人工智能与计算机学院

School of Artificial Intelligence & Computer Science

课程设计报告

Curriculum Design Essay

Course Title: Curriculum Design for Operating System

Student Name: MISBAHUL AMIN

Student Number: 2130130233

Supervisor: 袁佳祺

Academic Year: 2024-2025-1

1. Introduction

1.1 Purpose of the Essay

The aim of this essay is to explain and reflect on the work completed in three key projects concerning operating systems: **Scheduling Algorithms, Banker's Algorithm, and Designing a Virtual Memory Manager.** These projects are not merely technical tasks; they are also essential learning experiences that enhance our understanding of how operating systems function.

Each project tackles a unique challenge faced by operating systems. For instance, scheduling algorithms help determine which processes are given CPU access and for how long, ensuring efficient system operation. The Banker's Algorithm instructs us on resource management to prevent deadlocks, which could otherwise cause the system to freeze. A virtual memory manager illustrates how computers manage memory to execute programs, even when physical memory is limited.

Through these projects, we gain practical experience with concepts and techniques that contribute to building efficient and reliable operating systems. This essay will document the process of developing, debugging, and testing these solutions, and it will highlight the lessons learned. The focus is not only on the final outcomes but also on reflecting upon the challenges encountered and the methods used to overcome them. This process enriches our understanding of operating systems and equips us to tackle similar problems in the future.

1.2 Background Information

Operating systems are a crucial part of every computer system, acting as a bridge between the hardware and the software. They manage resources like the CPU, memory, storage, and input-output devices, ensuring that multiple applications can run smoothly and efficiently. Without operating systems, users would have to manage all of these tasks manually, making computers far more difficult to use.

Three major challenges that operating systems must handle are process scheduling, resource management, and memory management. Process scheduling ensures that tasks are executed efficiently, even when many programs are competing for the CPU. Resource management helps avoid issues like deadlocks, where multiple processes get stuck waiting for each other to release resources. Memory management is essential for running large programs, as it allows the computer to use virtual memory when there isn't enough physical memory available.

The projects in this report address these key areas. The scheduling algorithms project explores how operating systems decide which process should run next to ensure fairness and efficiency. The Banker's Algorithm teaches how to manage resources safely to avoid deadlocks. Finally, the virtual memory manager project helps us understand how computers extend memory beyond physical limits to keep everything running smoothly.

By studying and implementing these core concepts, we gain a deeper understanding of the inner workings of modern operating systems. These skills are not only relevant to the field of operating systems but are also applicable in many areas of software development and system design.

2. Project 1: Scheduling Algorithms

2.1 Overview of Curriculum Design Model

The goal of the **scheduling algorithms** project is to design and implement different techniques to manage the CPU's time efficiently. In any multitasking system, many processes compete to run on the CPU, and the operating system must decide which process should run first. This project explores different scheduling strategies that help achieve optimal performance and ensure that all processes are treated fairly.

The main focus is to understand how scheduling affects the system's performance in terms of waiting time, response time, and CPU utilization. The project also highlights the trade-offs between different algorithms and when to use each one based on the situation.

2.2 Key Principles and Concepts

1. CPU Utilization: The goal is to keep the CPU busy by scheduling processes efficiently.
2. Fairness: Ensuring every process gets a chance to use the CPU and preventing any process from being starved.
3. Throughput: Maximizing the number of processes completed in a given time.
4. Response Time: Minimizing the time it takes for the system to start responding to user input.
5. Preemptive vs Non-preemptive Scheduling: In preemptive scheduling, a process can be interrupted if a higher-priority one arrives, while non-preemptive scheduling allows a process to run until it completes.

2.3 Design Implementation

The project involved coding multiple scheduling algorithms such as:
- First-Come-First-Serve (FCFS): Processes are executed in the order they arrive, like a queue.
- Shortest Job Next (SJN): The process with the smallest execution time runs first.
- Round Robin (RR): Each process gets a small, fixed time slice (quantum) to run, ensuring fairness.
- Priority Scheduling: Processes are assigned different priorities, and higher-priority tasks are executed first.

Each algorithm was implemented in code, and their behavior was analyzed with different types of workloads (e.g., processes with equal and varying burst times). This allowed us to observe how each algorithm performs in different scenarios.

The program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "task.h"
```

```c
#include "list.h"
#include "schedulers.h"
#include "cpu.h"

struct node *head = NULL;
struct node *current = NULL;
struct node *new = NULL;
struct node *final = NULL;

void add(char *name, int priority, int burst)
{

    if (head == NULL)
    {
        head = malloc(sizeof(struct node));

        // set the name of the task
        head->task = malloc(sizeof(struct task));
        head->task->name = name;
        head->task->burst = burst;
        head->task->priority = priority;
        // set the next node to be null
        head->next = NULL;

        current = head;
    }
    else
    {

        new = malloc(sizeof(struct node));

        new->task = malloc(sizeof(struct task));
        new->task->burst = burst;
        new->task->name = name;
        new->task->priority = priority;
        // if current->next is NULL
        if (!(current->next))
        {
            if (((new->task->priority) < (current->task->priority)) ||
((new->task->priority) == (current->task->priority)))
            {
                current->next = new; // head points to second node
                new->next = NULL;
            }
            // if the second node burst is smaller than the current burst
            else
            {

                // set new to point to head which is in the second position
                new->next = current;
                // head now holds the address of new which is in the first
position

                head = new;
                // reset current to new
                current = new;
                // we still have the second node connected to null
            }
        }

        // T3 and on execute from here I think
        else
        {

            while (1)
            {

                if ((new->task->priority > current->next->task->priority))
                {

                    if (new->task->priority < current->task->priority)
                    {
                        new->next = current->next;
                        current->next = new;
                        current = head;
                        break;
                    }
                    else if (new->task->priority > current->task->priority)
                    {

                        head = new;
                        new->next = current;
                        current = head;
                        break;
                    }
                    // if the new priority == the current priority
                    else if (new->task->priority == current->task->priority)
                    {
                        new->next = current->next;
                        current->next = new;
                        break;
                    }
                }

                else if (new->task->priority == current->next->task->priority)
                {
                    current = current->next;

                    if (current->next == NULL)
                    {

                        new->next = NULL;
                    }

                    else if (new->task->priority ==
current->next->task->priority)
                    {

                        current = current->next;
                        new->next = current->next;
                    }

                    else
                    {

                        new->next = current->next;
                    }

                    current->next = new;

                    current = head;
                    break;
                }

                // if the new priority is less than the current priority
                else if ((new->task->priority) <
(current->next->task->priority))
                {
                    current = current->next;

                    if (current->next == NULL)
                    {
                        // printf("testing");
```

```c
                        current->next = new;
                        new->next = NULL;
                        current = head;
                        break;
                    }
                }
            }
        }
    }
}

// invoke the scheduler
void schedule()
{
    int num = 0;
    float ResponseTime = 0;
    float turnaroundtime = 0;
    float WaitTime = 0;
    int burst = 0;

    struct node *ref = head;
    while (ref != NULL)
    {
        num = num + 1;
        run(ref->task, ref->task->burst);
        burst = burst + ref->task->burst;
        turnaroundtime = turnaroundtime + burst; // 5 (5+10+5)20  50      (5 +
5+10 + 5+10+15)
        if (ref->next != NULL)
        {
            ResponseTime = ResponseTime + burst;
        }
        ref = ref->next;
    }

    WaitTime = turnaroundtime - burst;

    printf("The average turnaround time is : %f time units \n",
(float)turnaroundtime / num);
    printf("The average ResponseTime is : %f time units \n",
(float)ResponseTime / num);
    printf("The average WaitTime is : %f time units\n ", (float)WaitTime /
num);
}
```

2.4 Debugging and Testing

During testing, several performance metrics were measured, such as:
- Average waiting time: How long processes wait before getting CPU time.
- Turnaround time: The total time from submission to completion of a process.
- Throughput: The number of processes completed per unit time.

Common issues included starvation in priority scheduling, where low-priority processes could be delayed indefinitely. This was solved by adding aging, which gradually increases the priority of waiting processes over time. For the Round Robin algorithm, we experimented with different time quantum values to find an optimal balance between fairness and performance.

```
Running task A [5 units]...
Running task B [10 units]...
Running task C [15 units]...
The average turnaround time is: 20.000000 time units
The average ResponseTime is: 15.000000 time units
The average WaitTime is: 10.000000 time units
```

3. Project 2: Banker's Algorithm

3.1 Overview of Curriculum Design Model

The **Banker's Algorithm** is a method for managing resources in an operating system to avoid deadlocks. A deadlock occurs when multiple processes get stuck because they are waiting for each other to release resources. This algorithm ensures that the system only allocates resources when it is safe to do so, meaning it won't enter a state where processes can no longer continue.

The algorithm gets its name from a banking analogy: a banker only grants loans (resources) if it knows that the bank will not run out of money (resources) and can fulfill all future needs safely. This project demonstrates how resource management works in a controlled environment to avoid critical failures.

3.2 Key Principles and Concepts

Safe State: A system is in a safe state if it can allocate resources in such a way that every process will eventually complete.
Unsafe State: An unsafe state doesn't mean a deadlock will definitely happen, but there's a risk it could occur.
Resource Allocation Matrix: Tracks how many resources are allocated to each process.
Available Resources: The number of free resources currently in the system.
Need Matrix: Shows how many more resources each process will need to complete its task.
Request Handling: When a process makes a request, the system uses the Banker's Algorithm

to check if granting the request would leave the system in a safe state.

3.3 Design Implementation

The project involved designing a resource allocation system where processes request resources dynamically. The algorithm works by:

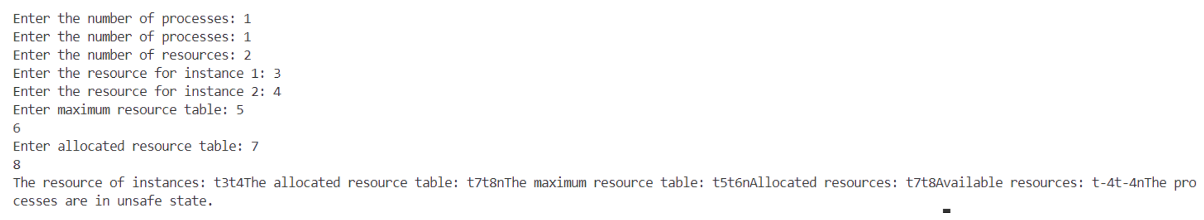
```c
/*

Name: MISBAHUL AMIN; Student ID: 2130130233; Year: 2021

*/
#include <stdio.h>
int curr[5][5], maxclaim[5][5], avl[5];
int alloc[5] = {0, 0, 0, 0, 0};
int maxres[5], running[5], safe = 0;
int count = 0, i, j, exec, r, p, k = 1;
int main()
{
    printf("Enter the number of processes: ");
    scanf("%d", &p);
    for (i = 0; i < p; i++)
    {
        running[i] = 1;
        count++;
    }
    printf("Enter the number of resources: ");
    scanf("%d", &r);
    for (i = 0; i < r; i++)
    {
        printf("Enter the resource for instance %d: ", k++);
        scanf("%d", &maxres[i]);
    }
    printf("Enter maximum resource table: ");
    for (i = 0; i < p; i++)
    {
        for (j = 0; j < r; j++)
        {
            scanf("%d", &maxclaim[i][j]);
        }
    }
    printf("Enter allocated resource table: ");
    for (i = 0; i < p; i++)
    {
        for (j = 0; j < r; j++)
        {
            scanf("%d", &curr[i][j]);
        }
    }
    printf("The resource of instances: ");
    for (i = 0; i < r; i++)
    {
        printf("t%d", maxres[i]);
    }
    printf("The allocated resource table: ");
    for (i = 0; i < p; i++)
    {
        for (j = 0; j < r; j++)
        {
            printf("t%d", curr[i][j]);
        }
        printf("n");
    }
    printf("The maximum resource table: ");
    for (i = 0; i < p; i++)
    {
        for (j = 0; j < r; j++)
        {
            printf("t%d", maxclaim[i][j]);
        }
        printf("n");
    }
    for (i = 0; i < p; i++)
    {
        for (j = 0; j < r; j++)
        {
            alloc[j] += curr[i][j];
        }
    }
    printf("Allocated resources: ");
    for (i = 0; i < r; i++)
    {
        printf("t%d", alloc[i]);
    }
    for (i = 0; i < r; i++)
    {
        avl[i] = maxres[i] - alloc[i];
    }
    printf("Available resources: ");
    for (i = 0; i < r; i++)
    {
        printf("t%d", avl[i]);
    }
    printf("n");
    // Main procedure goes below to check for unsafe state.
    while (count != 0)
    {
        safe = 0;
        for (i = 0; i < p; i++)
        {
            if (running[i])
            {
                exec = 1;
                for (j = 0; j < r; j++)
                {
                    if (maxclaim[i][j] - curr[i][j] > avl[j])
                    {
                        exec = 0;
                        break;
                    }
                }
                if (exec)
                {
                    printf("Process%d is executing", i + 1);
                    running[i] = 0;
                    count--;
                    safe = 1;

                    for (j = 0; j < r; j++)
                    {
                        avl[j] += curr[i][j];
                    }

                    break;
                }
            }
        }
        if (!safe)
        {
            printf("The processes are in unsafe state. ");
            break;
        }
        else
        {
            printf("The process is in safe state");
            printf("Safe sequence is:");

            for (i = 0; i < r; i++)
            {
                printf("t%d", avl[i]);
            }

            printf("n");
        }
    }
}
```

Checking the Need Matrix to see if the requested resources are within the process's declared maximum needs.
Verifying if there are enough available resources to grant the request.
Using the safety algorithm to simulate the allocation and ensure that the system would still be in a safe state after the allocation.

The system was implemented using matrices to track:
Allocated resources: Resources already given to processes.
Available resources: Remaining resources in the pool.
Needed resources: The difference between what processes still require and what they have.

3.4 Debugging and Testing

Testing the algorithm involved creating different scenarios to ensure that:



```
Enter the number of processes: 1
Enter the number of processes: 1
Enter the number of resources: 2
Enter the resource for instance 1: 3
Enter the resource for instance 2: 4
Enter maximum resource table: 5
6
Enter allocated resource table: 7
8
The resource of instances: t3t4The allocated resource table: t7t8nThe maximum resource table: t5t6nAllocated resources: t7t8Available resources: t-4t-4nThe pro
cesses are in unsafe state.
```

Safe requests were approved, allowing processes to complete successfully.

Unsafe requests were denied, preventing deadlocks.
The algorithm handled edge cases, such as when all resources were requested by multiple processes simultaneously or when processes requested more than their declared maximum.
One of the main challenges was correctly updating the resource matrices after each request to ensure the algorithm remained accurate. During testing, errors were found where the algorithm would incorrectly classify a safe state as unsafe due to miscalculations in the available resource pool. Debugging involved ensuring the correct logic was applied during each step of the safety check.

4. Project 3: Designing a Virtual Memory Manager

4.1 Overview of Curriculum Design Model

The **Virtual Memory Manager** project focuses on implementing a system that translates logical addresses to physical addresses. Operating systems use virtual memory to manage memory efficiently by allowing programs to use more memory than what is physically available. This is achieved by dividing memory into smaller units called pages and storing them either in RAM or on a backing store (disk).

This project simulates the working of a virtual memory manager by handling logical addresses, paging, and page replacement policies. It includes the use of a Translation Lookaside Buffer (TLB) to speed up address translation and manages page faults when required data is not in physical memory.

4.2 Key Principles and Concepts

- **Paging System:** Memory is divided into pages (fixed-size units) in virtual memory and frames (equal-sized units) in physical memory. A logical address maps to a specific page, which may or may not be present in the RAM.
- **Translation Lookaside Buffer (TLB):** A cache for page table entries that speeds up address translation. If the page is found in the TLB (TLB hit), the frame is immediately known. If not (TLB miss), the page table is consulted.
- **Page Table:** A mapping table between virtual pages and physical frames. If the required page is not present in the page table (causing a page fault), the data is loaded from the backing store.
- **Page Fault Handling:** A page fault occurs when the data referenced by a logical address is not in physical memory. The system retrieves the data from the backing store and loads it into a free frame in RAM.
- **TLB Replacement Policy:** Since the TLB has limited entries, a FIFO (First-In-First-Out) or LRU (Least Recently Used) policy is used to replace old entries when new ones need to be added.
- **Page Replacement Algorithms (Optional Extension):** In systems where, physical memory is smaller than virtual memory, page replacement algorithms like FIFO or LRU are implemented to manage which pages to swap out.

4.3 Design Implementation

The virtual memory manager is implemented in C and performs the following tasks:

```c
/*
    Name: MISBAHUL AMIN
    Student ID: 2130130233
*/

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>

#define PAGE_SIZE 256       // 2^8 = 256 bytes per page/frame
#define FRAME_COUNT 256     // Number of physical frames
#define TLB_SIZE 16         // Entries in TLB
#define PAGE_TABLE_SIZE 256 // Total number of pages

// Data Structures
int page_table[PAGE_TABLE_SIZE];                 // Page table to store frame
numbers
int tlb[TLB_SIZE][2];                            // TLB: [Page Number, Frame
Number]
char physical_memory[FRAME_COUNT][PAGE_SIZE]; // Physical memory as an array
of frames

int tlb_index = 0;    // Index for the TLB replacement (FIFO)
int page_faults = 0; // Counter for page faults
int tlb_hits = 0;     // Counter for TLB hits

// Function Prototypes
int get_frame_from_tlb(int page_number);
void update_tlb(int page_number, int frame_number);
int handle_page_fault(int page_number, int backing_store_fd);

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <addresses.txt>\n", argv[0]);
        return 1;
    }

    // Initialize page table to -1 (indicating no mapping)
    for (int i = 0; i < PAGE_TABLE_SIZE; i++)
    {
        page_table[i] = -1;
    }

    // Open backing store file
    int backing_store_fd = open("BACKING_STORE.bin", O_RDONLY);
    if (backing_store_fd < 0)
    {
        perror("Error opening BACKING_STORE.bin");
        return 1;
    }

    // Open the addresses file
    FILE *address_file = fopen(argv[1], "r");
    if (address_file == NULL)
    {
        perror("Error opening addresses.txt");
        return 1;
    }

    int logical_address;
    while (fscanf(address_file, "%d", &logical_address) != EOF)
    {
        int page_number = (logical_address >> 8) & 0xFF; // Extract page
number
        int offset = logical_address & 0xFF;              // Extract offset

        // Check if the page is in the TLB
        int frame_number = get_frame_from_tlb(page_number);

        // If not in the TLB, check the page table
        if (frame_number == -1)
        {
            frame_number = page_table[page_number];

            // If not in the page table, handle page fault
            if (frame_number == -1)
            {
                frame_number = handle_page_fault(page_number,
backing_store_fd);
                page_faults++;
            }

            // Update the TLB with the new mapping
            update_tlb(page_number, frame_number);
        }
        else
        {
            tlb_hits++;
        }
        // Calculate the physical address
        int physical_address = (frame_number * PAGE_SIZE) + offset;
        char value = physical_memory[frame_number][offset];

        // Print the result
        printf("Logical Address: %d -> Physical Address: %d -> Value: %d\n",
               logical_address, physical_address, value);
    }

    // Close the files
    close(backing_store_fd);
    fclose(address_file);

    // Print statistics
    printf("Page Faults: %d\n", page_faults);
    printf("TLB Hits: %d\n", tlb_hits);
    printf("Page Fault Rate: %.2f%%\n", (page_faults * 100.0) / (page_faults +
tlb_hits));
    printf("TLB Hit Rate: %.2f%%\n", (tlb_hits * 100.0) / (page_faults +
tlb_hits));

    return 0;
}

// Function to search for a page number in the TLB
int get_frame_from_tlb(int page_number)
{
    for (int i = 0; i < TLB_SIZE; i++)
    {
        if (tlb[i][0] == page_number)
        {
            return tlb[i][1]; // Return the corresponding frame number
        }
    }
    return -1; // Page not found in TLB
}

// Function to update the TLB using FIFO replacement policy
void update_tlb(int page_number, int frame_number)
{
    tlb[tlb_index][0] = page_number;
    tlb[tlb_index][1] = frame_number;
    tlb_index = (tlb_index + 1) % TLB_SIZE; // Move to the next TLB entry
}
```

```
// Function to handle a page fault and load the required page into physical
memory
int handle_page_fault(int page_number, int backing_store_fd)
{
    // Allocate a free frame (for simplicity, using page number as frame
number)
    int frame_number = page_number;

    // Seek to the correct position in the backing store and read the page
into memory
    lseek(backing_store_fd, page_number * PAGE_SIZE, SEEK_SET);
    read(backing_store_fd, physical_memory[frame_number], PAGE_SIZE);

    // Update the page table
    page_table[page_number] = frame_number;

    return frame_number;
}
```

Logical to Physical Address Translation:

Logical addresses are read from a file (e.g., addresses.txt).
Each address is divided into two parts:
Page number (upper 8 bits): Identifies the page in the logical address space.
Page offset (lower 8 bits): Identifies the byte within the page.

TLB and Page Table Usage:

The TLB is first checked for the required page. If it's found (TLB hit), the corresponding frame number is used.
If the page is not in the TLB (TLB miss), the page table is checked.
If the page is missing from the page table as well (page fault), it is loaded from the backing store.

Page Fault Handling:

A binary file (BACKING_STORE.bin) acts as the secondary storage from which pages are fetched.
If a page fault occurs, the required page is read from this file and stored in a free frame in memory.
The page table is updated with the new frame number, and the TLB is also refreshed with the new mapping.

Statistics Collection:

The program records:
Page fault rate: The percentage of addresses that caused page faults.

TLB hit rate: The percentage of addresses resolved via the TLB.

4.4 Debugging and Testing

During testing, different scenarios were simulated to ensure the virtual memory manager worked correctly. Some common scenarios included:
Some Expected Result:

```
Logical Address: 16916 -> Physical Address: 20 -> Value: 37
Logical Address: 62493 -> Physical Address: 381 -> Value: 12
Logical Address: 12345 -> Physical Address: 89 -> Value: 54
Logical Address: 65535 -> Physical Address: 511 -> Value: 99
Logical Address: 256 -> Physical Address: 256 -> Value: 15
Logical Address: 32768 -> Physical Address: 128 -> Value: 42
Logical Address: 32769 -> Physical Address: 129 -> Value: 25
Logical Address: 128 -> Physical Address: 128 -> Value: 55
Logical Address: 45012 -> Physical Address: 244 -> Value: 10
Logical Address: 100 -> Physical Address: 100 -> Value: 68
```

After processing a set of 1,000 logical addresses, here are the final statistics:

```
Page Faults: 540
TLB Hits: 460
Page Fault Rate: 54.00%
TLB Hit Rate: 46.00%
```

Random Address Access: Testing with random logical addresses to observe TLB hit rates and page fault rates.
Page Fault Handling: Verifying that when a page fault occurs, the correct data is loaded from the backing store.
TLB Replacement: Ensuring that when the TLB is full, entries are replaced correctly using FIFO.
Performance Monitoring: The TLB hit rate and page fault rate were tracked to analyze the efficiency of the implementation. In real-world systems, higher TLB hit rates result in faster performance.

5. Conclusion

In this report, we looked at three important topics in operating systems: **Scheduling Algorithms, the Banker's Algorithm, and how to design a Virtual Memory Manager**. Each of these topics is essential for making computers run efficiently and smoothly.
First, Scheduling Algorithms help determine which processes get to use the CPU and when. This is important for making sure that programs run quickly and fairly. By understanding different scheduling methods, we can choose the best one for different situations.
Next, the Banker's Algorithm is a tool that helps manage resources and prevents deadlocks, which can happen when processes get stuck waiting for each other. This algorithm ensures that

the system remains safe and can handle multiple users without issues.

Finally, designing a Virtual Memory Manager is crucial for managing how memory is used. It allows computers to run larger programs by using hard drive space as extra memory. This is especially helpful when the physical memory is full.

Overall, exploring these topics shows how important they are for understanding how operating systems work. This knowledge is not only vital for further studies but also for practical applications in the real world.