Netaji Subhas University of Technology

LAB REPORT

OPERATING SYSTEMS

Name **Kushagra Lakhwani**
Roll No. ***2021UCI8036***
Semester **4th**
Course ***CICPC09***

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

April 16, 2023

Abstract

The practical lab report *"Operating Systems"* is the original and unmodified content submitted by *Kushagra Lakhwani* (Roll No. 2021UCI8036).

The report is submitted to *Dr. Manoj Kumar* Department of Computer Science and Engineering, NSUT, Delhi, for the partial fulfillment of the requirements of the course (CICPC09).

Index

1 Process Creation and Termination

1.1 Process Creation

1.1.1 The `fork()` System Call

A process is created by the `fork()` system call. The `fork()` system call creates a child process that is an exact copy of the parent process.

```c
/**
 * Forks a new process and checks for parent and child processes
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  pid_t pid;
  pid = fork();

  if (pid == 0) {
    printf("Child process pid: %d\n", (int)getpid());
    exit(EXIT_SUCCESS);
  } else if (pid > 0) {
    printf("Parent process pid: %d", (int)getpid());
  } else {
    printf("Fork failed");
    exit(EXIT_FAILURE);
  }

  return 0;
}
```

The `fork()` system call returns twice. The first time it returns the process ID of the child process to the parent process. The second time it returns 0 to the child process.

```
Parent process pid: 96732
Child process pid: 96733
```

1.2 Process Termination

1.2.1 The exit() System Call

The exit() system call terminates the calling process. The exit() system call takes an integer argument that is returned to the parent process as the child's exit status.

```c
// In file <unistd.h>
/* Terminate program execution with the low-order 8 bits of STATUS.  */
extern void _exit(int __status) __attribute__((__noreturn__));
```

2 CPU Scheduling: FCFS

Implementation of CPU scheduling in a first come first serve (FCFS) manner.

2.1 Implementation

```c
/**
 * Implementation of FCFS scheduling algorithm
 */

#include <stdio.h>

void find_waiting_time(int processes[], int n, int bt[], int wt[]) {
  wt[0] = 0;
  for (int i = 1; i < n; i++) {
    wt[i] = bt[i - 1] + wt[i - 1];
  }
}

void find_turn_around_time(int processes[], int n, int bt[], int wt[],
                           int tat[]) {
  for (int i = 0; i < n; i++)
    tat[i] = bt[i] + wt[i];
}

void find_avg_time(int processes[], int n, int bt[]) {
  int wt[n], tat[n], total_wt = 0, total_tat = 0;
  find_waiting_time(processes, n, bt, wt);
  find_turn_around_time(processes, n, bt, wt, tat);
  printf("Processes  Burst Time   Waiting Time Turn Around Time\n");
  for (int i = 0; i < n; i++) {
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    printf("%d\t   ", i + 1);
    printf("%d\t\t", bt[i]);
    printf("%d\t\t", wt[i]);
    printf("%d\t\n", tat[i]);
  }
  printf("Average waiting time = %f\n", (float)total_wt / (float)n);
  printf("Average turn around time = %f", (float)total_tat / (float)n);
}

int main() {
  int processes[] = {1, 2, 3};
  int n = sizeof processes / sizeof processes[0];
  int burst_time[] = {10, 5, 8};
  find_avg_time(processes, n, burst_time);
  return 0;
}
```

The scheduler selects the process that has been waiting the longest.

```
Processes  Burst Time    Waiting Time     Turn Around Time
1          10            0                10
2          5             10               15
3          8             15               23
Average waiting time = 8.333333
Average turn around time = 16.000000
```

3 CPU Scheduling: Priority

Implementation of CPU scheduling using a "priority" based approach using assigned ranks/priority algorithms.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

3.1 Implementation

```
/**
 * CPU priority scheduler.
 */

#include <stdio.h>

typedef struct Process {
  int pid;
  int priority;
  int burst;
} Process;

int compare(Process a, Process b) { return a.priority < b.priority; }

void sort(Process *p, int n) {
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      if (compare(p[j], p[i])) {
        Process temp = p[i];
        p[i] = p[j];
        p[j] = temp;
      }
}

void findWaitingTime(Process proc[], int n, int wt[]) {
  wt[0] = 0;
  for (int i = 1; i < n; i++)
    wt[i] = proc[i - 1].burst + wt[i - 1];
}
```

```
void findTurnAroundTime(Process proc[], int n, int wt[], int tat[]) {
  for (int i = 0; i < n; i++)
    tat[i] = proc[i].burst + wt[i];
}

void findavgTime(Process proc[], int n) {
  int wt[n], tat[n], total_wt = 0, total_tat = 0;

  findWaitingTime(proc, n, wt);
  findTurnAroundTime(proc, n, wt, tat);

  printf("Processes Burst time Waiting time Turn around time\n");
  for (int i = 0; i < n; i++) {
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    printf("%d\t\t", proc[i].pid);
    printf("%d\t\t", proc[i].burst);
    printf("%d\t\t", wt[i]);
    printf("%d", tat[i]);
    printf("\n");
  }

  printf("Average waiting time = %f", (float)total_wt / (float)n);
  printf("Average turn around time = %f", (float)total_tat / (float)n);
}

void priorityScheduling(Process proc[], int n) {
  sort(proc, n);
  printf("Order in which processes gets executed\n");
  for (int i = 0; i < n; i++)
    printf("%d ", proc[i].pid);
  printf("\n");
  findavgTime(proc, n);
}

int main() {
  Process proc[] = {{1, 2, 100}, {2, 1, 19}, {3, 1, 27}, {4, 1, 25}};
  int n = sizeof proc / sizeof proc[0];
  priorityScheduling(proc, n);
  return 0;
}
```

```
Order in which processes gets executed
2 3 4 1
Process          BurstTime        WaitingTime      TurnAroundTime
2                19               0                19
3                27               19               46
4                25               46               71
1                100              71               171
Average waiting time = 34.000000
Average turn around time = 76.750000
```

4 CPU Scheduling: SJF

Implementation of CPU scheduling using the shortest job first (SJF) approach.

4.1 Algorithm

Algorithm 1 Shortest Job First

Input: n processes with their burst times bt_i and arrival times at_i
Output: The order in which the processes are executed
$t \leftarrow 0$ ▷ current time
$i \leftarrow 0$ ▷ current process
$bt \leftarrow \{bt_1, \ldots, bt_n\}$ ▷ burst times
$at \leftarrow \{at_1, \ldots, at_n\}$ ▷ arrival times
$bt' \leftarrow \{bt_1, \ldots, bt_n\}$ ▷ remaining burst times
$wt \leftarrow \{0, \ldots, 0\}$ ▷ waiting times
$tat \leftarrow \{0, \ldots, 0\}$ ▷ turnaround times
while $i < n$ **do**
    if $at_i \leq t$ and $bt'_i > 0$ **then** ▷ process is ready
        $bt'_i \leftarrow bt'_i - 1$
        $t \leftarrow t + 1$
        if $bt'_i = 0$ **then**
            $tat_i \leftarrow t - at_i$
            $wt_i \leftarrow tat_i - bt_i$
            $i \leftarrow i + 1$
        end if
    else
        $t \leftarrow t + 1$
    end if
end while
Return: wt and tat

4.2 Implementation

```c
/**
 * CPU Scheduling - Shortest Job First (SJF)
 */

#include <stdio.h>

void swap(int *x, int *y) {
  int temp;
  temp = *x;
  *x = *y;
```

```
  *y = temp;
}

int main() {
  int n, i, j, temp, total = 0, pos, avg_wait, avg_turnaround;
  int bt[20], p[20], wt[20], tat[20];

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

  printf("Enter Burst Time:\n");
  for (i = 0; i < n; i++) {
    printf("P[%d]: ", i + 1);
    scanf("%d", &bt[i]);
    p[i] = i + 1; // contains process number
  }

  // sorting burst time in ascending order using selection sort
  for (i = 0; i < n; i++) {
    pos = i;
    for (j = i + 1; j < n; j++) {
      if (bt[j] < bt[pos])
        pos = j;
    }

    swap(&bt[i], &bt[pos]);
    swap(&p[i], &p[pos]);
  }

  wt[0] = 0; // waiting time for first process is zero

  // calculate waiting time
  for (i = 1; i < n; i++) {
    wt[i] = 0;
    for (j = 0; j < i; j++)
      wt[i] += bt[j];

    total += wt[i];
  }

  avg_wait = total / n; // average waiting time

  printf("Process\t\tBurst Time\tWaiting Time\tTurnaround Time\n");
  total = 0;
  for (i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i]; // calculate turnaround time
    total += tat[i];
    printf("%d\t\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
  }

  avg_turnaround = total / n; // average turnaround time
```

```
  printf("Average Waiting Time: %d\n", avg_wait);
  printf("Average Turnaround Time: %d", avg_turnaround);

  return 0;
}
```

```
Enter number of process: 5
Enter Burst Time:
P[1]: 1
P[2]: 2
P[3]: 3
P[4]: 4
P[5]: 5
Process          BurstTime        WaitingTime      TurnAroundTime
1                1                0                1
2                2                1                3
3                3                3                6
4                4                6                10
5                5                10               15
Average Waiting Time: 4
Average Turnaround Time: 7
```

5 Producer-Consumer Problem

Implementation of the producer-consumer problem using semaphores.

5.1 Problem Statement

A producer-consumer problem is a classic synchronization problem.

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

- The producer is not allowed to add data into the buffer if it's full.
- Data can only be consumed by the consumer if the memory buffer is not empty.
- Accessing the buffer is mutually exclusive.

5.2 Algorithm

Algorithm 2 Producer-Consumer Problem

Input: n producers and m consumers
Output: The order in which the producers and consumers are executed
$i \leftarrow 0$ ▷ current producer
$j \leftarrow 0$ ▷ current consumer
$p \leftarrow \{p_1, \ldots, p_n\}$ ▷ producers
$c \leftarrow \{c_1, \ldots, c_m\}$ ▷ consumers
$buffer \leftarrow \emptyset$ ▷ buffer
$mutex \leftarrow 1$ ▷ mutex
$empty \leftarrow n$ ▷ empty slots
$full \leftarrow 0$ ▷ occupied slots
while $i < n$ or $j < m$ **do**
    if $i < n$ and $empty > 0$ **then**
        p_i ▷ produce
        $i \leftarrow i + 1$
    end if
    if $j < m$ and $full > 0$ **then**
        c_j ▷ consume
        $j \leftarrow j + 1$
    end if
end while
Return: $buffer$

5.3 Implementation

```c
/**
 * Producer-Consumer Problem
 */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 3

int buffer[BUFFER_SIZE];
int mutex = 1;
int full = 0;

int empty = BUFFER_SIZE;

int in = 0;
```

```
int out = 0;

void *producer(void *pno) {
  int item;
  for (int i = 0; i < 3; i++) {
    item = rand(); // Produce an random item
    while (mutex <= 0)
      ; // Do nothing
    mutex--;
    full++;
    empty--;
    buffer[in] = item;
    printf("Producer %d: Insert Item %d at %d\n", *((int *)pno), buffer[in], in);
    in = (in + 1) % BUFFER_SIZE;
    mutex++;
  }
  return NULL;
}

void *consumer(void *cno) {
  for (int i = 0; i < 3; i++) {
    while (mutex <= 0)
      ; // Do nothing
    mutex--;
    full--;
    empty++;
    int item = buffer[out];
    printf("Consumer %d: Remove Item %d from %d\n", *((int *)cno), item, out);
    out = (out + 1) % BUFFER_SIZE;
    mutex++;
  }
  return NULL;
}

int main() {
  pthread_t pro[3], con[3];
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  // Just used for numbering the producer and consumer
  int a[3] = {1, 2, 3}; // A vector of item

  // Create the producer threads
  for (int i = 0; i < 3; i++)
    pthread_create(&pro[i], &attr, producer, &a[i]);

  // Create the consumer threads
  for (int i = 0; i < 3; i++)
    pthread_create(&con[i], &attr, consumer, &a[i]);
```

```
  for (int i = 0; i < 3; i++)
    // Wait for the producer thread to exit
    pthread_join(pro[i], NULL);

  for (int i = 0; i < 3; i++)
    // Wait for the consumer thread to exit
    pthread_join(con[i], NULL);

  return 0;
}
```

```
Producer 1: Insert Item 1804289383 at 0
Consumer 1: Remove Item 1804289383 from 0
Producer 2: Insert Item 846930886 at 1
Consumer 1: Remove Item 1681692777 from 1
Consumer 1: Remove Item 1714636915 from 2
Producer 3: Insert Item 1681692777 at 1
Producer 3: Insert Item 424238335 at 0
Producer 3: Insert Item 719885386 at 1
Producer 2: Insert Item 1957747793 at 2
Producer 2: Insert Item 1649760492 at 0
Producer 1: Insert Item 1714636915 at 2
Producer 1: Insert Item 596516649 at 2
Consumer 2: Remove Item 1649760492 from 0
Consumer 2: Remove Item 719885386 from 1
Consumer 2: Remove Item 596516649 from 2
Consumer 3: Remove Item 1649760492 from 0
Consumer 3: Remove Item 719885386 from 1
Consumer 3: Remove Item 596516649 from 2
```

6 Bankers' Algorithm

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue. It was proposed by Edsger Dijkstra in 1965 and published in 1968.

6.1 Problem Statement

The following is a description of the resources used in the Banker's Algorithm.

- **Resource Type** - A resource type is a type of resource that can be allocated to a process. For example, a computer system may have three resource types: CPU, I/O, and Memory

- **Resource** - A resource is an instance of a resource type. For example, a computer system may have 10 CPU resources, 5 I/O resources, and 100 Memory resources.
- **Resource Vector** - A resource vector is a vector of resources. For example, a computer system may have a resource vector of 10 CPU resources, 5 I/O resources, and 100 Memory resources.
- **Maximum Claim** - The maximum claim of a process is the maximum number of resources of each type that the process may request. For example, a process may have a maximum claim of 10 CPU resources, 5 I/O resources, and 100 Memory resources.
- **Allocation** - The allocation of a process is the number of resources of each type that have been allocated to the process. For example, a process may have an allocation of 2 CPU resources, 1 I/O resource, and 50 Memory resources.
- **Need** - The need of a process is the number of resources of each type that the process still needs. For example, a process may have a need of 8 CPU resources, 4 I/O resources, and 50 Memory resources.

6.2 Algorithm

6.2.1 Initialization

The Banker's Algorithm is initialized with the following:

- A resource vector **available** that contains the number of available resources of each type.
- A maximum claim matrix **max** that contains the maximum number of resources of each type that each process may request.
- An allocation matrix **allocation** that contains the number of resources of each type that have been allocated to each process.

The need matrix **need** is then calculated as follows:

$$\mathbf{need} = \mathbf{max} - \mathbf{allocation} \tag{1}$$

6.2.2 Safe State

Algorithm 3 Safe State

Input: available, **max**, **allocation**
Output: true if **available** can satisfy **need**, **false** otherwise
Initialize: work = **available**, **finish** = **false**
while there exists a process **p** such that **finish[p]** is **false do**
    if need[p] $\leq$ **work then**
        work = **work** + **allocation[p]**
        finish[p] = **true**
    end if
end while
Return: true if **finish** is **true** for all processes, **false** otherwise

6.2.3 Request Resources

The request resources algorithm is as follows:

Algorithm 4 Request Resources

Require: A process P and a resource vector *request*
Ensure: The request is either granted or denied
$need = max - allocation$
if $request > need$ **then**
    Deny request
else
    if $request > available$ **then**
        Deny request
    else
        $available = available - request$
        $allocation = allocation + request$
        $need = need - request$
        if Safe State **then**
            Grant request
        else
            Deny request
        end if
    end if
end if

6.3 Implementation

```c
/**
 * @file bankers.c
 * @brief Banker's Algorithm
 * */

#include <stdio.h>
#include <stdlib.h>

int main() {
  int n = 5,                    // Number of processes
      m = 3;                    // Number of resources
  int alloc[5][3] = {{0, 1, 0},  // P0    // Allocation Matrix
                     {2, 0, 0},  // P1
                     {3, 0, 2},  // P2
                     {2, 1, 1},  // P3
                     {0, 0, 2}}; // P4

  int max[5][3] = {{7, 5, 3},  // P0    // MAX Matrix
                   {3, 2, 2},  // P1
                   {9, 0, 2},  // P2
                   {2, 2, 2},  // P3
                   {4, 3, 3}}; // P4

  int avail[3] = {3, 3, 2}; // Available Resources

  int f[n], ans[n], ind = 0;
  for (int k = 0; k < n; k++)
    f[k] = 0;

  int need[n][m];
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      need[i][j] = max[i][j] - alloc[i][j];

  int y = 0;
  for (int k = 0; k < 5; k++) {
    for (int i = 0; i < n; i++) {
      if (f[i] == 0) {
        int flag = 0;
        for (int j = 0; j < m; j++) {
          if (need[i][j] > avail[j]) {
            flag = 1;
            break;
          }
        }

        if (flag == 0) {
          ans[ind++] = i;
          for (y = 0; y < m; y++)
            avail[y] += alloc[i][y];
```

```
          f[i] = 1;
        }
      }
    }
  }

  printf("Following is the SAFE Sequence\n");
  for (int i = 0; i < n - 1; i++)
    printf(" P%d ->", ans[i]);
  printf(" P%d", ans[n - 1]);

  return 0;
}
```

```
Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2
```

7 Page Replacement Policies: FIFO

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

7.1 Algorithm

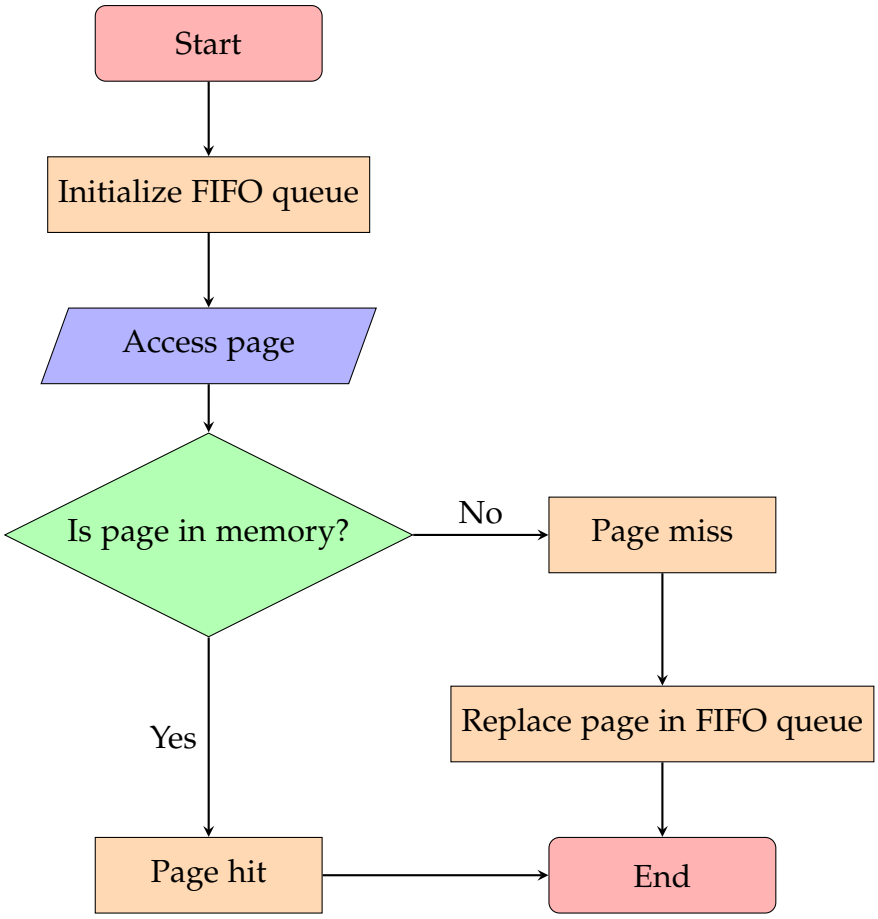

Figure 1: Page Replacement Policies: FIFO

7.2 Implementation

```c
#include <stdio.h>
#include <stdlib.h>

// Define the page table as a circular buffer using an array
int *page_table;
int page_table_size;
int page_table_capacity;
```

```c
int head;

void initialize_page_table() {
  page_table = (int *)malloc(page_table_capacity * sizeof(int));
  for (int i = 0; i < page_table_capacity; i++)
    page_table[i] = -1; // -1 means the page is empty
  head = 0;
}

int is_page_in_table(int page) {
  for (int i = 0; i < page_table_capacity; i++)
    if (page_table[i] == page)
      return 1;
  return 0;
}

void add_page_to_table(int page) {
  page_table[head] = page;
  head = (head + 1) % page_table_capacity;
}

// Function to simulate the FIFO page replacement algorithm
int simulate_fifo(int *pages, int num_pages) {
  int page_faults = 0;
  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    if (!is_page_in_table(page)) {
      add_page_to_table(page);
      page_faults++;
    }
  }
  return page_faults;
}

int main() {
  int num_pages, page_capacity;
  printf("Enter the number of pages: ");
  scanf("%d", &num_pages);
  printf("Enter the capacity of the page table: ");
  scanf("%d", &page_capacity);
  page_table_size = 0;
  page_table_capacity = page_capacity;
  initialize_page_table();
  int *pages = (int *)malloc(num_pages * sizeof(int));
  printf("Enter the page references:\n");
  for (int i = 0; i < num_pages; i++)
    scanf("%d", &pages[i]);
  int page_faults = simulate_fifo(pages, num_pages);
  printf("Number of page faults: %d\n", page_faults);
  free(page_table);
  free(pages);
```

```
  return 0;
}
```

```
Enter the number of pages: 4
Enter the capacity of the page table: 3
Enter the page references:
9 5 2 2
Number of page faults: 3
```

7.2.1 Analysis

- **Hit Ratio:** $H = \frac{N_{hits}}{N_{hits}+N_{misses}} = \frac{N_{hits}}{N_{hits}+N_{pages}-N_{hits}} = \frac{N_{hits}}{N_{pages}}$
- **Miss Ratio:** $M = \frac{N_{misses}}{N_{hits}+N_{misses}} = \frac{N_{misses}}{N_{pages}}$

8 Page Replacement Policies: LRU

The Least Recently Used (LRU) page replacement algorithm keeps track of the last time a page was accessed. When a page needs to be replaced, the page that was accessed the longest time ago is selected for removal.

8.1 Implementation

```
#include <stdio.h>
#include <stdlib.h>

int *page_table;
int *page_table_last_use;
int page_table_size;
int page_table_capacity;

void initialize_page_table() {
  page_table = (int *)malloc(page_table_capacity * sizeof(int));
  page_table_last_use = (int *)malloc(page_table_capacity * sizeof(int));
  for (int i = 0; i < page_table_capacity; i++) {
    page_table[i] = -1;          // -1 means the page is empty
    page_table_last_use[i] = -1; // -1 means the page has not been used yet
  }
}

int is_page_in_table(int page) {
  for (int i = 0; i < page_table_capacity; i++)
    if (page_table[i] == page)
      return 1;
  return 0;
}

int find_lru_page() {
```

```
  int lru_page = -1, lru_time = -1;
  for (int i = 0; i < page_table_capacity; i++) {
    if (page_table_last_use[i] < lru_time || lru_time == -1) {
      lru_page = i;
      lru_time = page_table_last_use[i];
    }
  }
  return lru_page;
}

void add_page_to_table(int page, int time) {
  int lru_page = find_lru_page();
  page_table[lru_page] = page;
  page_table_last_use[lru_page] = time;
}

// Simulate the LRU page replacement algorithm
int simulate_lru(int *pages, int num_pages) {
  int page_faults = 0, time = 0;
  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    if (!is_page_in_table(page)) {
      add_page_to_table(page, time);
      page_faults++;
    } else
      for (int j = 0; j < page_table_capacity; j++)
        if (page_table[j] == page) {
          page_table_last_use[j] = time;
          break;
        }

    time++;
  }
  return page_faults;
}
```

```
Enter the number of pages: 9
Enter the capacity of the page table: 3
Enter the page references:
9 5 2 2 5 9 2 5 9
Number of page faults: 8
```

9 Disk Scheduling: FCFS

Implementation of disk scheduling in a first come first serve (FCFS) manner.

9.1 Implementation

```c
/**
 * @file disk_fcfs.c
 * @brief Disk scheduling algorithm FCFS
 */

#include <stdio.h>
#include <stdlib.h>

void FCFS(int arr[], int head, int size) {
  int seek_count = 0;
  int cur_track, distance;

  for (int i = 0; i < size; i++) {
    cur_track = arr[i];
    distance = abs(head - cur_track); // calculate absolute distance
    seek_count += distance;           // increase the total count
    head = cur_track;                 // accessed track is now new head
  }
  printf("Total number of seek operations: %d\n", seek_count);
  printf("Seek Sequence is ");       // Seek sequence would be the same as request array sequence
  for (int i = 0; i < size; i++)
    printf("-> %d ", arr[i]);
}

int main() {
  int size;
  printf("Enter the size of the queue: ");
  scanf("%d", &size);
  int arr[size];
  printf("Enter the queue: ");
  for (int i = 0; i < size; i++)
    scanf("%d", &arr[i]);
  int head;
  printf("Enter the head: ");
  scanf("%d", &head);
  FCFS(arr, head, size);
  return 0;
}
```

```
Enter the size of the queue: 8
Enter the queue: 98 183 37 122 14 124 65 67
Enter the head: 460
Total number of seek operations: 957
Seek Sequence is -> 98 -> 183 -> 37 -> 122 -> 14 -> 124 -> 65 -> 67
```

10 Disk Scheduling: SSTF

Implementation of disk scheduling using the shortest seek time first (SSTF) approach.

10.1 Implementation

```c
/**
* @file disk_sstf.c
* @brief Implementation of the SSTF disk scheduling algorithm.
*/

#include <stdio.h>
#include <stdlib.h>

int disk_sstf(disk_request_t *requests, size_t num_requests, size_t initial_head_pos, size_t *head_m
{
    if (requests == NULL || num_requests == 0 || head_movement == NULL)
        return -1;

    // Copy the requests array so that we can modify it
    disk_request_t *requests_copy = (disk_request_t *)malloc(sizeof(disk_request_t) * num_requests);
    for (size_t i = 0; i < num_requests; i++)
        requests_copy[i] = requests[i];
    size_t head_pos = initial_head_pos; // The head starts at the initial position
    *head_movement = 0;

    // Loop through all the requests
    for (size_t i = 0; i < num_requests; i++)
    {
        size_t closest_request = 0; // Find the closest request
        size_t min_distance = abs(requests_copy[0].track - head_pos);
        for (size_t j = 1; j < num_requests; j++)
        {
            size_t distance = abs(requests_copy[j].track - head_pos);
            if (distance < min_distance)
            {
                closest_request = j;
                min_distance = distance;
            }
        }

        // Process the closest request
        *head_movement += min_distance;
        head_pos = requests_copy[closest_request].track;

        // Remove the request from the array
        for (size_t j = closest_request; j < num_requests - 1; j++)
            requests_copy[j] = requests_copy[j + 1];
    }

    free(requests_copy);
```

```
    return 0;
}
```

```
Enter the size of the queue: 8
Enter the queue: 98 183 37 122 14 124 65 67
Enter the head: 53
Total number of seek operations: 236
Seek Sequence is -> 65 -> 67 -> 37 -> 14 -> 53 -> 98 -> 122 -> 124 ->
    183
```