



**Netaji Subhas University
of Technology**

LAB REPORT

OPERATING SYSTEMS

| | |
|----------|--------------------------|
| Name | Kushagra Lakhwani |
| Roll No | 2021UCI8036 |
| Semester | 4th |
| Course | CICPC09 |

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

February 24, 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).

Contents

| | | |
|----------|---|----------|
| 1 | Process Creation and Termination | 3 |
| 1.1 | Process Creation | 3 |
| 1.1.1 | The fork() System Call | 3 |
| 1.2 | Process Termination | 3 |
| 1.2.1 | The exit() System Call | 3 |
| 2 | CPU Scheduling: FCFS | 4 |
| 3 | CPU Scheduling: Priority | 5 |
| 4 | CPU Scheduling: SJF | 7 |
| 4.1 | Algorithm | 7 |
| 4.2 | Implementation | 7 |
| 5 | Producer-Consumer Problem | 9 |
| 5.1 | Problem Statement | 9 |
| 5.2 | Algorithm | 10 |
| 5.3 | Implementation | 10 |

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.

```
/**
 * 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.

```
// 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.

```

/**
 * 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.

```
/**
 * 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

```

1: Input:  $n$  processes with their burst times  $bt_i$  and arrival times  $at_i$ 
2: Output: The order in which the processes are executed
3:  $t \leftarrow 0$  ▷ current time
4:  $i \leftarrow 0$  ▷ current process
5:  $bt \leftarrow \{bt_1, \dots, bt_n\}$  ▷ burst times
6:  $at \leftarrow \{at_1, \dots, at_n\}$  ▷ arrival times
7:  $bt' \leftarrow \{bt_1, \dots, bt_n\}$  ▷ remaining burst times
8:  $wt \leftarrow \{0, \dots, 0\}$  ▷ waiting times
9:  $tat \leftarrow \{0, \dots, 0\}$  ▷ turnaround times
10: while  $i < n$  do
11:   if  $at_i \leq t$  and  $bt'_i > 0$  then ▷ process is ready
12:      $bt'_i \leftarrow bt'_i - 1$ 
13:      $t \leftarrow t + 1$ 
14:     if  $bt'_i = 0$  then
15:        $tat_i \leftarrow t - at_i$ 
16:        $wt_i \leftarrow tat_i - bt_i$ 
17:        $i \leftarrow i + 1$ 
18:     end if
19:   else
20:      $t \leftarrow t + 1$ 
21:   end if
22: end while
23: Return:  $wt$  and  $tat$ 

```

4.2 Implementation

```

/**
 * 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\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%d\t%d\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

```

1: Input:  $n$  producers and  $m$  consumers
2: Output: The order in which the producers and consumers are executed
3:  $i \leftarrow 0$                                 ▷ current producer
4:  $j \leftarrow 0$                                 ▷ current consumer
5:  $p \leftarrow \{p_1, \dots, p_n\}$                 ▷ producers
6:  $c \leftarrow \{c_1, \dots, c_m\}$                 ▷ consumers
7:  $buffer \leftarrow \emptyset$                     ▷ buffer
8:  $mutex \leftarrow 1$                             ▷ mutex
9:  $empty \leftarrow n$                             ▷ empty slots
10:  $full \leftarrow 0$                             ▷ occupied slots
11: while  $i < n$  or  $j < m$  do
12:   if  $i < n$  and  $empty > 0$  then
13:      $p_i$                                 ▷ produce
14:      $i \leftarrow i + 1$ 
15:   end if
16:   if  $j < m$  and  $full > 0$  then
17:      $c_j$                                 ▷ consume
18:      $j \leftarrow j + 1$ 
19:   end if
20: end while
21: Return:  $buffer$ 

```

5.3 Implementation

```

/**
 * Producer-Consumer Problem
 */

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

#define BUFFER_SIZE 5

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 < 5; 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++;
    }
}

void *consumer(void *cno) {
    for (int i = 0; i < 5; 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++;
    }
}

int main() {
    pthread_t pro[5], con[5];
    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

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

    for (int i = 0; i < 3; i++)
        // Create the consumer thread
        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
Producer 3: Insert Item 1681692777 at 1
Producer 1: Insert Item 1714636915 at 2
Producer 3: Insert Item 1957747793 at 3
Producer 1: Insert Item 424238335 at 4
Producer 3: Insert Item 719885386 at 0
Producer 1: Insert Item 1649760492 at 1
Producer 3: Insert Item 596516649 at 2
Producer 1: Insert Item 1189641421 at 3
Producer 2: Insert Item 846930886 at 1
Producer 2: Insert Item 1350490027 at 0
Producer 2: Insert Item 783368690 at 1
Producer 2: Insert Item 1102520059 at 2
Producer 2: Insert Item 2044897763 at 3
Producer 3: Insert Item 1025202362 at 3
Consumer 1: Remove Item 1350490027 from 0
Consumer 1: Remove Item 783368690 from 1
Consumer 1: Remove Item 1102520059 from 2
Consumer 1: Remove Item 2044897763 from 3
Consumer 1: Remove Item 424238335 from 4
Consumer 2: Remove Item 1350490027 from 0
Consumer 2: Remove Item 783368690 from 1
Consumer 2: Remove Item 1102520059 from 2
Consumer 2: Remove Item 2044897763 from 3
Consumer 2: Remove Item 424238335 from 4
Consumer 3: Remove Item 1350490027 from 0
Consumer 3: Remove Item 783368690 from 1
Consumer 3: Remove Item 1102520059 from 2
Consumer 3: Remove Item 2044897763 from 3
Consumer 3: Remove Item 424238335 from 4
```