
```

```
                    WQDisk.deleteSpecific(pid)
                else:
                suspended_from = "WQInternet"
                p = res.process
                WQInternet.deleteSpecific(pid)
                else:
                suspended_from = "WQPrinter"
                p = res.process
                WQPrinter.deleteSpecific(pid)
        else:
                suspended_from = "RQ2"
                p = res.process
                RQ2.deleteSpecific(pid)
        else:
        suspended_from = "RQ1"
        p = res.process
        RQ1.deleteSpecific(pid)
        else:
        suspended_from = "RQ0"
        p = res.process
        RQ0.deleteSpecific(pid)

        display "Suspended From: " + suspended_from
        if res is NULL:
        display "Process not found!"
        return
        else:
        // Check if the process is already suspended
        if res.process.state is 3:
        display "Process already suspended"
        return

        // Update process state and move it to the Suspended Queue (SQ)
        p.state = 3 // Set state to suspended
        p.suspended_from = suspended_from
        SQ.insertEnd(p) // Insert into Suspended Queue
        display "Process suspended and moved to the Suspended Queue (SQ) from "
+ suspended_from
```

resumeProcess()

```
function resumeProcess():
        if SQ.first is NULL:
        display "No processes to resume!"
        else:
```

```
read pid from user
res = SQ.searchNode(pid)
if res is NULL:
display "Process not found!"
else:
p = res.process
p.state = 1
if p.suspended_from is "RQ0":
        RQ0.insertEnd(p)
else if p.suspended_from is "RQ1":
        RQ1.insertEnd(p)
else if p.suspended_from is "RQ2":
        RQ2.insertEnd(p)
else if p.suspended_from is "WQInternet":
        WQInternet.insertEnd(p)
else if p.suspended_from is "WQDisk":
        WQDisk.insertEnd(p)
else if p.suspended_from is "WQPrinter":
        WQPrinter.insertEnd(p)
SQ.deleteSpecific(p.pid)
display "Process resumed!"
```

SYSTEM CODE:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <vector>
#include <thread>
#include <chrono>
#include <cmath>
#include <stack>
#include <ctime>
#include <random>


using namespace std;


string getCurrentTime();
int getRandomNumber(int min, int max);

void mainMenu();
void dispatchMenu();

void viewProcesses();
string getProcessState(int state);
void viewTerminatedProcesses();
void dispatchMenu();


void processMenu();
void addProcess();
void switchToAutoMode();

void suspendProcess();
void killProcess();
void resumeProcess();




void printTableHeader();
void readProcessData(char* filename);


unsigned int CURRENT_PID;

int VIRTUAL_MEMORY_SIZE = 200;
int REAL_MEMORY_SIZE = 20;

unsigned int REAL_MEMORY_AVAILABLE = REAL_MEMORY_SIZE;
unsigned int VIRTUAL_MEMORY_AVAILABLE = VIRTUAL_MEMORY_SIZE;

class Page {
    public:
            bool available;
            unsigned int pid;
            unsigned int page_no;
            bool modified;
            bool reference;
            time_t referenceTime;

            Page() {
                    this->available = true;
                    this->pid = 0;
                    this->page_no = 0;
                    this->modified = false;
                    this->reference = false;
                    referenceTime = 0;
            }
```

```
        Page(bool available, unsigned int pid, unsigned int page_no, bool modified, bool
reference) {
                this->available = available;
                this->pid = pid;
                this->page_no = page_no;
                this->modified = modified;
                this->reference = reference;
                referenceTime = 0;
        }

        void display() {
        std::cout << std::setw(10) << (available ? "True" : "False") << std::setw(5) << pid <<
std::setw(10) << page_no << std::setw(10) << (modified ? "True" : "False") << std::setw(10)
<< (reference ? "True" : "False") << std::endl;         }
};



Page IPT[20];



class IO {
  public:
        unsigned int location, nature, time;

        IO() {
                this->location = 0;
                this->nature = 0;
                this->time = 0;
        }

        IO(unsigned int location, unsigned int nature, unsigned int time) {
                this->location = location;
                this->nature = nature;
                this->time = time;
        }
};


class History {


  public:
  string starting_time, completion_time;
  int completion_nature;
        History() {
                this->starting_time = getCurrentTime();
                }
};
```

```
class PCB {
   public:
          unsigned int pid, ppid, cpu_bursts, program_counter, state, remaining_cpu_bursts,
remaining_quanta;
          double size;
          History *completion_history;
          IO io_bursts[3];
          int registers[5];
          int priority;
          string suspended_from;

          void display() {
                 cout << setw(10) << pid << setw(10) << ppid << setw(15) << cpu_bursts <<
setw(5) << size << setw(18) << program_counter
         << setw(10) << getProcessState(this->state) << setw(25) << remaining_cpu_bursts <<
setw(10) << priority << setw(25) << this->completion_history->starting_time << endl;
          }

          void displayTerminatedInfo() {
                 cout << setw(10) << pid << setw(10) << ppid << setw(15) << size << setw(10)
<< getProcessState(this->state) <<
                 setw(25) << this->completion_history->starting_time << setw(25) <<
this->completion_history->completion_time << setw(25) <<
this->completion_history->completion_nature << endl;
          }

          PCB() {

          }

          PCB(unsigned int cpu_bursts, double size, IO io_bursts[3], int priority) {
                 this->ppid = 0;
                 this->pid = ++CURRENT_PID;
                 this->cpu_bursts = cpu_bursts;
                 this->size = size;
                 this->priority = priority;
                 this->program_counter = 0;
                 this->remaining_cpu_bursts = cpu_bursts;
                 this->state = 1; // 0: Running, 1: Ready, 2: Waiting, 3: Suspended, 4:
Terminated
                 this->completion_history = new History();
                 for (int i=0; i<3; i++) {
                        this->io_bursts[i] = io_bursts[i];
                 }
          }

          PCB(unsigned int cpu_bursts, double size, unsigned int priority) {
                 this->ppid = 0;
                 this->pid = ++CURRENT_PID;
			this->cpu_bursts = cpu_bursts;
			this->size = size;
			this->priority = priority;
			this->program_counter = 0;
			this->remaining_cpu_bursts = cpu_bursts;
			this->state = 1; // 0: Running, 1: Ready, 2: Waiting, 3: Suspended, 4:
Terminated
			this->completion_history = new History();
			for (int i=0; i<3; i++) {
				this->io_bursts[i] = IO();
			}
		}
};


struct node {
   PCB process;
   node *next;
};


class LinkedList {
   public:
		node *first;
		node *last;
		LinkedList() {
			first = last = NULL;
		}

		void insertEnd(PCB &p);
		void insertStart(PCB p);
		void insertAfter(PCB old, PCB _new);
		node *searchNode(int pid);
		void deleteFirst();
		void deleteLast();
		void deleteAll();
		void searchSpecific(int pid);
		void searchMultiple(int pid);
		void deleteSpecific(int pid);
		void displayLinkedList();
		void displayTerminated();
		void displayReverse(node *head);
		node *popLast();
		node *popFirst();
		node *getFirst();

};


LinkedList RQ0;
LinkedList RQ1;
LinkedList RQ2;

LinkedList WQInternet;
LinkedList WQPrinter;
LinkedList WQDisk;

LinkedList TQ;
LinkedList SQ;

stack<float> FRAGMENTATION_RECORD;
float TOTAL_INTERNAL_FRAGMENTATION;

void appendToHistory(PCB process);

node *selectAvailableProcess();

void processMenu(PCB p);

void executeBurst(PCB &p);

void sendToRQ1(PCB p);
void sendToRQ2(PCB p);


void killCurrentProcess(PCB p);
void suspendCurrentProcess(PCB p);

void saveState(PCB p);

void roundRobin(PCB p);
void shortestTimeRemainingFirst(PCB p);
void priorityScheduler(PCB p);

void roundRobinProcessMenu(PCB p);

void displayIPT(Page (&arr)[20]);


void initializeMemory();

void allocateMemoryToProcess(PCB p);
bool processCanFit(PCB p);
void deallocateMemory(PCB p);
void handlePageFault(unsigned int pid);


int main() {
   mainMenu();
   return 0;
```

```
}


void mainMenu() {

    int choice;
    while (true) {
            cout<<"Process Manager Menu"<<endl;
            cout<<"1. Load Processes from File"<<endl;
            cout<<"2. View Active Processes"<<endl;
            cout<<"3. Dispatch Menu"<<endl;
            cout<<"4. Add a new process"<<endl;
            cout<<"5. Kill a process"<<endl;
            cout<<"6. Suspend a process"<<endl;
            cout<<"7. View terminated processes"<<endl;
            cout<<"8. Resume a process"<<endl;
            cout<<"9. Display Inverted Page Table"<<endl;
            cout<<"10. Display available virtual memory"<<endl;
            cout<<"11. Display internal fragmentation"<<endl;
            cout<<"0. Exit"<<endl;
            cin>>choice;

            switch (choice) {
                    case 1:
                            readProcessData("input.txt");
                            break;
                    case 2:
                            viewProcesses();
                            break;
                    case 3:
                            dispatchMenu();
                            break;
                    case 4:
                            addProcess();
                            break;
                    case 5:
                            killProcess();
                            break;
                    case 6:
                            suspendProcess();
                            break;
                    case 7:
                            viewTerminatedProcesses();
                            break;
                    case 8:
                            printTableHeader();
                            SQ.displayLinkedList();
                            resumeProcess();
                            break;
                    case 9:
```

```
                              displayIPT(IPT);
                              break;
                    case 10:
                              cout<<"Available Virtual Memory:
"<<VIRTUAL_MEMORY_AVAILABLE<<endl;
                    case 11:
                              cout<<"Total Internal Fragmentation:
"<<TOTAL_INTERNAL_FRAGMENTATION<<endl;
                              break;
                    case 0:
                              exit(0);
                              break;
                    default:
                              cout<<"Invalid choice"<<endl;
          }

   }
}

void readProcessData(char *filename) {
   ifstream file(filename);
         if (!file.is_open()) {
         cerr << "Error opening file: " << filename << endl;
         return;
         }

         string line;
         while (getline(file, line)) {
         stringstream ss(line);
         vector<string> values;
         string value;

         // Read comma-separated values from the line
         while (getline(ss, value, ',')) {
         values.push_back(value);
         }

         // Check if there are exactly 12 values
         if (values.size() != 12) {
         cerr << "Error: Expected 12 values, but found " << values.size() << " values in line." <<
endl;
         continue;
         }

         // Extract values
         unsigned int cpu_bursts = stoul(values[0]);
         double size = stod(values[1]);
         int priority = stoi(values[2]);
         int io_location1 = stoi(values[3]);
         int io_nature1 = stoi(values[4]);
```

```
	int io_time1 = stoi(values[5]);
	int io_location2 = stoi(values[6]);
	int io_nature2 = stoi(values[7]);
	int io_time2 = stoi(values[8]);
	int io_location3 = stoi(values[9]);
	int io_nature3 = stoi(values[10]);
	int io_time3 = stoi(values[11]);



	if (
	cpu_bursts<0 || size<0 || (io_location1<0 || io_location1 > cpu_bursts) || (io_location2 <
0 || io_location2 > cpu_bursts)
	 || (io_location3 < 0 || io_location3 > cpu_bursts) || (io_nature1 < 0 || io_nature1 > 3) ||
(io_nature2 < 0 || io_nature2 > 3)
      || (io_nature3 < 0 || io_nature3 > 3) || io_time1<=0 || io_time2<=0 || io_time3<=0
      ) {

	cout << "Invalid value detected while reading a process from file\nProcess will not be
added!" << endl;




	} else {
	//cout << "Process will be created" << endl;

		IO i1(io_location1, io_nature1, io_time1);
		IO i2(io_location2, io_nature2, io_time2);
		IO i3(io_location3, io_nature3, io_time3);

		IO ios[3];
		ios[0] = i1; ios[1] = i2; ios[2] = i3;

		PCB p(cpu_bursts, size, ios, priority);
		if (processCanFit(p)) {
			allocateMemoryToProcess(p);
			RQ0.insertEnd(p);
			cout<<"Processes created!"<<endl;
		} else {
			cout<<"Cannot allocate memory to process!"<<endl;
		}

	}
	}

	file.close();
}

void viewProcesses() {
    cout<<"Process Informaion"<<endl;
    cout<<"\nReady Queue 0\n"<<endl;
    printTableHeader();
    RQ0.displayLinkedList();
    cout<<"\nReady Queue 1\n"<<endl;
    RQ1.displayLinkedList();
    cout<<"\nReady Queue 2\n"<<endl;
    RQ2.displayLinkedList();
    cout<<"\nInternet Waiting Queue\n"<<endl;
    WQInternet.displayLinkedList();
    cout<<"\nDisk Waiting Queue\n"<<endl;
    WQDisk.displayLinkedList();
    cout<<"\nPrinter Waiting Queue\n"<<endl;
    WQPrinter.displayLinkedList();
    cout<<"\nSuspended Queue\n"<<endl;
    SQ.displayLinkedList();
    cout<<endl;
}

void dispatchMenu() {
    int choice;
    while (true) {
        cout<<"Dispatch Menu"<<endl;
        cout<<"1. Dispatch available process"<<endl;
        cout<<"2. Switch mode to auto"<<endl;
        cout<<"0. Go back"<<endl;
        cin>>choice;

        switch (choice) {
            case 1: {
                node *n = selectAvailableProcess();
                if (n == NULL) {
                    cout<<"No process available!"<<endl;
                } else {
                    if (RQ0.first==NULL) {
                        shortestTimeRemainingFirst(n->process);
                    } else {
                        executeBurst(n->process);
                        executeBurst(n->process);
                        executeBurst(n->process);
                        executeBurst(n->process);
                        cout<<"Executed process"<<endl;
                        cout<<"CPU time expired, moving to RQ1"<<endl;
                        sendToRQ1(n->process);
                    }
                }
                break;
            }
            case 2:
                switchToAutoMode();
```

```
                        break;
                case 0:
                        return;
                default:
                        cout<<"Invalid choice"<<endl;
                        break;
        }
    }
}

void processMenu(PCB p) {
    int choice;
    while (true) {
        cout<<"Process Execution Menu"<<endl;
        cout<<"1. Execute next burst"<<endl;
        cout<<"0. Go back"<<endl;
        cin>>choice;

        switch (choice) {
                case 1:
                        executeBurst(p);
                        cout<<"Executed next burst"<<endl;
                        break;
                case 0:
                        saveState(p);
                        return;
                default:
                        break;
        }
    }
}

void roundRobinProcessMenu(PCB p) {
    int choice;
    bool executed = false;
    while (!executed) {
        cout<<"Process Execution Menu"<<endl;
        cout<<"1. Execute process"<<endl;
        cout<<"0. Go back"<<endl;
        cin>>choice;

        switch (choice) {
                case 1:
                        executeBurst(p);
                        executeBurst(p);
                        executeBurst(p);
                        executeBurst(p);
                        cout<<"Executed process"<<endl;
                        executed = true;
                        break;
```

```
                    case 0:
                              saveState(p);
                              return;
                    default:
                              break;

          }
   }
   cout<<"CPU time expired, moving to RQ1"<<endl;
   sendToRQ1(p);
}


string getCurrentTime() {
          // Get current time
          auto now = chrono::system_clock::now();
          time_t now_c = chrono::system_clock::to_time_t(now);

          // Convert to struct tm for easy formatting
          struct tm *tm_struct = localtime(&now_c);

          // Construct the time string
          stringstream ss;
          ss << put_time(tm_struct, "%H:%M:%S:%d-%m-%Y");
          return ss.str();
}


void killProcess() {
          int pid;
          cout << "Enter PID of the process: ";
          cin >> pid;
          string found = "";
          node * res;
          res = RQ0.searchNode(pid);
          if (res == NULL) {
          res = RQ1.searchNode(pid);
          if (res == NULL) {
          res = RQ0.searchNode(pid);
          if (res == NULL) {
                    res = WQPrinter.searchNode(pid);
                    if (res == NULL) {
                    res = WQInternet.searchNode(pid);
                    if (res == NULL) {
                    res = WQDisk.searchNode(pid);
                    if (res == NULL) {
                              res = SQ.searchNode(pid);
                              if (res != NULL) {
                              found = "SQ"; // Process found in SQ
                              }
```

```
                        } else {
                                found = "WQDisk"; // Process found in WQDisk
                        }
                        } else {
                        found = "WQInternet"; // Process found in WQInternet
                        }
                        } else {
                        found = "WQPrinter"; // Process found in WQPrinter
                        }
            } else {
                        found = "RQ0"; // Process found in RQ0
            }
            } else {
            found = "RQ1"; // Process found in RQ1
            }
            } else {
            found = "RQ0"; // Process found in RQ0
            }

            if (res == NULL) {
            cout << "Process not found!" << endl;
            return;
            } else {
            res -> process.state = 4;
            res -> process.completion_history -> completion_time = getCurrentTime();
            res -> process.completion_history -> completion_nature = 1;
            TQ.insertEnd(res -> process);
            appendToHistory(res->process);

            if (found == "RQ0") {
            RQ0.deleteSpecific(res -> process.pid);
            } else if (found == "RQ1") {
            RQ1.deleteSpecific(res -> process.pid);
            } else if (found == "WQPrinter") {
            WQPrinter.deleteSpecific(res -> process.pid);
            } else if (found == "WQInternet") {
            WQInternet.deleteSpecific(res -> process.pid);
            } else if (found == "WQDisk") {
            WQDisk.deleteSpecific(res -> process.pid);
            } else if (found == "SQ") {
            SQ.deleteSpecific(res -> process.pid);
            }
             deallocateMemory(res->process);
            }

            cout << "Killed Process" << endl;
}

void suspendProcess() {
            int pid;
```

```
cout << "Enter PID of the process: ";
cin >> pid;

string suspended_from;

PCB p;

// Search for the process in different queues
node* res = RQ0.searchNode(pid);
if (res == NULL) {
res = RQ1.searchNode(pid);
if (res == NULL) {
res = RQ2.searchNode(pid);
if (res == NULL) {
        res = WQPrinter.searchNode(pid);
        if (res == NULL) {
        res = WQInternet.searchNode(pid);
        if (res == NULL) {
        res = WQDisk.searchNode(pid);
        if (res == NULL) {
                res = SQ.searchNode(pid);
                if (res != NULL) {
                suspended_from = "SQ";
                p = res->process;
                SQ.deleteSpecific(pid);
                }
        } else {
                suspended_from = "WQDisk";
                p = res->process;
                WQDisk.deleteSpecific(pid);
        }
        } else {
        suspended_from = "WQInternet";
        p = res->process;
        WQInternet.deleteSpecific(pid);
        }
        } else {
        suspended_from = "WQPrinter";
        p = res->process;
         WQPrinter.deleteSpecific(pid);
        }
} else {
        suspended_from = "RQ2";
        p = res->process;
        RQ2.deleteSpecific(pid);
}
} else {
suspended_from = "RQ1";
p = res->process;
RQ1.deleteSpecific(pid);
```

```
            }
            } else {
            suspended_from = "RQ0";
            p = res->process;
            RQ0.deleteSpecific(pid);
            }
    cout<<"Suspended From: "<<suspended_from<<endl;
            if (res == NULL) {
            cout << "Process not found!" << endl;
            return;
            } else {
            // Check if the process is already suspended
            if (res->process.state == 3) {
            cout << "Process already suspended" << endl;
            return;
            }

            // Update process state and move it to the Suspended Queue (SQ)
            p.state = 3; // Set state to suspended
            cout<<"I am here!"<<endl;
            p.suspended_from = suspended_from;
            SQ.insertEnd(p); // Insert into Suspended Queue
            cout << "Process suspended and moved to the Suspended Queue (SQ) from " <<
suspended_from << endl;
            }
}



void saveState(PCB p) {
    if (RQ0.first!=NULL) {
             RQ0.insertStart(p);
    } else if (RQ0.first == NULL) {
             RQ1.insertStart(p);
    } else if (RQ1.first == NULL) {
             RQ2.insertStart(p);
    }
    cout<<"Saved State"<<endl;
}

void addProcess() {
    int cpu_bursts, priority;
    double size;
    int io_location1, io_location2, io_location3, io_nature1, io_nature2, io_nature3,  io_time1,
io_time2, io_time3;
    int needs_io;

    do {
             cout<<"Enter size of your process"<<endl;
             cin>>size;
```

```
   } while (size<1);

   do {
         cout<<"Enter CPU bursts of your process"<<endl;
         cin>>cpu_bursts;
   } while (cpu_bursts<1);

   cout<<"Enter priority of your process"<<endl;
   cin>>priority;

   do {
         cout<<"Does your process need any I/O?"<<endl;
         cin>>needs_io;
   } while (needs_io>1 || needs_io<0);

   if (needs_io==1) {
         do {
                  cout<<"Enter location of 1st IO burst (0 for no IO burst)"<<endl;
                  cin>>io_location1;
         } while (io_location1<0 || io_location1>cpu_bursts);

         do {
                  cout<<"Enter nature of 1st IO burst (1 for Disk, 2 for Internet, 3 for
Printer)"<<endl;
                  cin>>io_nature1;
         } while (io_nature1>3 || io_nature1 <1);

         do {
                  cout<<"Enter time of 1st IO burst (in seconds)"<<endl;
                  cin>>io_time1;
         } while (io_time1<0);


         do {
                  cout<<"Enter location of 2nd IO burst (0 for no IO burst)"<<endl;
                  cin>>io_location1;
         } while (io_location2<0 || io_location2>cpu_bursts);

         do {
                  cout<<"Enter nature of 2nd IO burst (1 for Disk, 2 for Internet, 3 for
Printer)"<<endl;
                  cin>>io_nature2;
         } while (io_nature2>3 || io_nature2 <1);

         do {
                  cout<<"Enter time of 2nd IO burst (in seconds)"<<endl;
                  cin>>io_time2;
         } while (io_time2<0);
```

```
            do {
                        cout<<"Enter location of 3rd IO burst (0 for no IO burst)"<<endl;
                        cin>>io_location1;
            } while (io_location3<0 || io_location3>cpu_bursts);


            do {
                        cout<<"Enter nature of 3rd IO burst (1 for Disk, 2 for Internet, 3 for
Printer)"<<endl;
                        cin>>io_nature3;
            } while (io_nature3>3 || io_nature3 <1);

            do {
                        cout<<"Enter time of 3rd IO burst (in seconds)"<<endl;
                        cin>>io_time3;
            } while (io_time3<0);


            IO i1(io_location1, io_nature1, io_time1);
            IO i2(io_location2, io_nature2, io_time2);
            IO i3(io_location3, io_nature3, io_time3);
            IO ios[3];
            ios[0] = i1;
            ios[1] = i2;
            ios[3] = i3;
            PCB p(cpu_bursts, size, ios, priority);
            if (processCanFit(p)) {
                        allocateMemoryToProcess(p);
                        RQ0.insertEnd(p);
            } else {
                        cout<<"Cannot allocate memory to process!"<<endl;
            }

    } else {
            PCB p(cpu_bursts,size,priority);
            if (processCanFit(p)) {
                        allocateMemoryToProcess(p);
                        RQ0.insertEnd(p);
            } else {
                        cout<<"Cannot allocate memory to process!"<<endl;
            }

    }



    cout<<"Added process"<<endl;
}

void switchToAutoMode() {
    cout<<"Switched to Auto Mode"<<endl;

    while (RQ0.first!=NULL) {
            node *n = RQ0.popFirst();
            PCB p = n->process;

            roundRobin(p);
    }
    if (RQ0.first==NULL) {
            while(RQ1.first!=NULL) {
                    node *n = RQ1.popFirst();
                    PCB p = n->process;
                    shortestTimeRemainingFirst(p);
            }
    }
    if (RQ0.first==NULL && RQ1.first==NULL) {
            while (RQ2.first!=NULL) {
                    node *n = RQ2.popFirst();
                    PCB p = n->process;
                    priorityScheduler(p);
            }
    }

}






void LinkedList::insertEnd(PCB &proc) {
            node *p = new node;
            p->process = proc;

            if (first == NULL) {
            first = last = p;
            } else {
            last->next = p;
            last = p;
            }
}

void LinkedList::insertStart(PCB proc) {
            node *p = new node;
            p->process = proc;
            if (first == NULL) {
            first = last = p;
            } else {
            p->next = first;
            first = p;
```

```
	}
}



void LinkedList::deleteFirst() {
	if (first == NULL) {
	cout<<"Linked list empty!"<<endl;
	} else {
	node *temp = first;
	first = first->next;
	delete temp;
	}
}

node *LinkedList::popFirst() {
	if (first == NULL) {
	cout<<"Linked list empty!"<<endl;
	return NULL;
	} else {
	node *temp = first;
	first = first->next;
	return temp;
	}
}

void LinkedList::deleteLast() {
	if (first == NULL) {
	cout<<"Linked list empty!"<<endl;
	} else {
	if (first == last) {
	delete first;
	first = NULL;
	} else {
	node *q = first;
	node *q1 = NULL;
	while (q!=last) {
	q1 = q;
	q = q->next;
	}
	delete q;
	last = q1;
	last->next = NULL;
	}
	}
}

node *LinkedList::popLast() {
	if (first == NULL) {
	cout<<"Linked list empty!"<<endl;
```

```
        } else {
        if (first == last) {
        node *t = first;
        delete first;
        first = NULL;
        return t;
        } else {
        node *q = first;
        node *q1 = NULL;
        while (q!=last) {
        q1 = q;
        q = q->next;
        }
        last = q1;
        last->next = NULL;
        return q;
        }
        }
}

void LinkedList::deleteAll() {
        if (first == NULL) {
        cout<<"Linked list empty!"<<endl;
        } else {
        while (first!=NULL) {
        deleteFirst();
        }
        cout<<"Deleted linked list"<<endl;
        }
}

node* LinkedList::searchNode(int pid) {
        if (first == NULL) {
        cout<<"Linked list empty!"<<endl;
        return NULL;
        } else {
        node *temp = first;
        while (temp!=NULL) {
        if (temp->process.pid == pid) {
                return temp;
        }
        temp = temp->next;
        }
        return NULL;
        }
}

void LinkedList::searchSpecific(int pid) {
        if (first == NULL) {
        cout<<"Linked list empty!"<<endl;
        } else {
        node *res = searchNode(pid);
        if (res==NULL) {
        cout<<"Element not found!"<<endl;
        } else {
        cout<<"Found "<<res->process.pid<<endl;
        res->process.display();
        }
        }
}



void insertElements() {
        int elem;
        char choice;
        while (true) {
        cout<<"Enter elements"<<endl;
        cin>>elem;
        //insertEnd(elem);
        cout<<"Do you want to continue? (y/n)"<<endl;
        cin>>choice;
        if (choice == 'n') {
        cout<<endl;
        return;
        }
        }
}

void LinkedList::deleteSpecific(int pid) {
        if (first == NULL) {
        cout<<"Linked List is empty!"<<endl;
        } else {
        if (first->process.pid == pid) {
        deleteFirst();
        } else if (last->process.pid == pid) {
        deleteLast();
        } else {
        node *q1 = NULL;
        node *q = first;
        while (q!=NULL) {
                if (q->process.pid == pid) {
                break;
                }
                q1 = q;
                q = q->next;
        }

        if (q == NULL) {
                cout<<"Element not found!"<<endl;
```

```
                        return;
            } else {
                        q1->next = q->next;
                        delete q;
            }
            }
            }
}


void LinkedList::displayLinkedList() {
            if (first == NULL) {
            cout<<"Linked List is empty!"<<endl;
            } else {
            node *temp = first;
            while (temp!=NULL) {
            temp->process.display();
            temp = temp->next;
            }
            }
            cout<<endl;
}

void LinkedList::displayTerminated() {
            if (first == NULL) {
            cout<<"Linked List is empty!"<<endl;
            } else {
            node *temp = first;
            while (temp!=NULL) {
            temp->process.displayTerminatedInfo();
            temp = temp->next;
            }
            }
            cout<<endl;
}

void LinkedList::insertAfter(PCB old, PCB _new) {
            if (first == NULL) {
            cout<<"Linked list is empty!"<<endl;
            } else {
            node *res = searchNode(old.pid);
            if (res == NULL) {
            cout<<"Element not found!"<<endl;
            } else if (res == last) {
            insertEnd(_new);
            cout<<"Inserted!"<<endl;
            } else {
            node *p = new node;
            p->process = _new;
            p->next = res->next;
```

```
        res->next = p;
        cout<<"Inserted!"<<endl;
        }
        }
}


node *LinkedList::getFirst() {
   if (first==NULL) {
         return NULL;
   } else {
         return first;
   }
}

void printTableHeader() {
   cout << setw(10) << "PID" << setw(10) << "PPID" << setw(15) << "CPU Bursts" << setw(5)
<< "Size" << setw(18)
        << "Program Counter" << setw(10) << "State" << setw(25) << "Remaining CPU
Bursts" << setw(10) << "Priority"<< setw(15) << "Creation Time" <<endl;
}

void printTerminatedHeader() {
   cout << setw(10) << "PID" << setw(10) << "PPID" << setw(15) << "Size" << setw(10) <<
"State" <<
        setw(25) << "Starting Time" << setw(25) << "Completion Time" << setw(25) <<
"Completion Nature" << endl;

}

void viewTerminatedProcesses() {
   printTerminatedHeader();
   TQ.displayTerminated();
}


void appendToHistory(PCB process) {
        // Open the file in append mode
        ofstream historyFile("history.txt", ios::app);
        if (!historyFile.is_open()) {
        cerr << "Error opening history file!" << endl;
        return;
        }

        // Write process information to the file
        historyFile << "Process ID: " << process.pid << endl;
        historyFile << "Parent PID: " << process.ppid << endl;
        historyFile << "Size: " << process.size << endl;
        historyFile << "Starting Time: " << process.completion_history->starting_time << endl;
        historyFile << "Completion Time: " << process.completion_history->completion_time
```

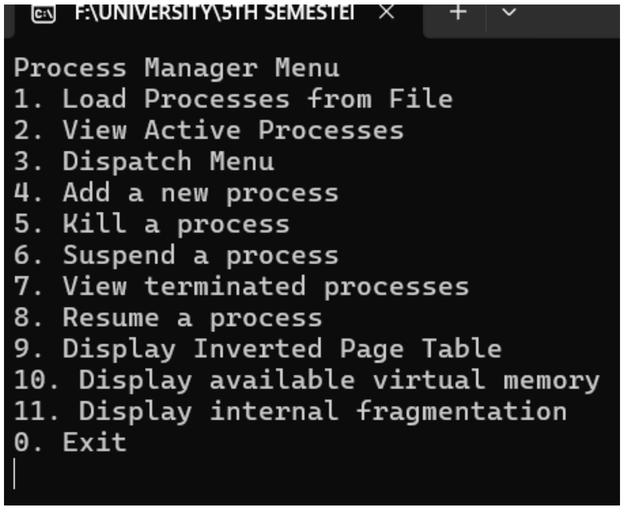
```
<< endl;
          historyFile << "Completion Nature: ";
          if (process.completion_history->completion_nature == 0) {
          historyFile << "Normal" << endl;
          } else {
          historyFile << "Abnormal" << endl;
          }

          // Close the file
          historyFile.close();
}


node *selectAvailableProcess() {
   node *p;
   if (!(RQ0.first==NULL)) {
          p = RQ0.popFirst();
          return p;
   }
   if (RQ0.first==NULL && RQ1.first!=NULL) {
          p = RQ1.popFirst();
          return p;
   }
   if (RQ1.first==NULL && RQ2.first!=NULL) {
          p = RQ2.popFirst();
          return p;
   } else {
          return NULL;
   }

}

void executeBurst(PCB &p) {
          if (p.remaining_cpu_bursts == 0) {
          cout << "Process Finished" << endl;
          p.completion_history->completion_time = getCurrentTime();
          p.completion_history->completion_nature = 0;
          deallocateMemory(p);
          TQ.insertEnd(p);
          appendToHistory(p);
          } else {
          p.remaining_cpu_bursts--;
          cout << "Remaining CPU Bursts: " << p.remaining_cpu_bursts << " ";
          p.program_counter++;
          cout << "Program Counter: " << p.program_counter << endl;

          int random_value = rand() % 100;

          // Simulate the possibility of a page fault occurring during a CPU burst
          if ( random_value < 30) {
          cout << "Page fault occurred for process with PID: " << p.pid << endl;
          int ipt_index;
          for (int i = 0; i < 20; i++) {
                      if (IPT[i].pid == p.pid) {
                      ipt_index = i;
                      break;
                      }
          }
          int page_replaced = rand() % (int)p.size;
          IPT[ipt_index].modified = false;
          IPT[ipt_index].reference = false;
          IPT[ipt_index].page_no = page_replaced;

          cout << "Page fault handled!" << endl;
          }

          // Check if there is an I/O burst at the current program counter
          // location
          for (int i = 0; i < 3; i++) {
          if (p.program_counter == p.io_bursts[i].location) {
                      // If there is an I/O burst, service the request
                      if (p.io_bursts[i].nature == 1) {
                      cout << "Servicing Disk I/O Request" << endl;
                      } else if (p.io_bursts[i].nature == 2) {
                      cout << "Servicing Internet I/O Request" << endl;
                      } else if (p.io_bursts[i].nature == 3) {
                      cout << "Servicing Printer I/O Request" << endl;
                      }
                      // Simulate servicing the I/O request
                      this_thread::sleep_for(chrono::seconds(p.io_bursts[i].time));
                      cout << "Serviced request!" << endl;
          }
          }
          }
}



void sendToRQ1(PCB p) {
   if (RQ1.first==NULL) {
          RQ1.insertEnd(p);
   } else {
          node *top = RQ1.getFirst();
          if (top->process.remaining_cpu_bursts <= p.remaining_cpu_bursts) {
                      node *popped = RQ1.popFirst();
                      RQ1.insertStart(p);
                      RQ1.insertStart(popped->process);
          } else {
                      RQ1.insertStart(p);
          }
```

```
    }
}

void sendToRQ2(PCB p) {
    if (RQ2.first==NULL) {
            RQ2.insertEnd(p);
    } else {
            node *top = RQ2.getFirst();
            if (top->process.priority <= p.priority) {
                    node *popped = RQ2.popFirst();
                    RQ2.insertStart(p);
                    RQ2.insertStart(popped->process);
            } else {
                    RQ2.insertStart(p);
            }
    }
}

void resumeProcess() {
        if (SQ.first == NULL) {
        cout << "No processes to resume!" << endl;
        } else {
        int pid;
        cout << "Enter PID of process to resume" << endl;
        cin >> pid;

        node * res = SQ.searchNode(pid);
        if (res == NULL) {
        cout << "Process not found!" << endl;
        } else {
        PCB p = res -> process;
        p.state = 1;

        if (p.suspended_from == "RQ0") {
                cout<<"I am here"<<endl;
                RQ0.insertEnd(p);
        } else if (p.suspended_from == "RQ1") {
                RQ1.insertEnd(p);
        } else if (p.suspended_from == "RQ2") {
                RQ2.insertEnd(p);
        } else if (p.suspended_from == "WQInternet") {
                WQInternet.insertEnd(p);
        } else if (p.suspended_from == "WQDisk") {
                WQDisk.insertEnd(p);
        } else if (p.suspended_from == "WQPrinter") {
                WQPrinter.insertEnd(p);
        }
        SQ.deleteSpecific(p.pid);
        cout<<"Process resumed!"<<endl;
        }
        }
}

void roundRobin(PCB p) {
   for (int i = 0; i<4; i++) {
        executeBurst(p);
   }
   //this_thread::sleep_for(chrono::seconds(1));
   sendToRQ1(p);
}

void shortestTimeRemainingFirst(PCB p) {
   while (p.remaining_cpu_bursts!=0) {
        executeBurst(p);
   }
   deallocateMemory(p);
   this_thread::sleep_for(chrono::seconds(1));
}

void priorityScheduler(PCB p) {
   while (p.remaining_cpu_bursts!=0) {
        executeBurst(p);
   }
   deallocateMemory(p);
   this_thread::sleep_for(chrono::seconds(1));
}


string getProcessState(int state) {
   switch (state) {
        case 0:
                return "Running";
        case 1:
                return "Ready";
                break;
        case 2:
                return "Waiting";
        case 3:
                return "Suspended";
        case 4:
                return "Terminated";
        default:
                return "Invalid";
   }
}

bool processCanFit(PCB p) {
   cout<<"SIZE OF PROCESS IS: "<<ceil(p.size)<<endl;
   return ceil(p.size)<VIRTUAL_MEMORY_AVAILABLE;
```

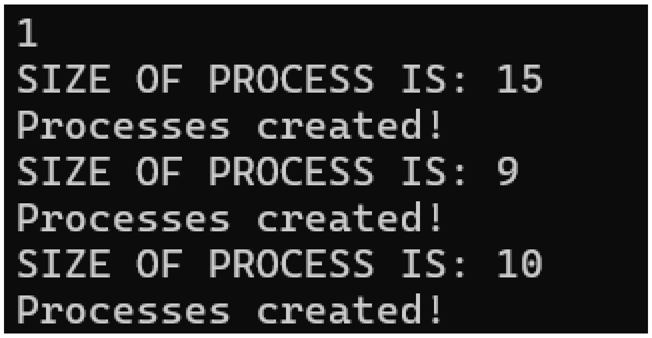
```
}

void allocateMemoryToProcess(PCB p) {
   int size_in_int = ceil(p.size);
   double fragmentation = modf(p.size/3.0, nullptr);
   TOTAL_INTERNAL_FRAGMENTATION += fragmentation;
   FRAGMENTATION_RECORD.push(fragmentation);
   int allocation = size_in_int;
   int frames_required = ceil(p.size/3);
   int real_allocation = frames_required;
   VIRTUAL_MEMORY_AVAILABLE-=size_in_int;

   for (int i=0; i<REAL_MEMORY_SIZE; i++) {
         if (IPT[i].available && real_allocation!=0) {
                 cout<<"I go here"<<endl;
                 IPT[i].available = false;
                 IPT[i].page_no = i;
                 IPT[i].pid = p.pid;
                 real_allocation--;
         }
   }
}

void deallocateMemory(PCB p) {
   for(int i=0; i<REAL_MEMORY_SIZE; i++) {
         if (IPT[i].pid == p.pid) {
                 IPT[i].available = true;
                 IPT[i].page_no = 0;
                 IPT[i].pid = 0;
         }
   }
   double subtract = FRAGMENTATION_RECORD.top();
   FRAGMENTATION_RECORD.pop();
   TOTAL_INTERNAL_FRAGMENTATION -= subtract;
   VIRTUAL_MEMORY_AVAILABLE+=ceil(p.size);
}

void initializeMemory() {
   for (int i=0; i<REAL_MEMORY_SIZE; i++) {
         IPT[i] = Page();
   }
}

void displayIPT(Page (&arr)[20]) {
        std::cout << std::left << std::setw(10) << "Available" << std::setw(5) << "PID" <<
std::setw(10) << "Page No" << std::setw(10) << "Modified" << std::setw(10) << "Reference"
<< std::endl;    for(int i=0; i<20; i++) {
         arr[i].display();
   }
}
```

```
int getRandomNumber(int min, int max) {
        // Seed the random number generator using the current time
        mt19937 rng(std::time(nullptr));

        // Define the distribution for the range [min, max]
        uniform_int_distribution<int> distribution(min, max);

        // Generate and return a random number within the specified range
        return distribution(rng);
}

void handlePageFault(unsigned int pid) {
        // Simulate page fault handling
        cout << "Page fault occurred for process with PID: " << pid << endl;

        // Find a page to replace using the 2nd chance page replacement algorithm with LRU
scheme
        int victimIndex = -1;
        time_t minReferenceTime = INT_MAX; // Initialize with maximum value
        for (int i = 0; i < 20; ++i) {
        // Check if the page is available and not referenced
        if (IPT[i].pid == pid && !IPT[i].reference && IPT[i].referenceTime < minReferenceTime)
{
        victimIndex = i;
        minReferenceTime = IPT[i].referenceTime;
        }
        }

        // If no unreferenced page found, use LRU to find the least recently used page
        if (victimIndex == -1) {
        // Use LRU scheme to find the least recently used page
        for (int i = 0; i < 20; ++i) {
        if (IPT[i].pid == pid && IPT[i].referenceTime < minReferenceTime) {
                victimIndex = i;
                minReferenceTime = IPT[i].referenceTime;
        }
        }
        }

        if (victimIndex != -1) {
        // Generate a new page number that isn't already in use by the process
        int newSize = 0; // Placeholder for the size of the process (you need to provide this
value)
        int newPageNo = 1;
        bool found = false;
        while (!found && newPageNo < newSize) {
        found = true; // Assume the page number is valid unless proven otherwise
        for (int i = 0; i < 20; ++i) {
                if (IPT[i].pid == pid && IPT[i].page_no == newPageNo) {
                found = false; // Page number already exists in the process's IPT entries
                break;
                }
        }
        if (!found) {
                newPageNo++;
        }
        }

        if (found) {
        // Replace the victim page with the new page
        IPT[victimIndex].pid = pid;
        IPT[victimIndex].page_no = newPageNo;
        IPT[victimIndex].modified = false;
        IPT[victimIndex].reference = false;
        IPT[victimIndex].referenceTime = time(nullptr); // Update reference time

        // Update virtual memory accordingly (not implemented here)

        // Output information about the replacement
        cout << "Replaced page " << IPT[victimIndex].page_no << " in IPT" << endl;
        } else {
        // No suitable page found, unable to replace the page
        cout << "Error: Unable to find a suitable page for replacement" << endl;
        }
        } else {
        // No suitable page found, unable to replace the page
        cout << "Error: Unable to find a suitable page for replacement" << endl;
        }
}
```

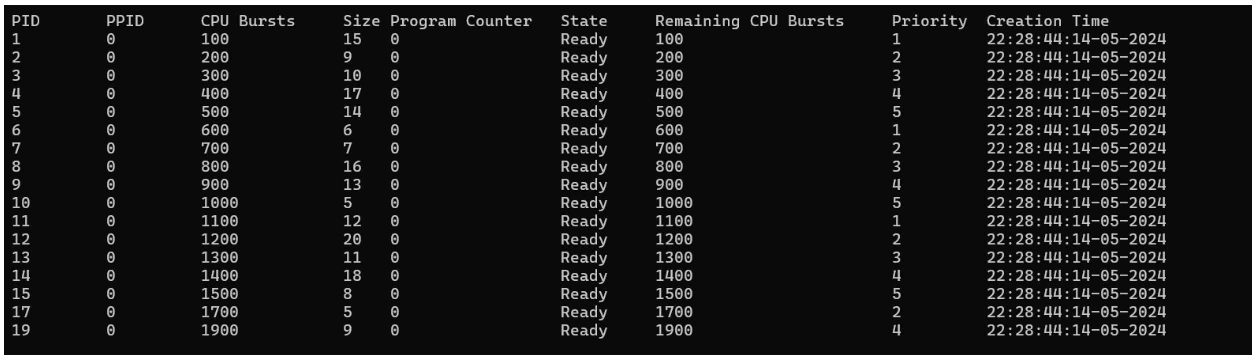
SCREENSHOTS:



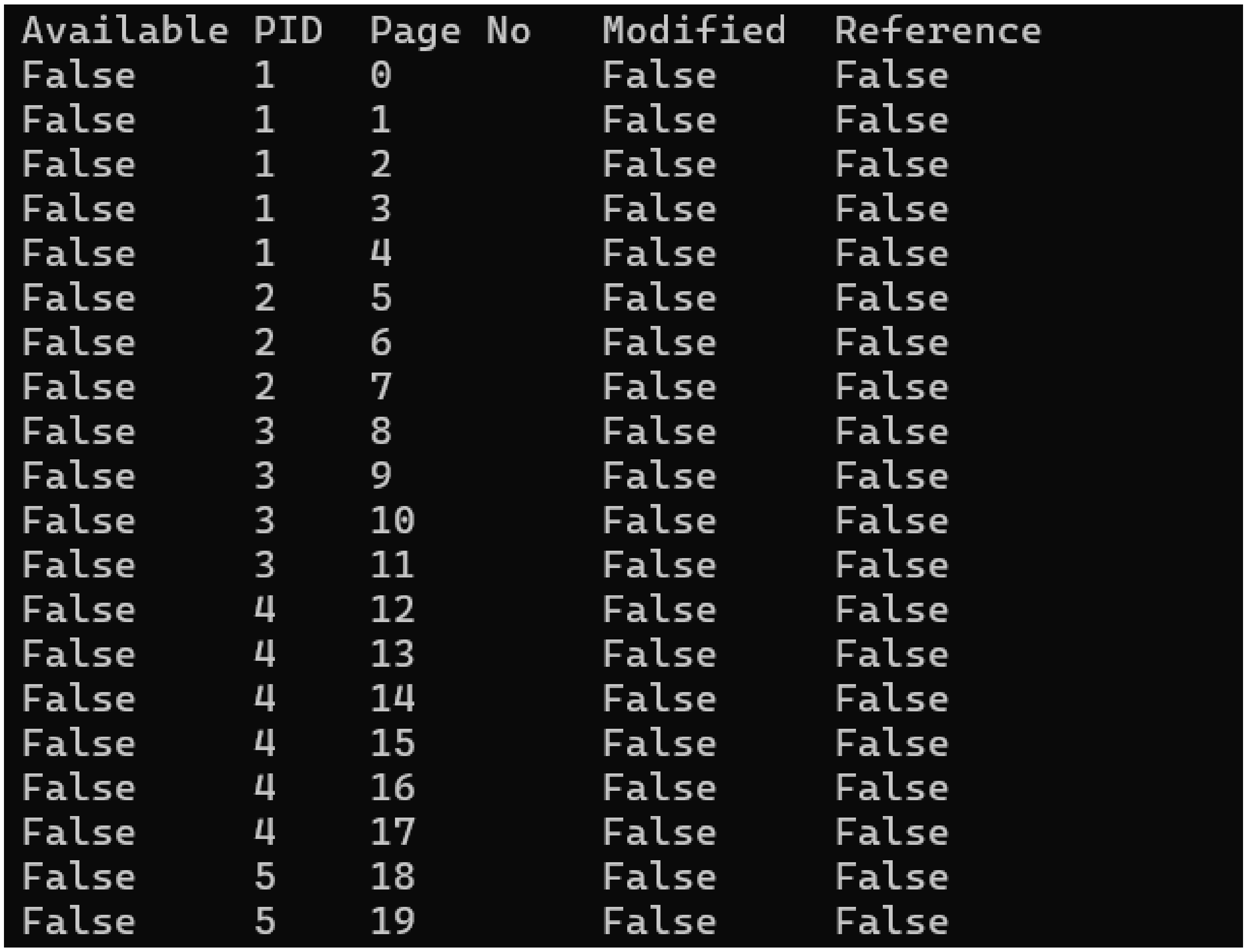
```
F:\UNIVERSITY\5TH SEMESTEI

Process Manager Menu
1. Load Processes from File
2. View Active Processes
3. Dispatch Menu
4. Add a new process
5. Kill a process
6. Suspend a process
7. View terminated processes
8. Resume a process
9. Display Inverted Page Table
10. Display available virtual memory
11. Display internal fragmentation
0. Exit
```



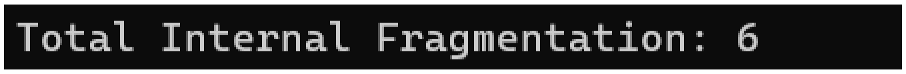
```
1
SIZE OF PROCESS IS: 15
Processes created!
SIZE OF PROCESS IS: 9
Processes created!
SIZE OF PROCESS IS: 10
Processes created!
```



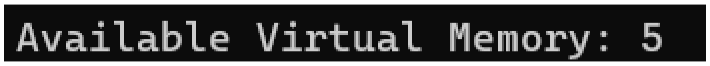
```
PID       PPID      CPU Bursts     Size Program Counter   State     Remaining CPU Bursts     Priority  Creation Time
1         0         100            15   0                 Ready     100                      1         22:28:44:14-05-2024
2         0         200            9    0                 Ready     200                      2         22:28:44:14-05-2024
3         0         300            10   0                 Ready     300                      3         22:28:44:14-05-2024
4         0         400            17   0                 Ready     400                      4         22:28:44:14-05-2024
5         0         500            14   0                 Ready     500                      5         22:28:44:14-05-2024
6         0         600            6    0                 Ready     600                      1         22:28:44:14-05-2024
7         0         700            7    0                 Ready     700                      2         22:28:44:14-05-2024
8         0         800            16   0                 Ready     800                      3         22:28:44:14-05-2024
9         0         900            13   0                 Ready     900                      4         22:28:44:14-05-2024
10        0         1000           5    0                 Ready     1000                     5         22:28:44:14-05-2024
11        0         1100           12   0                 Ready     1100                     1         22:28:44:14-05-2024
12        0         1200           20   0                 Ready     1200                     2         22:28:44:14-05-2024
13        0         1300           11   0                 Ready     1300                     3         22:28:44:14-05-2024
14        0         1400           18   0                 Ready     1400                     4         22:28:44:14-05-2024
15        0         1500           8    0                 Ready     1500                     5         22:28:44:14-05-2024
17        0         1700           5    0                 Ready     1700                     2         22:28:44:14-05-2024
19        0         1900           9    0                 Ready     1900                     4         22:28:44:14-05-2024
```



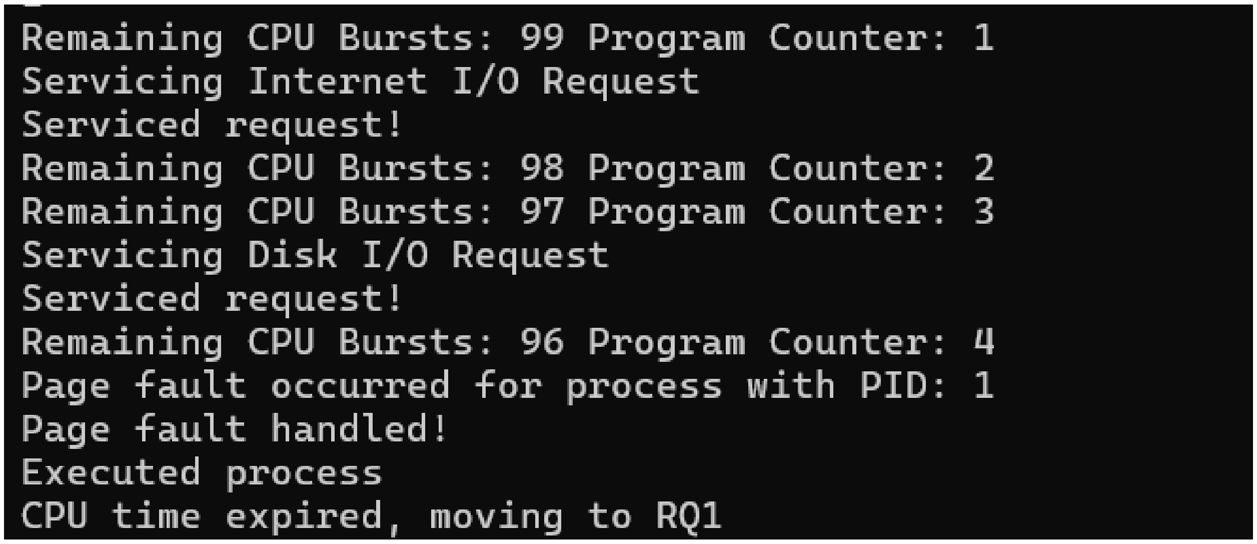
```
Available PID  Page No   Modified  Reference
False     1    0         False     False
False     1    1         False     False
False     1    2         False     False
False     1    3         False     False
False     1    4         False     False
False     2    5         False     False
False     2    6         False     False
False     2    7         False     False
False     3    8         False     False
False     3    9         False     False
False     3    10        False     False
False     3    11        False     False
False     4    12        False     False
False     4    13        False     False
False     4    14        False     False
False     4    15        False     False
False     4    16        False     False
False     4    17        False     False
False     5    18        False     False
False     5    19        False     False
```



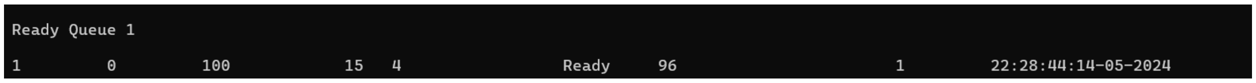
```
Total Internal Fragmentation: 6
```



```
Available Virtual Memory: 5
```



```
Remaining CPU Bursts: 99 Program Counter: 1
Servicing Internet I/O Request
Serviced request!
Remaining CPU Bursts: 98 Program Counter: 2
Remaining CPU Bursts: 97 Program Counter: 3
Servicing Disk I/O Request
Serviced request!
Remaining CPU Bursts: 96 Program Counter: 4
Page fault occurred for process with PID: 1
Page fault handled!
Executed process
CPU time expired, moving to RQ1
```



```
Ready Queue 1
1         0         100            15   4                Ready     96                       1         22:28:44:14-05-2024
```