ASSIGNMENT

By
Sambhav Manhas
2022a1r051
3rd semester
CSE

Model Institute of Engineering & Technology (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade) Jammu, India

2023

ASSIGNMENT

Subject Code: Operating System (OS)

Due Date:4 December

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	
Faculty Signature Email:mekhal.cse@mietjammu.in				

Task 1:

Create a program that simulates the FCFS scheduling algorithm and SJF scheduling in a simple operating system environment to demonstrate its behavior and advantages. Provide clear output that shows the execution order of processes, waiting times, turnaround times, and average statistics. Compare the response time of each algorithm.

Task 2:

Create a shell script that generates random passwords:

A. The script should accept the desired password length as a command-line argument. It should generate a random password containing a mix of letters (both uppercase and lowercase), numbers, and special character

Task 1

Algorithm Explanation:

Process Class:

1. Define a Process class to represent individual processes with attributes such as name, arrival time, burst time, waiting time, and turnaround time.

FCFS Scheduler:

1. Implement the FCFS scheduling algorithm:
    1. Sort the processes based on their arrival times.
    2. Iterate through the sorted processes.
    3. Calculate waiting time and turnaround time for each process.
    4. Update the current time and move to the next process.

SJF Scheduler:

1. Implement the SJF scheduling algorithm:
    1. Sort the processes based on arrival time and burst time.
    2. Iterate through the sorted processes.
    3. Calculate waiting time and turnaround time for each process.
    4. Update the current time and move to the next process.

Print Results:

1. Display the results, including the arrival time, burst time, waiting time, and turnaround time for each process.

User Interaction:

1. Allow the user to choose between FCFS and SJF scheduling algorithms.
2. Simulate the chosen algorithm on a set of example processes.

Example Usage:

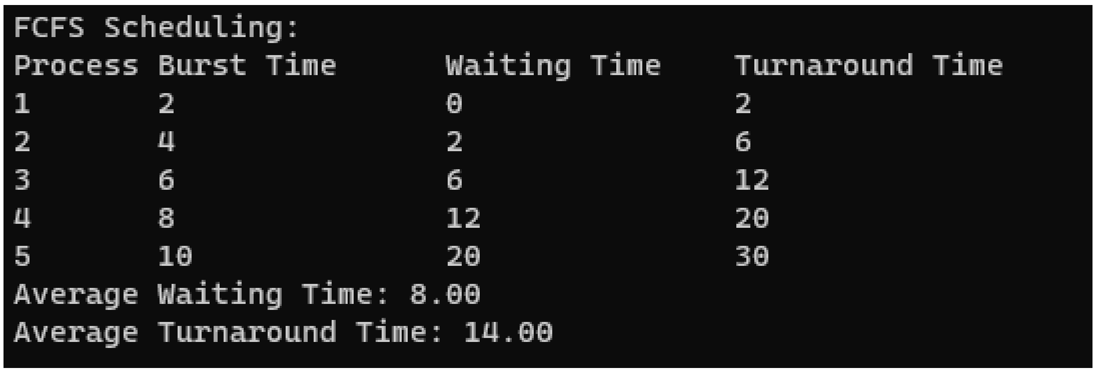
1. Create a list of example processes with different arrival times and burst times.
2. Prompt the user to choose FCFS or SJF scheduling.
3. Simulate the selected algorithm on the example processes.
4. Display the results, including waiting time and turnaround time for each process.

OVERALL:

The program simulates the FCFS and SJF scheduling algorithms for a set of processes in a simple operating system environment. It defines a `Process` class, sorts processes based on arrival and burst times, calculates waiting and turnaround times, and allows user interaction to choose between FCFS and SJF. The results, including process details and performance metrics, are displayed for the selected scheduling algorithm. The program serves as a basic illustration of these scheduling strategies in an operating system.

EXECUTION SEQUENCE:

FCFS Execution Sequence(for when all process arrive at time zero):



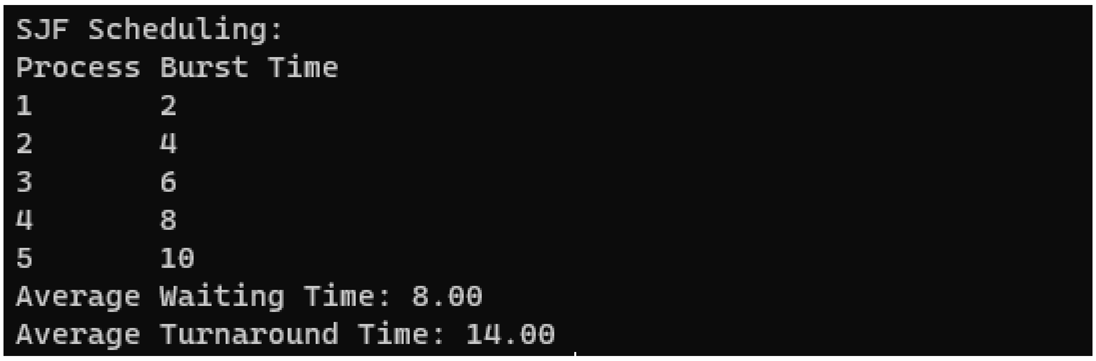
```
FCFS Scheduling:
Process Burst Time      Waiting Time    Turnaround Time
1       2               0               2
2       4               2               6
3       6               6               12
4       8               12              20
5       10              20              30
Average Waiting Time: 8.00
Average Turnaround Time: 14.00
```

Gantt Chart:

P1	P2	P3	P4	P5

0 2 6 12 20 30

SJF Execution Sequence(for when all process arrive at time zero):



```
SJF Scheduling:
Process Burst Time
1       2
2       4
3       6
4       8
5       10
Average Waiting Time: 8.00
Average Turnaround Time: 14.00
```

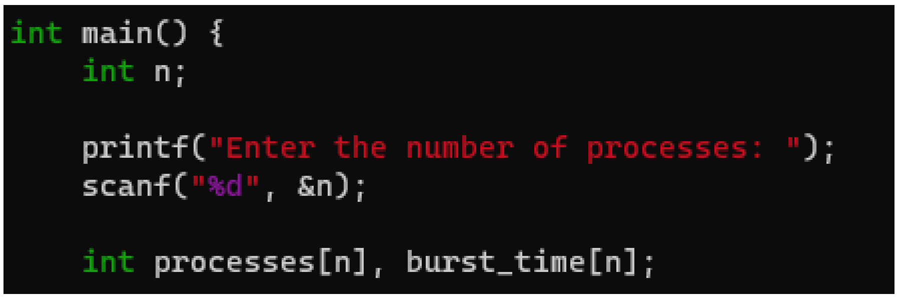
Gantt Chart:

P1	P2	P3	P4	P5

0 2 6 12 20 30

CODE EXPLANATION

1. Variable Declaration :

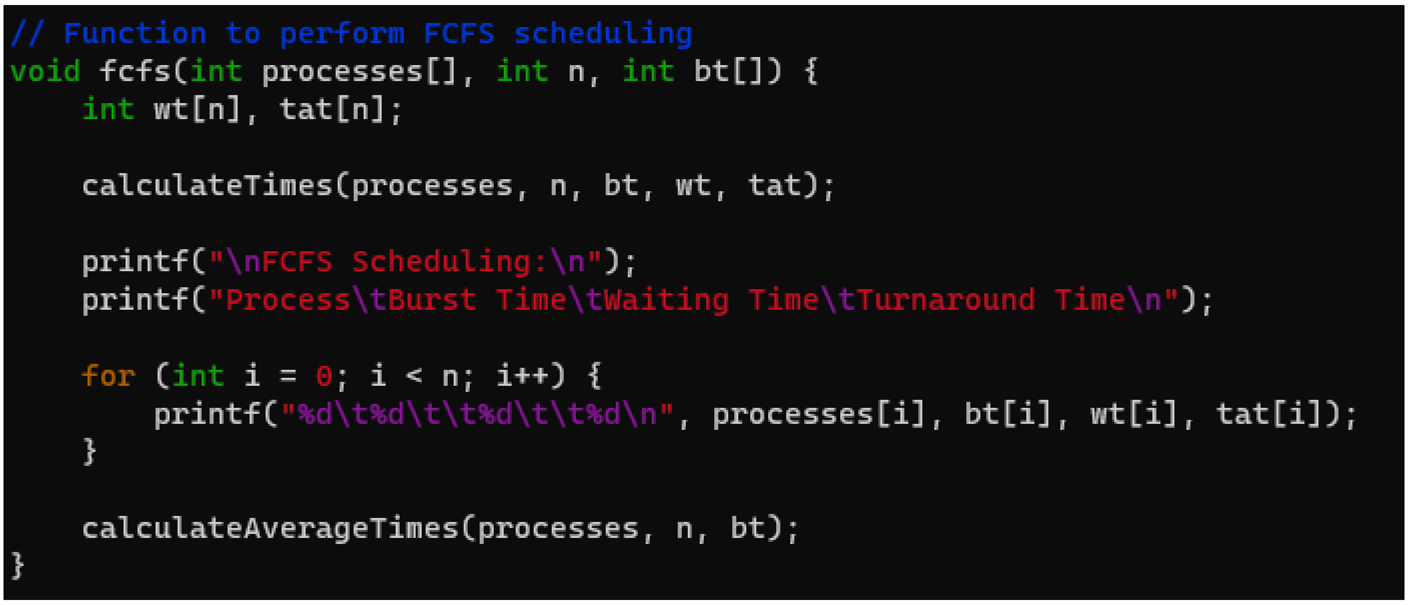


```c
int main() {
    int n;

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

    int processes[n], burst_time[n];
```

Explanation:

- This section defines a Process class with attributes for a process, including its name, arrival time, burst time, waiting time, and turnaround time.
- Each process object will store these attributes.

2. FCFS Scheduler:

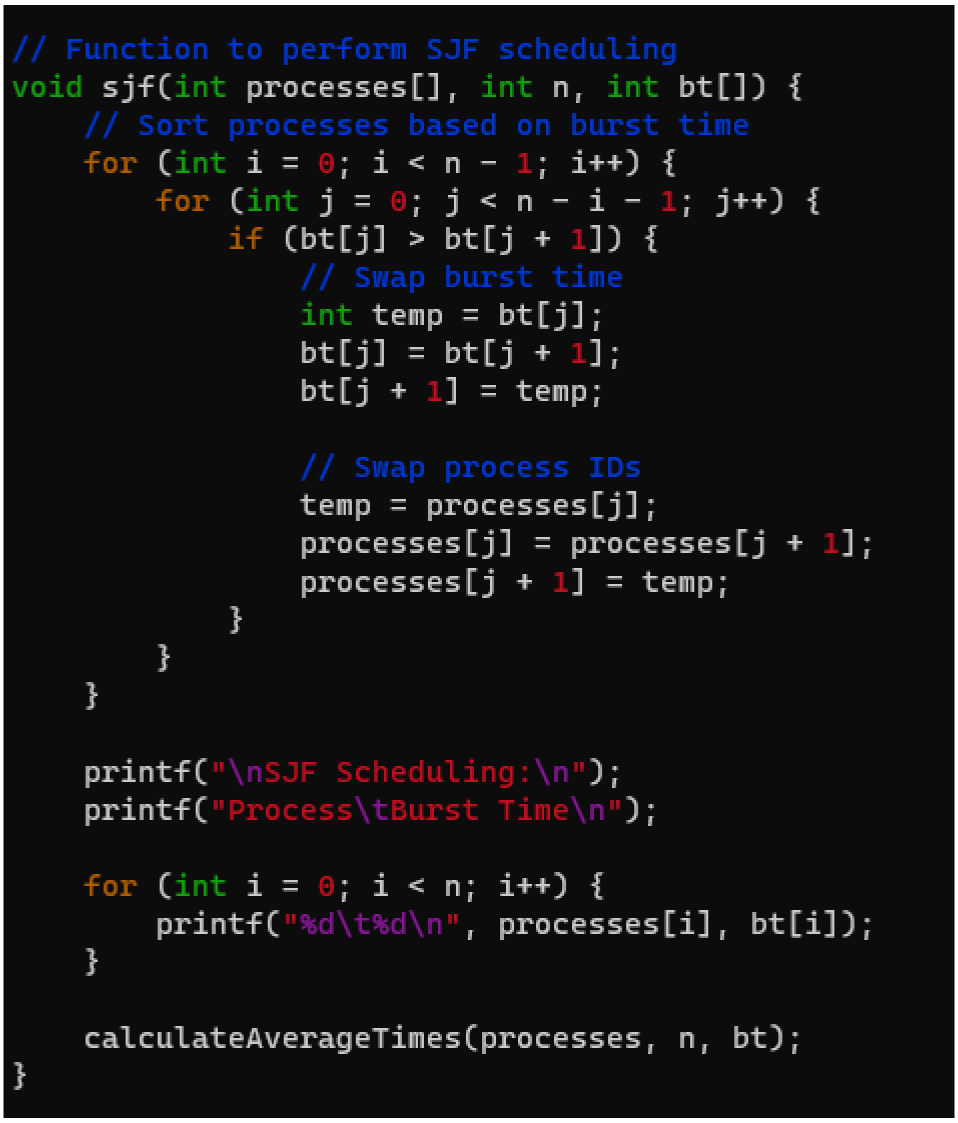


```c
// Function to perform FCFS scheduling
void fcfs(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    calculateTimes(processes, n, bt, wt, tat);

    printf("\nFCFS Scheduling:\n");
    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], bt[i], wt[i], tat[i]);
    }

    calculateAverageTimes(processes, n, bt);
}
```

Explanation:

- This section defines the function fcfs_scheduler that takes a list of processes as input.
- The processes are sorted based on their arrival times.
- The code iterates through the sorted processes, calculates waiting and turnaround times, and updates the current time.
- The execution sequence is recorded in the execution_sequence list.

3. SJF Scheduler:

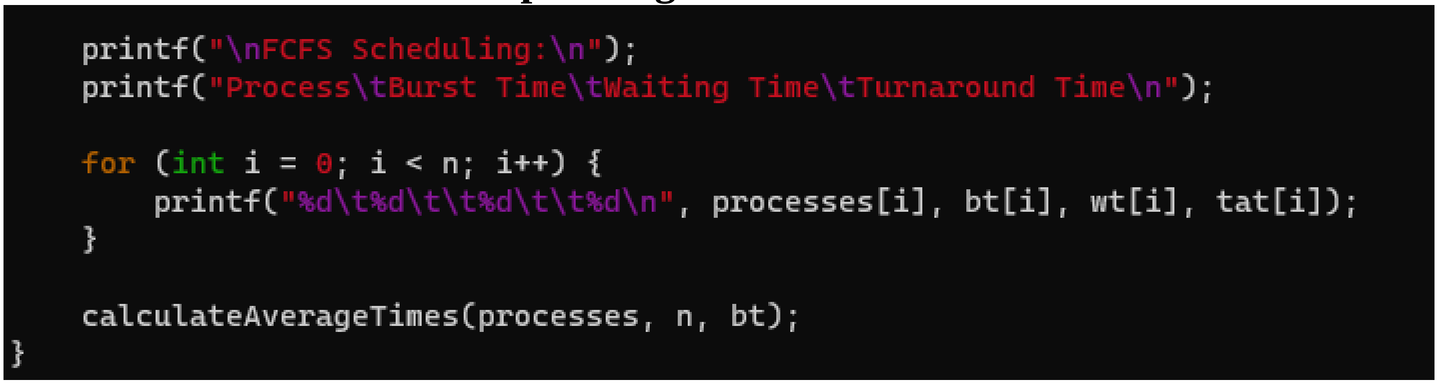


```c
// Function to perform SJF scheduling
void sjf(int processes[], int n, int bt[]) {
    // Sort processes based on burst time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (bt[j] > bt[j + 1]) {
                // Swap burst time
                int temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;

                // Swap process IDs
                temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("\nSJF Scheduling:\n");
    printf("Process\tBurst Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\n", processes[i], bt[i]);
    }

    calculateAverageTimes(processes, n, bt);
}
```

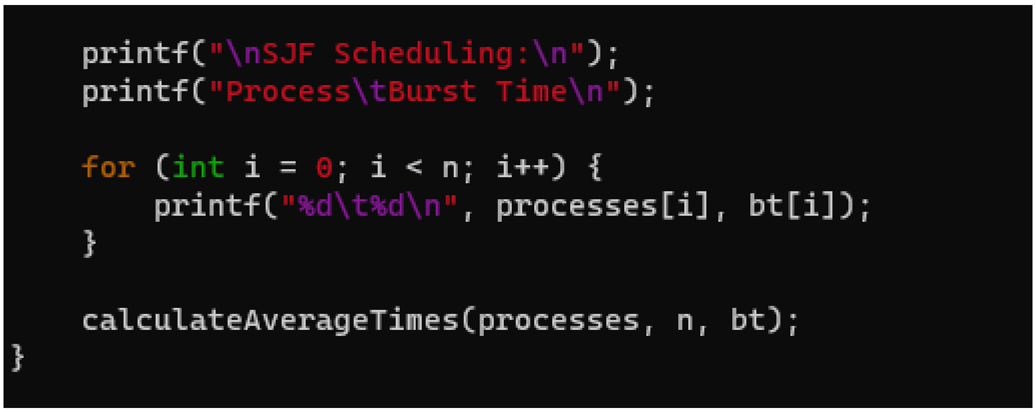
Explanation:

- This section defines the function sjf scheduler that takes a list of processes as input.
- The processes are sorted based on arrival time and burst time.
- The code iterates through the sorted processes, calculates waiting and turnaround times, and updates the current time.
- The execution sequence is recorded in the execution sequence list

4. Print Results and Example Usage:



```
    printf("\nFCFS Scheduling:\n");
    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], bt[i], wt[i], tat[i]);
    }

    calculateAverageTimes(processes, n, bt);
}
```

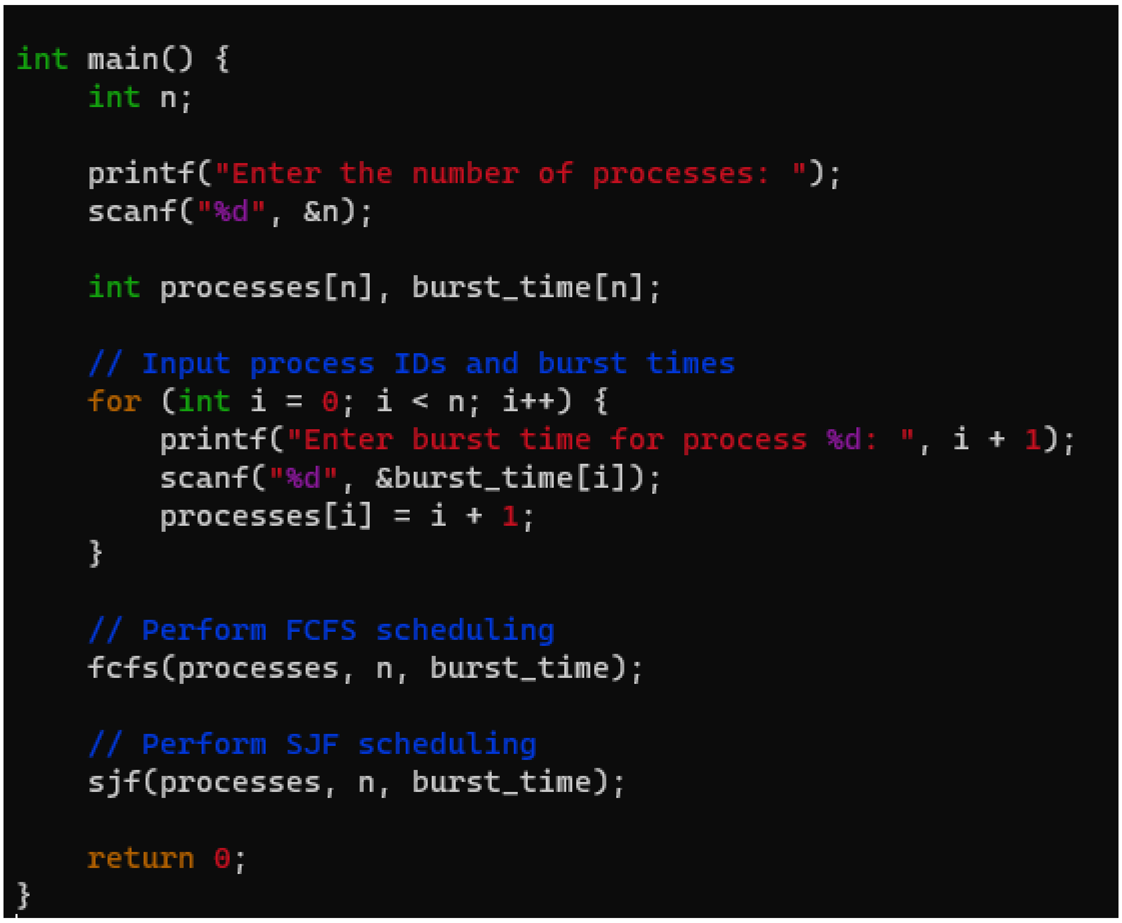


```
    printf("\nSJF Scheduling:\n");
    printf("Process\tBurst Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\n", processes[i], bt[i]);
    }

    calculateAverageTimes(processes, n, bt);
}
```

Explanation:

- This section defines the print_results function for displaying process details.
- An example list processes_combined is created with instances of the Process class.

5. User Interaction :

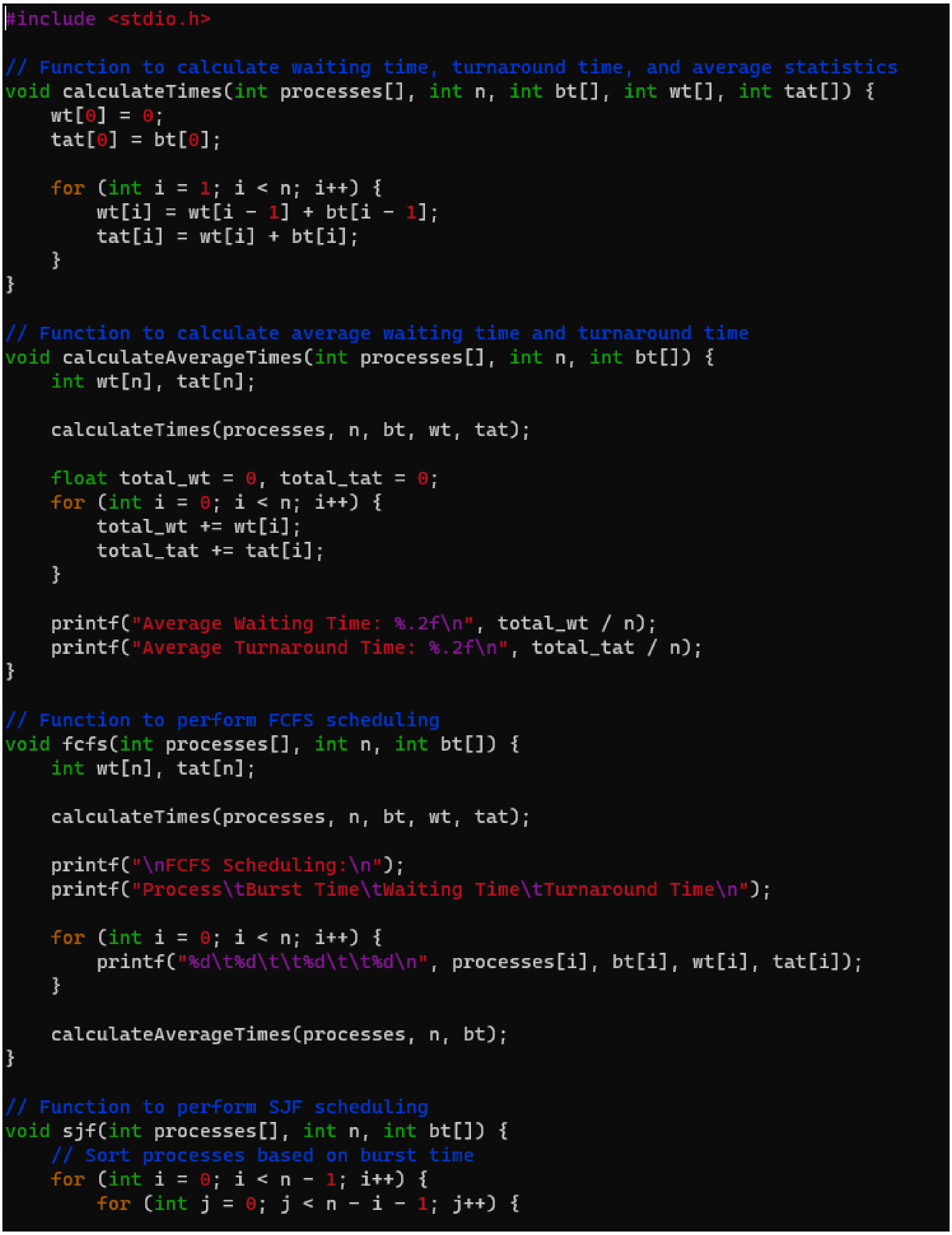


```c
int main() {
    int n;

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

    int processes[n], burst_time[n];

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

    // Perform FCFS scheduling
    fcfs(processes, n, burst_time);

    // Perform SJF scheduling
    sjf(processes, n, burst_time);

    return 0;
}
```

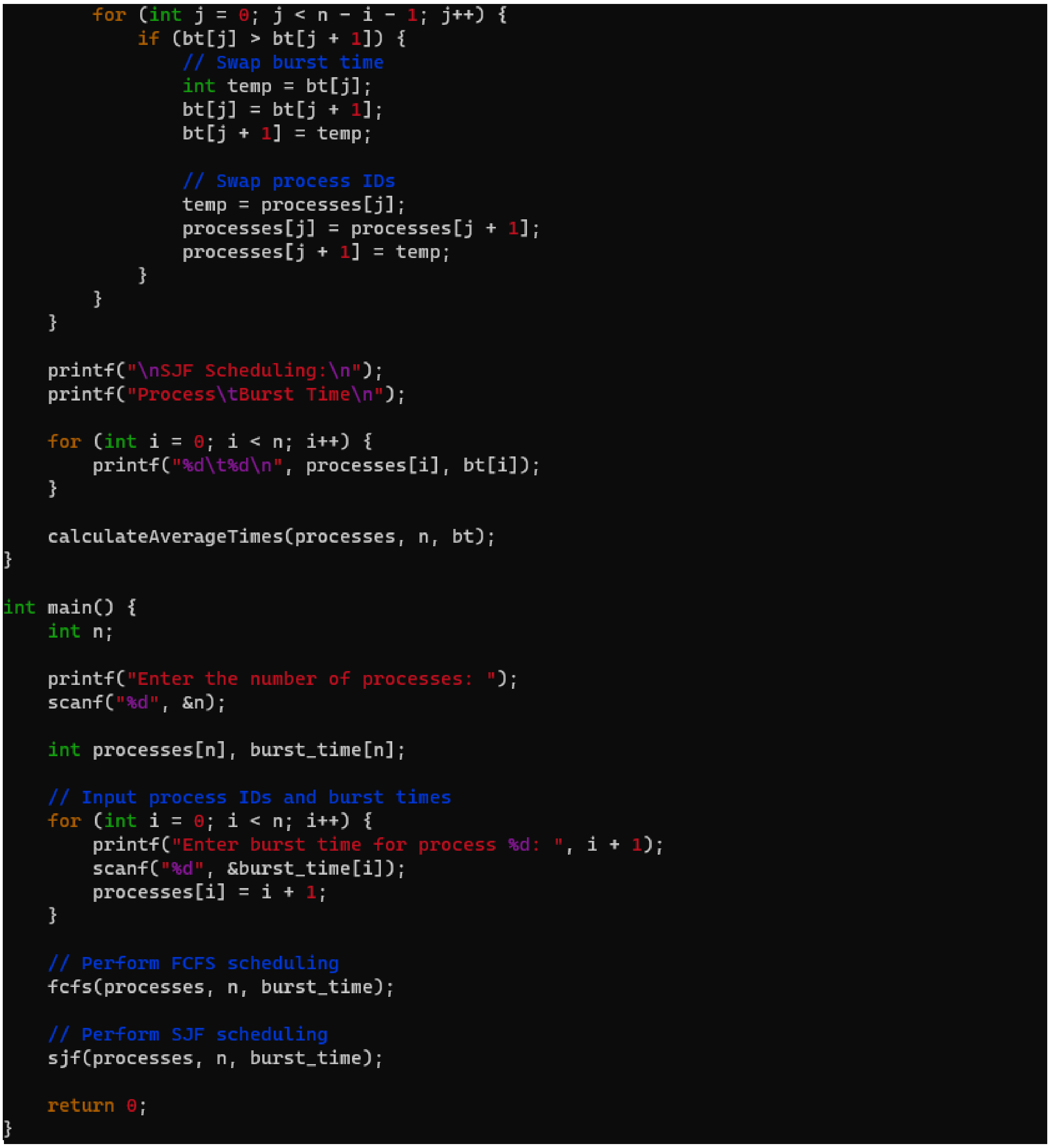
Explanation:

- This section prompts the user to choose between FCFS and SJF scheduling algorithms.
- It calls the respective scheduler function based on the user's choice.
- Process details and the execution sequence are displayed.

Source code:



```c
#include <stdio.h>

// Function to calculate waiting time, turnaround time, and average statistics
void calculateTimes(int processes[], int n, int bt[], int wt[], int tat[]) {
    wt[0] = 0;
    tat[0] = bt[0];

    for (int i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
        tat[i] = wt[i] + bt[i];
    }
}

// Function to calculate average waiting time and turnaround time
void calculateAverageTimes(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    calculateTimes(processes, n, bt, wt, tat);

    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("Average Waiting Time: %.2f\n", total_wt / n);
    printf("Average Turnaround Time: %.2f\n", total_tat / n);
}

// Function to perform FCFS scheduling
void fcfs(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    calculateTimes(processes, n, bt, wt, tat);

    printf("\nFCFS Scheduling:\n");
    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], bt[i], wt[i], tat[i]);
    }

    calculateAverageTimes(processes, n, bt);
}

// Function to perform SJF scheduling
void sjf(int processes[], int n, int bt[]) {
    // Sort processes based on burst time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
```



```c
        for (int j = 0; j < n - i - 1; j++) {
            if (bt[j] > bt[j + 1]) {
                // Swap burst time
                int temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;

                // Swap process IDs
                temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("\nSJF Scheduling:\n");
    printf("Process\tBurst Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\n", processes[i], bt[i]);
    }

    calculateAverageTimes(processes, n, bt);
}

int main() {
    int n;

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

    int processes[n], burst_time[n];

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

    // Perform FCFS scheduling
    fcfs(processes, n, burst_time);

    // Perform SJF scheduling
    sjf(processes, n, burst_time);

    return 0;
}
```

Task 2

Algorithm Explanation:

Check Command-Line Argument:

1. **Brief:**
   1. Ensure the user provides exactly one command-line argument.
2. **Deep:**
   1. Check if the number of command-line arguments is not equal to 1.
   2. If the condition is true, display a usage message and exit with an error.

Generate Password Function:

3. **Brief:**
   1. Define a function to generate a random password of a specified length.
4. **Deep:**
   1. Capture the password length from the function argument.
   2. Define character sets for lowercase letters, uppercase letters, numbers, and special characters.
   3. Combine character sets into all_chars.
   4. Use a loop to randomly select characters from all_chars and construct the password.

Get Password Length from Command-Line:

5. **Brief:**
   1. Capture the desired password length from the command-line argument.

2.

Generate and Print Password:

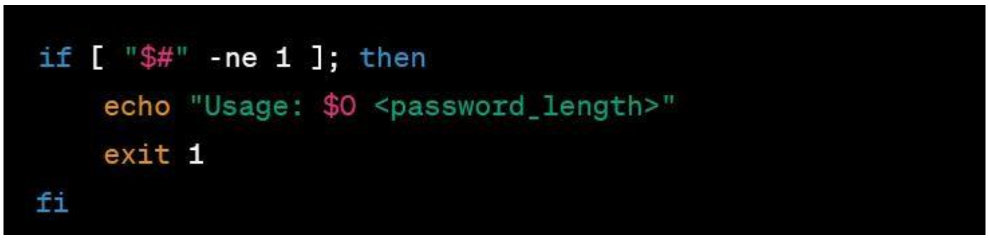
1. **Brief:**
    1. Call the generate_password function with the specified length.
    2. Print the generated password.
2. **Deep:**
    1. Call the function with the captured password length.
    2. Store the generated password in a variable.
    3. Echo the message "Generated Password: " followed by the actual password.

Overall:

The script ensures proper user input, defines a function to generate random passwords with a mix of characters, captures the desired password length, and prints the generated password. It provides a flexible and secure way to create random passwords of varying lengths with a mix of characters.

CODE EXPLANATION

1. Checking Command-Line Argument:



```
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <password_length>"
    exit 1
fi
```

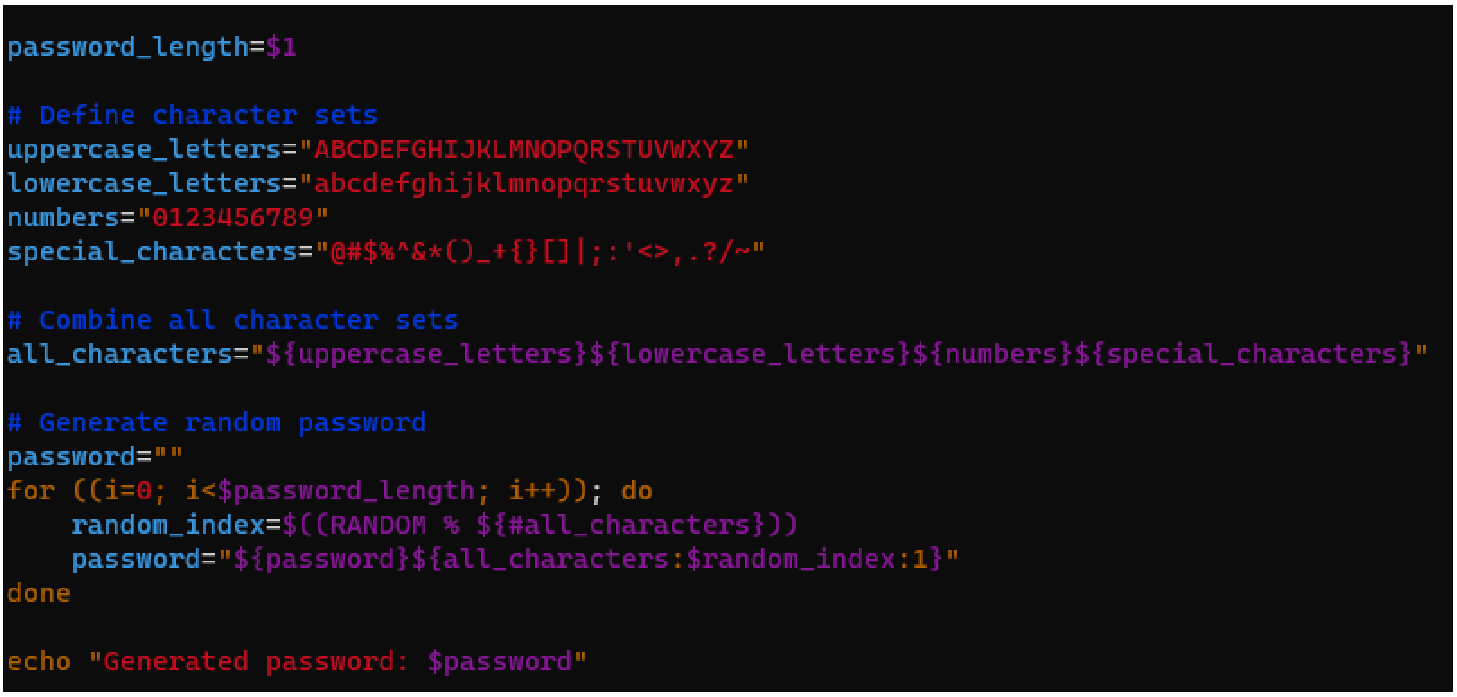
Brief Explanation:

- This section checks if the user provided exactly one command-line argument.
- If not, it displays a usage message and exits with an error.

Deep Explanation:

- "$#" represents the number of command-line arguments.
- The condition ["$#" -ne 1] checks if the number of arguments is not equal to 1.
- If the condition is true, it echoes a usage message displaying the script name ($0) and the expected argument.
- The script then exits with an error code (exit 1).

2.Function to Generate Password:



```
password_length=$1

# Define character sets
uppercase_letters="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
lowercase_letters="abcdefghijklmnopqrstuvwxyz"
numbers="0123456789"
special_characters="@#$%^&*()_+{}[]|;:'<>,.?/~"

# Combine all character sets
all_characters="${uppercase_letters}${lowercase_letters}${numbers}${special_characters}"

# Generate random password
password=""
for ((i=0; i<$password_length; i++)); do
    random_index=$((RANDOM % ${#all_characters}))
    password="${password}${all_characters:$random_index:1}"
done

echo "Generated password: $password"
```

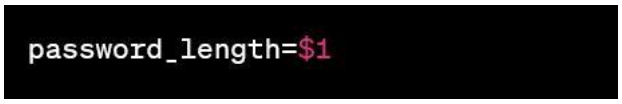
Brief Explanation:

- This section defines a function generate_password to generate a random password of a specified length.
- It uses a mix of lowercase letters, uppercase letters, numbers, and special characters.

Deep Explanation:

- local length=$1 captures the password length from the function argument.
- Character sets for lowercase, uppercase, numbers, and special characters are defined.
- These character sets are combined into all_chars.
- The function uses a loop to randomly select characters from all_chars and append them to the password variable.
- The final password is echoed.

3. Get Password Length from Command-Line:



```
password_length=$1
```

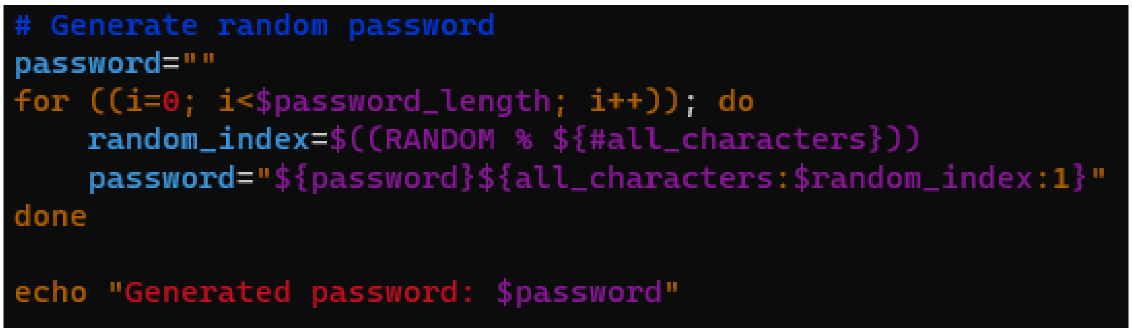
Brief Explanation:

- This section captures the desired password length from the command-line argument.

Deep Explanation:

- password_length=$1 assigns the value of the first command-line argument to the variable password_length.

4. Generate and Print Password:



```
# Generate random password
password=""
for ((i=0; i<$password_length; i++)); do
    random_index=$((RANDOM % ${#all_characters}))
    password="${password}${all_characters:$random_index:1}"
done

echo "Generated password: $password"
```

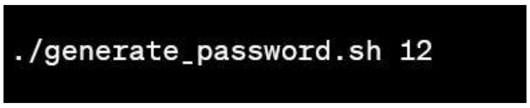
Brief Explanation:

- This section calls the generate_password function with the specified password length and prints the result.

Deep Explanation:

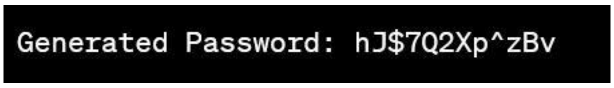
- generate_password "$password_length" calls the function with the captured password length.
- The generated password is stored in the variable generated_password.
- The script then echoes the message "Generated Password: " followed by the actual password

`Example Usage:`



```
./generate_password.sh 12
```

This command generates and displays a random password with a length of 12 characters. The output will look like:



```
Generated Password: hJ$7Q2Xp^zBv
```

This breakdown provides a high-level overview and explanation of each section of the code for generating random passwords in a shell script.

Algorithm

Run the Script:

- The user runs the script from the command line, providing a desired password length as a command-line argument.

· · · **Check Command-Line Argument:**

- · The script checks if the user provided exactly one command-line argument.

· **Generate Password Function:**

- · The script defines the generate_password function to create a random password of the specified length.

· · · **Get Password Length from Command-Line:**

- · The script captures the desired password length from the command-line argument.

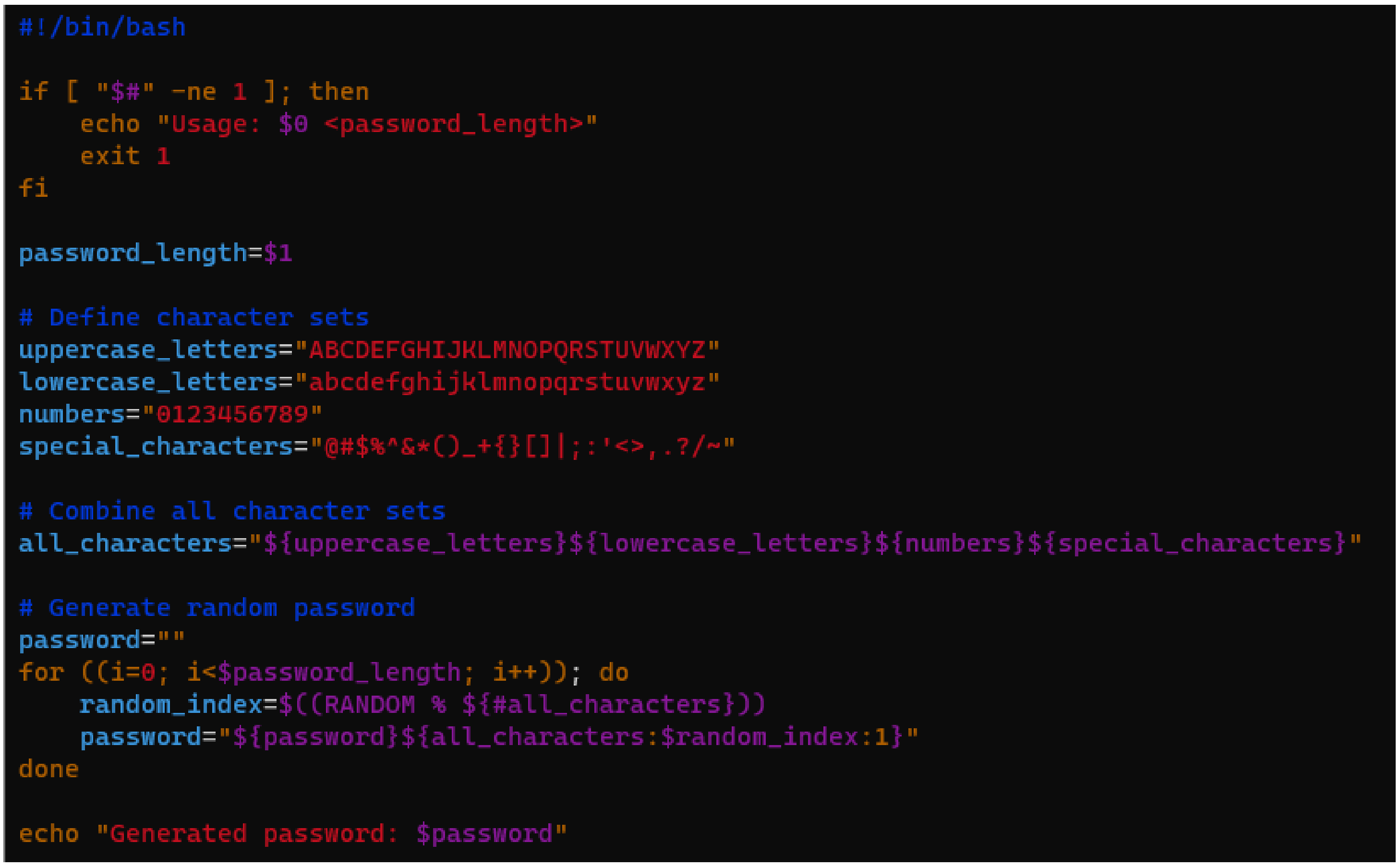
· **Generate and Print Password:**

- · The script calls the generate_password function with the specified password length.

The generated password is stored in the variable generated_password.

○ The script prints the message "Generated Password: " followed by theactual generated password.

SOURCE CODE



```bash
#!/bin/bash

if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <password_length>"
    exit 1
fi

password_length=$1

# Define character sets
uppercase_letters="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
lowercase_letters="abcdefghijklmnopqrstuvwxyz"
numbers="0123456789"
special_characters="@#$%^&*()_+{}[]|;:'<>,.?/~"

# Combine all character sets
all_characters="${uppercase_letters}${lowercase_letters}${numbers}${special_characters}"

# Generate random password
password=""
for ((i=0; i<$password_length; i++)); do
    random_index=$((RANDOM % ${#all_characters}))
    password="${password}${all_characters:$random_index:1}"
done

echo "Generated password: $password"
```

GROUP MEMBERS