NOAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

Department of Computer Science & Telecommunication Engineering

Lab Report

COURSE TITLE: Operating System and System Programming Lab

COURSE CODE: CSTE 3108

Session: 2019-20

Year-03, Term-01

Submitted by:

Farman Arefin Tamim
ROLL: ASH2001043M

Submitted to:

Ratnadip Kuri
Assistant Professor
Department of Computer Science and Telecommunication Engineering
Noakhali Science and Technology University

Submission date: 09 September, 2023

Linux Shell Scripting

1. Finding the maximum element of an array

```
#!/bin/bash

my_array=(15 20 7 8 5 3)
mx=${my_array[0]}
for i in "${my_array[@]}"; do
    if ((i > mx)); then
        mx=$i
    fi
done
echo "The maximum element in the array is: $mx"
```

2. Finding Factorial in iterative method

```
#!/bin/bash

calculate_factorial() {
    local n="$1"
    local factorial=1

    if [ "$n" -lt 0 ]; then
        echo "Factorial is not defined for negative numbers."
        return
    fi

    for ((i = 1; i <= n; i++)); do
        factorial=$((factorial * i))
    done

    echo "The factorial of $n is: $factorial"
}

num=5
calculate_factorial "$num"
```

3. Finding Factorial in recursive method

```
#!/bin/bash

calculate_factorial_recursive() {
    local n="$1"

    if [ "$n" -eq 0 ]; then
        echo 1
    elif [ "$n" -lt 0 ]; then
        echo "Factorial is not defined for negative numbers."
    else
        local prev_factorial=$(calculate_factorial_recursive "$((n - 1))")
        echo "$((n * prev_factorial))"
    fi
}

num=5

result=$(calculate_factorial_recursive "$num")
echo "The factorial of $num is: $result"
```

4. <u>Fibonacci series in iterative method</u>

```
#!/bin/bash

calculate_fibonacci_iterative() {
    local n="$1"
    local a=0
    local b=1

    if [ "$n" -eq 0 ]; then
        echo "0"
        return
    fi

    if [ "$n" -eq 1 ]; then
        echo "0 1"
        return
    fi

    echo -n "0 1"

    for ((i = 2; i < n; i++)); do
        local next=$((a + b))
        echo -n " $next"
        a="$b"
        b="$next"
    done

    echo ""
}

num=10

calculate_fibonacci_iterative "$num"
```

5. Fibonacci series in recursive method

```
#!/bin/bash

calculate_fibonacci_recursive() {
    local n="$1"

    if [ "$n" -eq 0 ]; then
        echo -n "0"
    elif [ "$n" -eq 1 ]; then
        echo -n "0 1"
    else
        local prev_series=$(calculate_fibonacci_recursive "$((n - 1))")
        local prev_terms=($prev_series)
        local len=${#prev_terms[@]}
        local prev_term_1=${prev_terms[$((len - 1))]}
        local prev_term_2=${prev_terms[$((len - 2))]}
        local next_term=$((prev_term_1 + prev_term_2))
        echo -n "$prev_series $next_term"
    fi
}

num=10

result=$(calculate_fibonacci_recursive "$num
```

System Programming

1. FCFS algorithm

```
#include <stdio.h>

struct Process {
    int id;          // Process ID
    int arrival_time; // Arrival time
    int burst_time;   // Burst time
};
void calculateTimes(struct Process processes[], int n, int waiting_time[], int
turnaround_time[]) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    waiting_time[0] = 0;

    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
        total_waiting_time += waiting_time[i];
    }

    for (int i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
        total_turnaround_time += turnaround_time[i];
    }
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].arrival_time, processes[i].burst_time, waiting_time[i],
turnaround_time[i]);
    }

    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
```

```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];
    int waiting_time[n];
    int turnaround_time[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }

    calculateTimes(processes, n, waiting_time, turnaround_time);

    return 0;
}
```

2. <u>Shortes Job First algorithm</u>

```
#include <stdio.h>

struct Process {
    int id;          // Process ID
    int burst_time;   // Burst time
};
void sjfScheduling(struct Process processes[], int n) {
    int waiting_time[n];
    int turnaround_time[n];

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                // Swap processes[j] and processes[j+1]
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
    waiting_time[0] = 0;
    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
    }
```

```
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
    }
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
waiting_time[i], turnaround_time[i]);
    }
}

int main() {
    int n;


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

    struct Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }

    sjfScheduling(processes, n);

    return 0;
}
```

3. <u>Priority Scheduling Algorithm</u>

```
#include <stdio.h>
struct Process {
    int id;          // Process ID
    int priority;
    int burst_time;   // Burst time
};

void priorityScheduling(struct Process processes[], int n) {
    int waiting_time[n];
    int turnaround_time[n];
    waiting_time[0] = 0;

    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
    }
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
    }
    printf("Process\tPriority\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].priority,
processes[i].burst_time, waiting_time[i], turnaround_time[i]);
    }
}
```

```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Input process details (ID, priority, burst time)
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter priority for process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }


    priorityScheduling(processes, n);

    return 0;
}
```

4. Round Robin Scheduling Algorithm

```
#include <stdio.h>
struct Process {
    int id;          // Process ID
    int burst_time;   // Burst time
};
void roundRobinScheduling(struct Process processes[], int n, int
time_quantum) {
    int remaining_time[n];
    int waiting_time[n];
    int turnaround_time[n];
    int time = 0;

    for (int i = 0; i < n; i++) {
        remaining_time[i] = processes[i].burst_time;
    }

    while (1) {
        int all_finished = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
                all_finished = 0;
                if (remaining_time[i] > time_quantum) {
                    time += time_quantum;
                    remaining_time[i] -= time_quantum;
                } else {
                    time += remaining_time[i];
                    waiting_time[i] = time - processes[i].burst_time;
                    remaining_time[i] = 0;
                }
            }
        }

        if (all_finished) {
            break;
        }
    }
```

```
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = processes[i].burst_time + waiting_time[i];
    }


    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
waiting_time[i], turnaround_time[i]);
    }
}

int main() {
    int n;
    int time_quantum;

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


    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);

    struct Process processes[n];


    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }


    roundRobinScheduling(processes, n, time_quantum);

    return 0;
}
```

5. <u>Bankers Algorithm</u>

```
#include <stdio.h>
#define NUM_PROCESSES 5
#define NUM_RESOURCES 3
int isSafe(int available[], int max[][NUM_RESOURCES], int
allocation[][NUM_RESOURCES], int need[][NUM_RESOURCES], int process)
{
    int i;
    int work[NUM_RESOURCES];
    int finish[NUM_PROCESSES];
    for (i = 0; i < NUM_RESOURCES; i++) {
        work[i] = available[i];
    }

    for (i = 0; i < NUM_PROCESSES; i++) {
        finish[i] = 0;
    }
    int count = 0;
    while (count < NUM_PROCESSES) {
        int found = 0;
        for (i = 0; i < NUM_PROCESSES; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < NUM_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        break;
                    }
                }
                if (j == NUM_RESOURCES) {
                    for (j = 0; j < NUM_RESOURCES; j++) {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = 1;
                    found = 1;
                    count++;
                }
            }
        }
```

```
        if (!found) {
            return 0;
        }
    }
    return 1;
}

int main() {
    int available[NUM_RESOURCES] = {3, 3, 2};
    int max[NUM_PROCESSES][NUM_RESOURCES] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3},
    };
    int allocation[NUM_PROCESSES][NUM_RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2},
    };
    int need[NUM_PROCESSES][NUM_RESOURCES];
    int i, j;
    for (i = 0; i < NUM_PROCESSES; i++) {
        for (j = 0; j < NUM_RESOURCES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
    for (i = 0; i < NUM_PROCESSES; i++) {
        if (isSafe(available, max, allocation, need, i)) {
            printf("Process %d can safely request resources.\n", i);
        } else {
            printf("Process %d cannot safely request resources.\n", i);
        }
    }
    return 0;
}
```

6. Producer Consumer

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

int main() {
    int buffer[10];
    int bufsize = 10;
    int in = 0, out = 0;
    int produce, consume, choice;

    for (;;) {
        printf("\nMenu:\n");
        printf("1. Produce an item\n");
        printf("2. Consume an item\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if ((in + 1) % bufsize == out) {
                    printf("Buffer is full. Cannot produce.\n");
                } else {
                    printf("Enter the item to produce: ");
                    scanf("%d", &produce);
                    buffer[in] = produce;
                    in = (in + 1) % bufsize;
                }
                break;
            case 2:
                if (in == out) {
                    printf("Buffer is empty. Nothing to consume.\n");
                } else {
                    consume = buffer[out];
                    printf("Consumed item: %d\n", consume);
                    out = (out + 1) % bufsize;
                }
                break;
            case 3:
```

```
            printf("Exiting the program.\n");
            exit(0);
        default:
            printf("Invalid choice. Please enter a valid option (1-3).\n");
        }
    }

    return 0;
}
```

7. <u>Petersons Algorithm</u>

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

#define NUM_PROCESSES 2
bool in_critical_section[NUM_PROCESSES];
int current_turn;

void enter_critical_section(int process) {
    int other_process = 1 - process;
    in_critical_section[process] = true;
    current_turn = process;
    while (in_critical_section[other_process] && current_turn == process);
    printf("Process %d is in the critical section.\n", process);
}


void exit_critical_section(int process) {
    in_critical_section[process] = false;
}
```

```
int main() {
    for (int i = 0; i < NUM_PROCESSES; i++)
        in_critical_section[i] = false;
    current_turn = 0;


    for (int i = 0; i < 10; i++) {
        int process = i % NUM_PROCESSES;
        enter_critical_section(process);
        exit_critical_section(process);
    }

    return 0;
}
```

8. <u>Semaphore</u>

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

int buffer_mutex = 1, buffer_full = 0, buffer_empty = 3, item_count = 0;

int main() {
    int choice;
    void produce();
    void consume();
    int wait(int);
    int signal(int);

    printf("\n1. Produce\n2. Consume\n3. Exit");
    while (1) {
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if ((buffer_mutex == 1) && (buffer_empty != 0))
                    produce();
                else
                    printf("Buffer is full!!");
                break;
            case 2:
                if ((buffer_mutex == 1) && (buffer_full != 0))
                    consume();
                else
                    printf("Buffer is empty!!");
                break;
            case 3:
                exit(0);
                break;
        }
    }
    return 0;
}
```

```
int wait(int s) {
    return (--s);
}

int signal(int s) {
    return (++s);
}

void produce() {
    buffer_mutex = wait(buffer_mutex);
    buffer_full = signal(buffer_full);
    buffer_empty = wait(buffer_empty);
    item_count++;
    printf("\nProducer produces item %d", item_count);
    buffer_mutex = signal(buffer_mutex);
}

void consume() {
    buffer_mutex = wait(buffer_mutex);
    buffer_full = wait(buffer_full);
    buffer_empty = signal(buffer_empty);
    printf("\nConsumer consumes item %d", item_count);
    item_count--;
    buffer_mutex = signal(buffer_mutex);
}
```