‘येथे बहुतांचे हित ।’

Marathwada Mitra Mandal’s

COLLEGE OF ENGINEERING

Karvenagar, PUNE – 411 052

Accredited with “A++” Grade by NAAC // Accredited by NBA (Mechanical Engg. & Electrical Engg.)
Recipient of “Best College Award 2019” by SPPU // Recognized under section 2(f) and 12B of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY

LABORATORY MANUAL

TE (INFORMATION TECHNOLOGY)

(SEMESTER – I)

314446: OPERATING SYSTEMS LABORATORY

2019 Course

‘येथे बहुतांचे हित ।’

Marathwada Mitra Mandal’s

COLLEGE OF ENGINEERING

Karvenagar, PUNE – 411 052

Accredited with “A++” Grade by NAAC // Accredited by NBA (Mechanical Engg. & Electrical Engg.)

Recipient of “Best College Award 2019” by SPPU // Recognized under section 2(f) and 12B of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY

Course Code: 314446

Course Name: OPERATING SYSTEMS LABORATORY

Teaching Scheme: **Credits:** 02 **Examination Scheme:**

Practical: 4 hrs/week

TW: 25 Marks

Practical: 25 Marks

Course Outcomes:

Course Outcome	**Statement**
	At the end of the course, a student will be able to:
CO1	Apply the basics of Linux commands.
CO2	Build shell scripts for various applications.

CO3	Implement basic building blocks like processes, threads under the Linux.
CO4	Develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling in Linux.
CO5	Develop various system programs for the functioning of OS concepts in user space like Memory Management and Disk Scheduling in Linux.
CO6	Develop system programs for Inter Process Communication in Linux.

'येथे बहुतांचे हित ।'

Marathwada Mitra Mandal's
COLLEGE OF ENGINEERING
Karvenagar, PUNE – 411 052
Accredited with "A++" Grade by NAAC // Accredited by NBA (Mechanical Engg. & Electrical Engg.)
Recipient of "Best College Award 2019" by SPPU // Recognized under section 2(f) and 12B of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY

Sr. No.	Title of Assignment	PO Mapping	PSO Mapping
1	A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc. B. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1

2	Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states. A. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states. B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
3	Implement the C program for CPU Scheduling Algorithms: Shortest Job First	PO1, PO2, PO3, PO4, PO5, PO12	PSO1, PSO2

	(Preemptive) and Round Robin with different arrival time.		
4	Thread synchronization using counting semaphores. A. Application to demonstrate: producer- consumer problem with counting semaphores and mutex. B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1, PSO2
5	Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
6	Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
7	Inter process communication in Linux using following. A. FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words	PO1, PO2, PO3, PO4, PO5, PO12	PSO1

	and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output. B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.		
8	Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1

Assignment No. 1A

Aim: Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

1. echo

The echo command is used to display a line of text/string that is passed as an argument.

Example:

echo "Hello, World!"

This will print Hello, World! to the terminal.

2. ls

The ls command lists the contents of a directory.

Examples:

ls

This will list all files and directories in the current directory.

ls -l

This will list all files and directories in the current directory with detailed information.

3. read

The read command reads a line of input from the standard input.

Example:

echo "Enter your name: "

read name

echo "Hello, $name!"

This script will prompt the user to enter their name and then greet them.

4. cat

The cat command is used to concatenate and display files.

Examples:

cat file.txt

This will display the contents of file.txt.

cat file1.txt file2.txt > combined.txt

This will concatenate file1.txt and file2.txt and save the result to combined.txt.

5. touch

The touch command is used to create an empty file or update the timestamp of an existing file.

Example:

```
touch newfile.txt
```

This will create an empty file named newfile.txt.

6. test

The test command is used to evaluate conditional expressions.

Example:

```
test -e file.txt && echo "File exists" || echo "File does not exist"
```

This will check if file.txt exists and print the appropriate message.

7. Loops

For Loop

A for loop iterates over a list of items.

Example:

```
for i in 1 2 3 4 5; do
  echo "Welcome $i times"
done
```

While Loop

A while loop continues as long as the condition is true.

Example:

```
counter=1
while [ $counter -le 5 ]; do
  echo "Counter: $counter"
  ((counter++))
done
```

This will print the counter from 1 to 5.

8. Arithmetic Comparison

Arithmetic operations can be performed using the (( )) syntax.

Example:

```
a=5
b=10
if (( a < b )); then
  echo "$a is less than $b"
fi
```

This will print 5 is less than 10.

9. Conditional Statements

Conditional statements use if, else if, and else.

Example:

```
read -p "Enter a number: " num
if (( num > 0 )); then
  echo "Positive number"
elif (( num < 0 )); then
  echo "Negative number"
else
  echo "Zero"
fi
```

This script will classify the entered number as positive, negative, or zero.

10. grep

The grep command searches for patterns in files.

Example:

```
grep "pattern" file.txt
```

This will search for the string "pattern" in file.txt and display matching lines.

11. sed

The sed command is a stream editor for filtering and transforming text.

Example:

```
sed 's/old/new/g' file.txt
```

This will replace all occurrences of old with new in file.txt.

Assignment No. 1B

}

Aim: Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

```
#!/bin/bash

ADDRESS_BOOK="address_book.txt"

create_address_book() {
  if [ ! -f "$ADDRESS_BOOK" ]; then
    touch "$ADDRESS_BOOK"
    echo "Address book created."
  else
    echo "Address book already exists."
  fi
}

view_address_book() {
  if [ -f "$ADDRESS_BOOK" ]; then
    if [ -s "$ADDRESS_BOOK" ]; then
      cat "$ADDRESS_BOOK"
    else
      echo "Address book is empty."
    fi
  else
    echo "Address book does not exist. Please create it first."
  fi
}
```

```
insert_record() {
  echo "Enter Name:"
  read name
  echo "Enter Phone:"
  read phone
  echo "Enter Email:"
  read email
  echo "$name, $phone, $email" >> "$ADDRESS_BOOK"
  echo "Record inserted."
}

delete_record() {
  echo "Enter the Name of the record to delete:"
  read name
  if grep -q "^$name," "$ADDRESS_BOOK"; then
    grep -v "^$name," "$ADDRESS_BOOK" > temp.txt && mv temp.txt
"$ADDRESS_BOOK"
    echo "Record deleted."
  else
    echo "Record not found."
  fi
}

modify_record() {
  echo "Enter the Name of the record to modify:"
  read name
  if grep -q "^$name," "$ADDRESS_BOOK"; then
    grep -v "^$name," "$ADDRESS_BOOK" > temp.txt
    echo "Enter new Phone:"
    read phone
```

```
    echo "Enter new Email:"
    read email
    echo "$name, $phone, $email" >> temp.txt
    mv temp.txt "$ADDRESS_BOOK"
    echo "Record modified."
  else
    echo "Record not found."
  fi
}

while true; do
  echo ""
  echo "Address Book Menu:"
  echo "a) Create address book"
  echo "b) View address book"
  echo "c) Insert a record"
  echo "d) Delete a record"
  echo "e) Modify a record"
  echo "f) Exit"
  echo "Enter your choice: "
  read choice
  case $choice in
    a) create_address_book ;;
    b) view_address_book ;;
    c) insert_record ;;
    d) delete_record ;;
    e) modify_record ;;
    f) echo "Exiting..."; exit 0 ;;
    *) echo "Invalid choice. Please try again." ;;
  esac
```

done

Explanation:

1. **Creating the Address Book (create_address_book)**:
   - This function checks if the address book file exists. If not, it creates an empty file.
2. **Viewing the Address Book (view_address_book)**:
   - This function displays the contents of the address book if it exists and is not empty.
3. **Inserting a Record (insert_record)**:
   - Prompts the user to enter a name, phone number, and email, then appends this information to the address book.
4. **Deleting a Record (delete_record)**:
   - Prompts the user to enter the name of the record to delete. It then removes the record if found.
5. **Modifying a Record (modify_record)**:
   - Prompts the user to enter the name of the record to modify. If found, it prompts for new phone and email details and updates the record.
6. **Menu Loop**:
   - Provides a menu for the user to choose an option and calls the corresponding function based on the user's input.

#!/bin/bash: Shebang line to specify the script should be run using the bash shell.
ADDRESS_BOOK: Variable storing the name of the address book file.

if [! -f "$ADDRESS_BOOK"]; then: Checks if the address book file does not exist.
touch "$ADDRESS_BOOK": Creates an empty file for the address book.
echo: Prints a message to the terminal.
if [-f "$ADDRESS_BOOK"]; then: Checks if the address book file exists.
if [-s "$ADDRESS_BOOK"]; then: Checks if the address book file is not empty.
cat "$ADDRESS_BOOK": Displays the contents of the address book file.
read name: Reads user input and stores it in the name variable.
echo "$name, $phone, $email" >> "$ADDRESS_BOOK": Appends the new record to the address book file.
grep -q "^$name," "$ADDRESS_BOOK": Checks if a record with the given name exists in the address book.
grep -v "^$name," "$ADDRESS_BOOK" > temp.txt: Writes all lines not matching the given name to a temporary file.
mv temp.txt "$ADDRESS_BOOK": Replaces the address book file with the temporary file, effectively deleting the record.
grep -v "^$name," "$ADDRESS_BOOK" > temp.txt: Removes the old record by writing all other lines to a temporary file.
echo "$name, $phone, $email" >> temp.txt: Appends the modified record to the temporary file.
mv temp.txt "$ADDRESS_BOOK": Replaces the address book file with the updated temporary file.

- while true; do: Infinite loop to keep displaying the menu until the user chooses to exit.

- case $choice in ... esac: Selects the appropriate function based on the user's choice.
- exit 0: Exits the script.

Summary:

- **File Operations**: Checking for file existence (-f), creating a file (touch), displaying file contents (cat), and modifying file contents (grep, mv).
- **User Input**: Using read to get input from the user.
- **Control Structures**: if statements for conditional checks and case statements for menu selection.
- **String Manipulation**: Using echo and grep for handling text and patterns.

Assignment No. 2A

Aim: Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.Implement the C program in which the main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using the WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

Theory:
Implementation Details:
PART A:
Algorithm:
1: START
2: Accept array of integers from the user
3: Call Merge_sort() to sort the array of integers
4: Call fork() to create child
5: Using wait system call, wait for child to finish execution
6: if (pid == 0) i.e. if child process, then call quick_sort() to sort array using quick sort.
7: END
Input:
1: Array of Integers
Steps to execute:
Step 1: Compile file using ―gcc program_name.c –o program_name.out‖
Step 2: Execute the program using ./program_name.out
Step 3: Open 2 new terminals and using command ps –e –o pid, ppid, stat, command show
various states of the processes that are created & terminated from the program (use sleep and
wait functions as per the requirement to show zombie and orphan states).

PART B:
Algorithm:
File1.c
1: START
2: Accept array of integers from the user
3: Sort the array of integers using any sorting technique

4: Convert the integer numbers into string using sprintf(char * str, const char * format, ...)
5: Call the fork() function to create child process.
6: Using wait system call, wait for child to finish execution.
7: Call the execve() system call, and pass the second program name and the string converted
array as parameters to the execve system call.
8: END

File2.c
1: START
2: Convert the received string into integer array using atoi() function.
3: Ask the user which number to find
4: Call binary search function to search the particular number.
5: Return the location of the number if found, else print error.
6: END
Input:
1: Array of Integers
2: Number to find

Steps to execute:
Step 1: Compile file1 & file2 using ―gcc program_name.c –o program_name.out‖
Step 2: Execute the program using ―./file1.out file2.out

Theory:

System Calls:

1. **FORK**: Creates a new process by duplicating the calling process. The new process is called the child process.
2. **EXECVE**: Replaces the current process image with a new process image.
3. **WAIT**: Makes the parent process wait until all of its child processes have terminated.

Process States:

- **Zombie State**: When a child process has completed execution but its entry still exists in the process table.
- **Orphan State**: When a parent process terminates but its child processes are still running, the child processes become orphaned.

Implementation Details:

PART A:

Algorithm:

1. **START**
2. **Accept array of integers from the user**
3. **Call Merge_sort() to sort the array of integers**
4. **Call fork() to create a child process**
5. **Using wait system call, wait for the child to finish execution**
6. **If (pid == 0) i.e., if child process, then call quick_sort() to sort the array using quick sort**
7. **END**

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

// Function prototypes
void merge_sort(int arr[], int l, int r);
void merge(int arr[], int l, int m, int r);
void quick_sort(int arr[], int low, int high);
int partition(int arr[], int low, int high);
void print_array(int arr[], int size);

int main() {
   int n;
   printf("Enter the number of integers to sort: ");
   scanf("%d", &n);

   int arr[n];
```

```
    printf("Enter the integers: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Parent process sorting using merge sort
    merge_sort(arr, 0, n - 1);

    // Fork a child process
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process sorting using quick sort
        quick_sort(arr, 0, n - 1);
        printf("Child Process: Sorted array using Quick Sort: ");
        print_array(arr, n);
        exit(0);
    } else {
        // Parent process waits for the child process to finish
        wait(NULL);
        printf("Parent Process: Sorted array using Merge Sort: ");
        print_array(arr, n);
    }

    return 0;
}
```

```
// Merge Sort function
void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Merge function for Merge Sort
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
```

```
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Quick Sort function
void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

// Partition function for Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
```

```
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

// Function to print the array
void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Steps to Execute:

1. Compile the file using: gcc program_name.c -o program_name.out
2. Execute the program using: ./program_name.out
3. Open 2 new terminals and use the command ps -e -o pid,ppid,stat,command to show various states of the processes that are created & terminated from the program (use sleep and wait functions as required to show zombie and orphan states).

Modify the Part1.c to demonstrate zombie state:

- **To create a zombie process, make the child process exit while the parent process sleeps for a while before calling wait().**

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>


void merge_sort(int arr[], int l, int r);

void merge(int arr[], int l, int m, int r);

void quick_sort(int arr[], int low, int high);

int partition(int arr[], int low, int high);
```

```c
void print_array(int arr[], int size);


int main() {
    int n;
    printf("Enter the number of integers to sort: ");
    scanf("%d", &n);


    int arr[n];
    printf("Enter the integers: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }


    // Fork a child process
    pid_t pid = fork();


    if (pid < 0) {
        perror("Fork failed");
        return 1;
```

```
    } else if (pid == 0) {
        // Child process sorting using quick sort
        quick_sort(arr, 0, n - 1);
        printf("Child Process: Sorted array using Quick Sort: ");
        print_array(arr, n);
        exit(0); // Child process exits
    } else {
        // Parent process sleeps for a while to let the child process
become a zombie
        sleep(10);  // Sleep long enough to observe the zombie
state
        // Now parent process waits for the child process to finish
        wait(NULL);
        printf("Parent Process: Child has finished.\n");
    }

    return 0;
}

void merge_sort(int arr[], int l, int r) {
```

```
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
```

```
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
```

```
        arr[k] = R[j];
        j++;
        k++;
    }
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
```

```
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
```

```
        printf("\n");

    }
```

Modify the Part1.c to demonstrate orphan state:

- **To create an orphan process, make the parent process exit before the child process finishes execution.**

```
#include <stdio.h>

#include <stdlib.h>
```

```c
#include <unistd.h>
#include <sys/wait.h>

void merge_sort(int arr[], int l, int r);
void merge(int arr[], int l, int m, int r);
void quick_sort(int arr[], int low, int high);
int partition(int arr[], int low, int high);
void print_array(int arr[], int size);

int main() {
    int n;
    printf("Enter the number of integers to sort: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the integers: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
```

```
    // Fork a child process
    pid_t pid = fork();


    if (pid < 0) {
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process sorting using quick sort
        quick_sort(arr, 0, n - 1);
        printf("Child Process: Sorted array using Quick Sort: ");
        print_array(arr, n);
        sleep(50); // Sleep to allow the parent process to
terminate
        printf("Child Process: Orphaned, adopted by init
process.\n");
        exit(0); // Child process exits
    } else {
        // Parent process exits before the child process finishes
```

```
        printf("Parent Process: Exiting before child process
finishes.\n");
        exit(0); // Parent exits, making the child process an orphan
    }

    return 0;
}

void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
```

```
int L[n1], R[n2];


for (int i = 0; i < n1; i++)
    L[i] = arr[l + i];
for (int j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];


int i = 0, j = 0, k = l;


while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

```
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
```

```
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
```

```
    return (i + 1);
}

void print_array(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

—---##################
#-------------------#############

Steps to Execute and Observe Zombie and Orphan States:

gcc Part1.c -o Part1.out

./Part1.out

Enter the number of integers and the integers when prompted. Do not press Enter after inputting the integers immediately to give you time to observe the states.

Expected Output:

Enter the number of integers to sort: 6

Enter the integers: 5 2 9 1 5 6

Before the parent calls wait: Open another terminal and run:

ps -e -o pid,ppid,stat,command | grep 'Part1.out'

You should see the child process in a zombie state (marked as Z):

1234 1230 Z ./Part1.out

After the parent calls wait:

Parent Process: Child has finished.

PART B:

Algorithm for File1.c:

1. **START**
2. **Accept array of integers from the user**
3. **Sort the array of integers using any sorting technique**
4. **Convert the integer numbers into string using sprintf(char *str, const char *format, ...)**
5. **Call the fork() function to create a child process**
6. **Using wait system call, wait for the child to finish execution**
7. **Call the execve() system call, and pass the second program name and the string converted array as parameters to the execve system call**
8. **END**

Code for File1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

void bubble_sort(int arr[], int n);
void convert_array_to_string(int arr[], int n, char *str);

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

```
int arr[n];
printf("Enter the integers: ");
for (int i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
}

// Sorting the array
bubble_sort(arr, n);

// Convert sorted array to string
char str[100];
convert_array_to_string(arr, n, str);

// Fork a child process
pid_t pid = fork();

if (pid < 0) {
    perror("Fork failed");
    return 1;
} else if (pid == 0) {
    // Child process
    char *args[] = {"./file2.out", str, NULL};
    execve(args[0], args, NULL);
    perror("execve failed");
    exit(1);
} else {
    // Parent process waits for the child process to finish
    wait(NULL);
}
```

```c
    return 0;
}

void bubble_sort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

void convert_array_to_string(int arr[], int n, char *str) {
    char temp[10];
    str[0] = '\0';
    for (int i = 0; i < n; i++) {
        sprintf(temp, "%d ", arr[i]);
        strcat(str, temp);
    }
}
```

Algorithm for File2.c:

1. **START**
2. **Convert the received string into integer array using atoi() function**
3. **Ask the user which number to find**
4. **Call binary search function to search the particular number**
5. **Return the location of the number if found, else print error**
6. **END**

Code for File2.c:

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

void convert_string_to_array(char *str, int arr[], int *n);
int binary_search(int arr[], int l, int r, int x);

int main(int argc, char *argv[]) {
   if (argc < 2) {
      printf("Usage: %s <sorted_numbers>\n", argv[0]);
      return 1;
   }

   int n;
   int arr[100];
   convert_string_to_array(argv[1], arr, &n);

   int x;
   printf("Enter the number to find: ");
   scanf("%d", &x);
```

```
    int result = binary_search(arr, 0, n-1, x);
    if (result == -1) {
        printf("Element not found\n");
    } else {
        printf("Element found at index %d\n", result);
    }

    return 0;
}

void convert_string_to_array(char *str, int arr[], int *n) {
    char *token = strtok(str, " ");
    *n = 0;
    while (token != NULL) {
        arr[*n] = atoi(token);
        (*n)++;
        token = strtok(NULL, " ");
    }
}

int binary_search(int arr[], int l, int r, int x) {
    while (l <= r) {
        int m = l + (r - l) / 2;

        if (arr[m] == x) {
            return m;
        }

        if (arr[m] < x) {
```

```
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return -1;
}
```

Steps to Execute:

1. Compile the files using: gcc file1.c -o file1.out and gcc file2.c -o file2.out
2. Execute the first program using: ./file1.out
3. Follow the prompts to input the array and search number. The second program will be called via execve and perform the binary search.

—----------------------------------#########################---

PART A: Output Demonstration

Let's assume we input the array [5, 2, 9, 1, 5, 6].

Execution Steps:

Compile and run the program:

gcc partA.c -o partA.out
./partA.out

Program output:

Enter the number of integers to sort: 6
Enter the integers: 5 2 9 1 5 6
Child Process: Sorted array using Quick Sort: 1 2 5 5 6 9
Parent Process: Sorted array using Merge Sort: 1 2 5 5 6 9

Demonstrating Zombie and Orphan States:

Open a new terminal and run the following command to show the process states:

ps -e -o pid,ppid,stat,command

You may see output like this showing the parent and child processes:

```
PID   PPID  STAT  COMMAND
1234  1220  S     ./partA.out
1235  1234  Z     [partA.out] <defunct>
```

Explanation: The child process becomes a zombie after it finishes execution and before the parent calls wait(). After the parent calls wait(), the child process is removed from the process table.

PART B: Output Demonstration

File1.c Output:

Compile and run the program:

```
gcc file1.c -o file1.out
gcc file2.c -o file2.out
./file1.out
```

Program output:

Enter the number of integers to sort: 6
Enter the integers: 5 2 9 1 5 6
Enter the number to find: 9

File2.c Output:

After the execve call, the second program (file2.out) runs:

Element found at index 5

Explanation: The file1.out sorts the array and passes it to file2.out using the execve call. The file2.out then performs a binary search on the sorted array.

Full Example Output:

Terminal 1 (file1.c):

```
gcc file1.c -o file1.out
gcc file2.c -o file2.out
./file1.out
Enter the number of integers to sort: 6
Enter the integers: 5 2 9 1 5 6
```

Terminal 2 (file2.c after execve):

```
Enter the number to find: 9
Element found at index 5
```

Process States:

```
ps -e -o pid,ppid,stat,command
PID   PPID  STAT  COMMAND
1234  1220  S     ./file1.out
1235  1234  S     ./file2.out
```

This demonstration provides a clear picture of how the processes are created and managed, including showing zombie and orphan states in the process table.

Assignment No. 2B

Aim: Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.

Algorithm

1. **Input Array**: The main program accepts an array from the user.
2. **Fork Process**: The main program uses the fork() system call to create a new process.
   - **Parent Process**:
     - Waits for the child process to terminate.
     - Sorts the array.

        - Converts the sorted array to strings and prepares command line arguments.
        - Uses execve() to execute the child process with the sorted array as arguments.
    - **Child Process**:
        - Loads a new program using execve().
        - This new program reverses and displays the array.

Code Description

1. **Parent Process**:
    - Sorts the input array.
    - Converts the sorted array elements to strings.
    - Prepares the arguments for execve().
    - Calls execve() to execute the child program with the sorted array.
2. **Child Process**:
    - Receives the sorted array as command line arguments.
    - Reverses the array.
    - Displays the reversed array.

C Program

Main Program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

void sort(int arr[], int n) {
    int temp;
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

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

    int arr[n];
    printf("Enter the elements:\n");
    for(int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
```

```
    pid_t pid = fork();

    if(pid < 0) {
        perror("Fork failed");
        exit(1);
    } else if(pid == 0) {
        // Child process
        char *args[n+2];
        args[0] = "./child";  // Assuming the child program is named
"child"
        for(int i = 0; i < n; i++) {
            char *num_str = (char *)malloc(12);
            sprintf(num_str, "%d", arr[i]);
            args[i+1] = num_str;
        }
        args[n+1] = NULL;

        execve(args[0], args, NULL);
        perror("execve failed");
        exit(1);
    } else {
        // Parent process
        wait(NULL);
        sort(arr, n);
        printf("Sorted array: ");
        for(int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
```

```
        // Convert sorted array to string arguments for execve
        char *args[n+2];
        args[0] = "./child";
        for(int i = 0; i < n; i++) {
            char *num_str = (char *)malloc(12);
            sprintf(num_str, "%d", arr[i]);
            args[i+1] = num_str;
        }
        args[n+1] = NULL;

        execve(args[0], args, NULL);
        perror("execve failed");
        exit(1);
    }

    return 0;
}
```

Child Program:

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

int main(int argc, char *argv[]) {
    int n = argc - 1;
```

```
    int arr[n];

    for(int i = 1; i < argc; i++) {
        arr[i-1] = atoi(argv[i]);
    }

    printf("Reversed array: ");
    for(int i = n-1; i >= 0; i--) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Execution Process

1. **Compile the Programs**:

```
gcc -o main main.c
gcc -o child child.c
```

Run the Main Program:

```
./main
```

1. **Input**:
   - Number of elements: 5
   - Elements: 3, 1, 4, 1, 5
2. **Output**:
   - Parent process sorts the array and prints: Sorted array: 1 1 3 4 5
   - Child process reverses and prints: Reversed array: 5 4 3 1 1

Explanation

- The main program receives the array and forks a new process.
- The parent process sorts the array and uses execve() to pass the sorted array to the child program.
- The child program receives the sorted array and prints it in reverse order.

Assignment No. 3

Aim: Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

Explanation and Execution Steps for Shortest Job First (Preemptive)

1. **Structure Definition**: Define a structure Process to store details of each process.
2. **Input Process Details**: Input the number of processes and their arrival and burst times.
3. **Calculate Waiting Time**: Implement the logic to calculate the waiting time of each process by continuously selecting the process with the shortest remaining time.
4. **Calculate Turnaround Time**: Calculate the turnaround time using waiting time and burst time.
5. **Print Results**: Display the process details, including completion time, waiting time, and turnaround time.

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

// Structure to represent a process
struct Process {
    int pid; // Process ID
    int arrival_time; // Arrival time
    int burst_time; // Burst time
    int remaining_time; // Remaining burst time
    int completion_time; // Completion time
    int waiting_time; // Waiting time
    int turnaround_time; // Turnaround time
};

void calculateWaitingTime(struct Process proc[], int n) {
    int time = 0; // Current time
    int completed = 0; // Number of completed processes
    bool is_completed[n]; // To check if process is completed
    for (int i = 0; i < n; i++) is_completed[i] = false;

    while (completed != n) {
        int min_index = -1;
```

```
        int min_time = 1000000; // A large value to start with

        // Find the process with the minimum remaining time
        for (int i = 0; i < n; i++) {
            if (proc[i].arrival_time <= time && !is_completed[i] &&
    proc[i].remaining_time < min_time) {
                min_time = proc[i].remaining_time;
                min_index = i;
            }
        }

        if (min_index != -1) {
            proc[min_index].remaining_time--;
            time++;

            // If a process is completed
            if (proc[min_index].remaining_time == 0) {
                proc[min_index].completion_time = time;
                proc[min_index].turnaround_time =
    proc[min_index].completion_time - proc[min_index].arrival_time;
                proc[min_index].waiting_time =
    proc[min_index].turnaround_time - proc[min_index].burst_time;
                is_completed[min_index] = true;
                completed++;
            }
        } else {
            time++;
        }
    }
}
```

```
void calculateTurnaroundTime(struct Process proc[], int n) {
   for (int i = 0; i < n; i++) {
      proc[i].turnaround_time = proc[i].burst_time +
proc[i].waiting_time;
   }
}

void printResults(struct Process proc[], int n) {
   printf("PID\tArrival Time\tBurst Time\tCompletion
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\t\t%d\n", proc[i].pid,
proc[i].arrival_time, proc[i].burst_time, proc[i].completion_time,
proc[i].waiting_time, proc[i].turnaround_time);
   }
}

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

   struct Process proc[n];

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

```
        proc[i].remaining_time = proc[i].burst_time;
    }

    calculateWaitingTime(proc, n);
    calculateTurnaroundTime(proc, n);
    printResults(proc, n);
    return 0;
}
```

Compile the Program: Use a C compiler to compile the program.

```
gcc -o scheduling scheduling.c
```

Run the Program: Execute the compiled program.

```
./scheduling
```

Input Process Details: Follow the prompts to enter the arrival and burst times for each process.

View Results: The program will display the scheduling results, including completion time, waiting time, and turnaround time for each process.

Explanation and Execution Steps for Round Robin

Round Robin Scheduling Algorithm

The Round Robin (RR) scheduling algorithm is one of the simplest and most widely used scheduling algorithms in operating systems. It assigns a fixed time unit per process, called a time quantum, and cycles through the processes in a round-robin fashion. Each process is executed for a time slice, and if it doesn't finish within that time, it is put back in the queue and the next process is picked up.

Key Points of Round Robin Scheduling with Different Arrival Times:

1. **Time Quantum**: Each process gets executed for a fixed amount of time, called the time quantum.
2. **Arrival Time**: Processes arrive at different times.
3. **Waiting Time**: The total time a process spends in the ready queue before it gets executed.
4. **Turnaround Time**: The total time taken from the arrival of the process to its completion.

Steps for Execution:

- **Input Processes**: Number of processes, their arrival times, burst times, and the time quantum.
- **Sort by Arrival Time**: Initially sort the processes based on their arrival time.
- **Simulate Round Robin**: Use a queue to manage process execution. Track remaining burst times.
- **Calculate Waiting and Turnaround Time**: As each process finishes, calculate its waiting and turnaround time.
- **Output**: Print the waiting time and turnaround time for each process along with the average waiting time and turnaround time.

- **Structure Definition**: Define a structure Process to store details of each process.
- **Input Process Details**: Input the number of processes and their arrival and burst times.
- **Input Time Quantum**: Input the time quantum for the round-robin algorithm.
- **Calculate Times**: Implement the logic to calculate the waiting and turnaround times by using a queue to maintain the order of processes.
- **Print Results**: Display the process details, including completion time, waiting time, and turnaround time.

```
// Structure to represent a process
struct Process {
    int pid; // Process ID
    int arrival_time; // Arrival time
    int burst_time; // Burst time
    int remaining_time; // Remaining burst time
    int completion_time; // Completion time
    int waiting_time; // Waiting time
    int turnaround_time; // Turnaround time
};

void calculateTimes(struct Process proc[], int n, int quantum) {
    int time = 0; // Current time
    int completed = 0; // Number of completed processes
    int queue[n]; // Queue to maintain process order
    int front = 0, rear = 0; // Queue pointers
    int in_queue[n]; // To check if process is in queue
```

```
    for (int i = 0; i < n; i++) in_queue[i] = 0;

    queue[rear++] = 0; // Start with the first process
    in_queue[0] = 1;

    while (completed != n) {
        int index = queue[front++];

        if (proc[index].remaining_time <= quantum &&
proc[index].remaining_time > 0) {
            time += proc[index].remaining_time;
            proc[index].remaining_time = 0;
            completed++;
            proc[index].completion_time = time;
            proc[index].turnaround_time = proc[index].completion_time -
proc[index].arrival_time;
            proc[index].waiting_time = proc[index].turnaround_time -
proc[index].burst_time;
        } else if (proc[index].remaining_time > 0) {
            time += quantum;
            proc[index].remaining_time -= quantum;
        }

        for (int i = 0; i < n; i++) {
            if (i != index && proc[i].arrival_time <= time && !in_queue[i]
&& proc[i].remaining_time > 0) {
                queue[rear++] = i;
                in_queue[i] = 1;
            }
        }
```

```
        if (proc[index].remaining_time > 0) {
            queue[rear++] = index;
        }
    }
}

void printResults(struct Process proc[], int n) {
    printf("PID\tArrival Time\tBurst Time\tCompletion
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\t\t%d\n", proc[i].pid,
proc[i].arrival_time, proc[i].burst_time, proc[i].completion_time,
proc[i].waiting_time, proc[i].turnaround_time);
    }
}

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

    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", i +
1);
        scanf("%d%d", &proc[i].arrival_time, &proc[i].burst_time);
        proc[i].remaining_time = proc[i].burst_time;
```

```
    }

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

    calculateTimes(proc, n, quantum);
    printResults(proc, n);

    return 0;
}
```

Execution Steps for Both Programs

Steps to Execute the Program Round Robin:

Compile the Program:

gcc round_robin.c -o round_robin

Run the Program:

./round_robin

Input:

- Enter the number of processes.
- Enter the arrival time and burst time for each process.
- Enter the time quantum.

Output:

- The program will display the waiting time and turnaround time for each process.
- The average waiting time and turnaround time will also be displayed.

Enter the number of processes: 3
Enter arrival time and burst time for process 1: 0 5
Enter arrival time and burst time for process 2: 1 3
Enter arrival time and burst time for process 3: 2 8
Enter the time quantum: 2

PID	Arrival	Burst	Waiting	Turnaround
1	0	5	8	13
2	1	3	5	8
3	2	8	12	20

Average Waiting Time: 8.33
Average Turnaround Time: 13.67

Assignment No. 4A

Aim: Thread synchronization using counting semaphores. Application to demonstrate: producer- consumer problem with counting semaphores and mutex.

Thread Synchronization with Counting Semaphores: Producer-Consumer Problem

Thread synchronization is crucial in concurrent programming to ensure that multiple threads or processes can work together without interfering with each other. One classic synchronization problem is the **Producer-Consumer Problem**, which involves coordinating access to a shared buffer by producer and consumer threads.

Key Concepts

1. **Threads and Processes**:
   - A **thread** is the smallest unit of execution in a program, while a **process** is a program in execution.
   - Multiple threads can exist within a single process, sharing the same resources.
2. **Critical Section**:
   - A critical section is a part of the code where shared resources are accessed. Only one thread should execute in a critical section at a time to avoid data corruption.
3. **Mutual Exclusion (Mutex)**:
   - A mutex is a synchronization primitive used to protect critical sections. It ensures that only one thread can access the critical section at a time.
4. **Semaphores**:
   - A semaphore is a signaling mechanism used to control access to a shared resource. It can be binary (0 or 1, similar to a mutex) or a counting semaphore, which allows a certain number of threads to access the resource concurrently.

Counting Semaphores

A counting semaphore is represented by an integer value and two atomic operations: **wait (P)** and **signal (V)**.

- **wait (P)**: Decrements the semaphore's value. If the value becomes negative, the thread is blocked until the value is positive.
- **signal (V)**: Increments the semaphore's value. If there are blocked threads, one of them is unblocked.

Producer-Consumer Problem

The Producer-Consumer Problem involves two types of threads:

- **Producers**: Generate data and place it into a shared buffer.
- **Consumers**: Remove data from the buffer and process it.

The challenge is to ensure that:

1. Producers do not add data when the buffer is full.
2. Consumers do not remove data when the buffer is empty.

This requires synchronization to prevent data corruption and to manage the buffer's contents correctly.

Implementation with Counting Semaphores and Mutex

We use three synchronization primitives:

1. **Mutex (mutex)**: Ensures mutual exclusion when accessing the buffer.

2. **Empty (empty)**: Counting semaphore indicating the number of empty slots in the buffer.
3. **Full (full)**: Counting semaphore indicating the number of full slots in the buffer.

Initialization:

```
int buffer[MAX]; // Shared buffer
int in = 0, out = 0; // Index for producer and consumer
int item; // Data item to be produced or consumed

sem_t empty = MAX; // Number of empty slots in the buffer
sem_t full = 0; // Number of full slots in the buffer
sem_t mutex = 1; // Mutual exclusion lock
```

Producer:

```
void producer() {
   while (true) {
      // Produce an item
      item = produce_item();
```

```
        // Wait if buffer is full
        sem_wait(&empty);

        // Enter critical section
        sem_wait(&mutex);

        // Add the item to the buffer
        buffer[in] = item;
        in = (in + 1) % MAX;

        // Exit critical section
        sem_post(&mutex);

        // Signal that buffer is not empty
        sem_post(&full);
    }
}
```

Consumer:

```
void consumer() {
    while (true) {
        // Wait if buffer is empty
        sem_wait(&full);

        // Enter critical section
```

```
        sem_wait(&mutex);

        // Remove the item from the buffer
        item = buffer[out];
        out = (out + 1) % MAX;

        // Exit critical section
        sem_post(&mutex);

        // Signal that buffer is not full
        sem_post(&empty);

        // Consume the item
        consume_item(item);
    }
}
```

Explanation

1. **Mutex (mutex)**:
    - Used to protect the critical sections where the buffer is accessed.
    - Ensures that only one producer or consumer accesses the buffer at a time.
2. **Empty and Full Semaphores**:
    - empty tracks the number of empty slots in the buffer.
    - full tracks the number of full slots in the buffer.
3. **Producer Process**:
    - The producer first waits on the empty semaphore, ensuring there is space in the buffer.

    - The producer then waits on the mutex to enter the critical section, adds an item to the buffer, and updates the in index.
    - The producer signals the mutex to release the critical section and signals the full semaphore to indicate a new item is available.
4. **Consumer Process**:
    - The consumer first waits on the full semaphore, ensuring there is an item to consume.
    - The consumer then waits on the mutex to enter the critical section, removes an item from the buffer, and updates the out index.
    - The consumer signals the mutex to release the critical section and signals the empty semaphore to indicate a slot is available.

Practical Considerations

1. **Buffer Size**:
    - The buffer size should be chosen based on the application's requirements and available resources.
2. **Starvation and Deadlock**:
    - Proper design of the synchronization mechanism ensures that neither producers nor consumers get indefinitely blocked.
3. **Performance**:
    - The use of semaphores and mutexes can introduce overhead, so it's essential to balance synchronization and performance.

To execute the Producer-Consumer problem code with counting semaphores and mutex, you can follow these steps:

1. **Setup the Environment**:
    - Ensure you have a C/C++ compiler installed, such as GCC (GNU Compiler Collection).
    - Make sure you have the necessary libraries for working with threads and semaphores. On Linux, these are typically part of the standard C library.
2. **Write the Code**:
    - Create a C file (e.g., `producer_consumer.c`) and write the implementation of the Producer-Consumer problem using the provided pseudo code.
3. **Compile the Code**:
    - Use a C compiler to compile the code. You may need to link with the pthread library for thread support.
4. **Run the Executable**:
    - Execute the compiled program to see the output.

Here's a detailed example of how to proceed:

1. Setting Up the Environment

Make sure your system has GCC installed. On a Linux system, you can check if GCC is installed by running:

```
gcc --version
```

If it's not installed, you can install it using your package manager, for example:

```
sudo apt-get install build-essential
```

2. Writing the Code

Create a file named producer_consumer.c and add the following code:

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

#define MAX 5 // Maximum size of the buffer

int buffer[MAX]; // Shared buffer
int in = 0, out = 0; // Index for producer and consumer

sem_t empty; // Semaphore for empty slots
sem_t full; // Semaphore for full slots
sem_t mutex; // Mutex for critical section

void* producer(void* arg) {
   int item;
   while (1) {
      // Produce an item
      item = rand() % 100;
      printf("Produced: %d\n", item);
```

```
        // Wait if buffer is full
        sem_wait(&empty);

        // Enter critical section
        sem_wait(&mutex);

        // Add the item to the buffer
        buffer[in] = item;
        in = (in + 1) % MAX;

        // Exit critical section
        sem_post(&mutex);

        // Signal that buffer is not empty
        sem_post(&full);

        sleep(1); // Sleep to simulate time taken to produce an item
    }
    return NULL;
}

void* consumer(void* arg) {
    int item;
    while (1) {
        // Wait if buffer is empty
        sem_wait(&full);

        // Enter critical section
        sem_wait(&mutex);
```

```
        // Remove the item from the buffer
        item = buffer[out];
        out = (out + 1) % MAX;

        // Exit critical section
        sem_post(&mutex);

        // Signal that buffer is not full
        sem_post(&empty);

        printf("Consumed: %d\n", item);

        sleep(1); // Sleep to simulate time taken to consume an item
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Initialize semaphores
    sem_init(&empty, 0, MAX); // MAX empty slots
    sem_init(&full, 0, 0); // 0 full slots
    sem_init(&mutex, 0, 1); // 1 mutex

    // Create producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Wait for the threads to finish (they won't, in this infinite loop)
```

```
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    // Destroy the semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

3. Compiling the Code

Use the GCC compiler to compile the program. You'll need to link the pthread library, which provides support for multithreading.

```
gcc -o producer_consumer producer_consumer.c -lpthread
```

Here, -o producer_consumer specifies the output executable's name, and -lpthread links the pthread library.

4. Running the Executable

Run the compiled program:

```
./producer_consumer
```

You should see output similar to:

```
Produced: 42
Consumed: 42
Produced: 56
Consumed: 56
...
```

Understanding the Code

- **Headers**: The necessary headers (<stdio.h>, <stdlib.h>, <pthread.h>, and <semaphore.h>) are included for input/output, standard library functions, thread, and semaphore functionalities.
- **Buffer**: A shared buffer array buffer[MAX] is used to store produced items.
- **Semaphores**: empty, full, and mutex are semaphores used for synchronization.
  - empty starts at MAX, indicating all slots are empty initially.
  - full starts at 0, indicating no slots are full initially.
  - mutex ensures that only one thread can enter the critical section at a time.
- **Producer and Consumer Functions**: These are implemented as infinite loops, producing and consuming items while managing buffer access using semaphores.
- **Main Function**: Initializes the semaphores and creates the producer and consumer threads.

Notes

- **Infinite Loops**: Both producer and consumer functions have infinite loops. In a real-world application, you would have some condition to exit these loops.
- **Synchronization**: Proper synchronization using semaphores ensures that the producer does not overwrite a full buffer and the consumer does not consume from an empty buffer.
- **Random Number Generation**: The producer generates random numbers as items. You can modify this to suit your specific application needs.
- **Error Handling**: The above code does not include error handling for simplicity. In production code, always check the return values of system calls and handle errors appropriately.

Assignment No. 4B

Aim: Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.

Thread Synchronization and Mutual Exclusion using Mutex: Reader-Writer Problem with Reader Priority

The **Reader-Writer Problem** is a classic synchronization problem that involves a shared resource (such as a file or a database) accessed by multiple threads. The threads are categorized into readers, who only read the data, and writers, who modify the data. The problem requires designing a synchronization mechanism to prevent data inconsistencies and ensure safe access to the shared resource.

In the **Reader-Writer Problem with Reader Priority**, readers are given priority over writers. This means that if readers are currently accessing the resource or waiting, writers must wait until all readers are done.

Key Concepts

1. **Mutual Exclusion (Mutex)**:
   - A mutex is used to ensure that only one thread can access a critical section at a time, preventing data corruption.
2. **Reader Priority**:
   - Readers are given priority over writers. Writers can only proceed when there are no active readers.
3. **Critical Section**:
   - The part of the code that accesses the shared resource. Proper synchronization is needed to avoid concurrent access issues.

Implementation Details

We use three key synchronization primitives:

1. **Mutex (mutex)**:
   - Ensures mutual exclusion when accessing the shared resource or updating shared state variables.
2. **Semaphore (rw_mutex)**:
   - Controls access to the shared resource. Writers wait on this semaphore to gain exclusive access.
3. **Semaphore (read_count_mutex)**:
   - Protects the read_count variable, which tracks the number of active readers.

Initialization:

```
int read_count = 0; // Number of active readers
sem_t rw_mutex; // Semaphore for read/write access
sem_t read_count_mutex; // Semaphore to protect read_count
```

Reader:

```
void reader() {
   while (true) {
      // Enter critical section to update read_count
      sem_wait(&read_count_mutex);
      read_count++;
```

```
        if (read_count == 1) {
            sem_wait(&rw_mutex); // First reader locks the resource
        }
        sem_post(&read_count_mutex);

        // Reading the resource
        read_resource();

        // Exit critical section
        sem_wait(&read_count_mutex);
        read_count--;
        if (read_count == 0) {
            sem_post(&rw_mutex); // Last reader unlocks the resource
        }
        sem_post(&read_count_mutex);

        sleep(1); // Simulate reading time
    }
}
```

Writer:

```
void writer() {
    while (true) {
        // Wait to get exclusive access to the resource
        sem_wait(&rw_mutex);

        // Writing to the resource
        write_resource();
```

```
        // Release the resource
        sem_post(&rw_mutex);

        sleep(1); // Simulate writing time
    }
}
```

Steps to Execute the Program

1. **Setup the Environment**:
   - Ensure you have a C/C++ compiler and necessary libraries for threads and semaphores.
2. **Write the Code**:
   - Create a C file (e.g., `reader_writer.c`) with the implementation.
3. **Compile the Code**:
   - Compile the code using a C compiler and link with the pthread library.
4. **Run the Executable**:
   - Execute the compiled program.

Here's the detailed implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
```

```
int read_count = 0; // Number of active readers
sem_t rw_mutex; // Semaphore for read/write access
sem_t read_count_mutex; // Semaphore to protect read_count

void* reader(void* arg) {
    while (1) {
        // Enter critical section to update read_count
        sem_wait(&read_count_mutex);
        read_count++;
        if (read_count == 1) {
            sem_wait(&rw_mutex); // First reader locks the resource
        }
        sem_post(&read_count_mutex);

        // Reading the resource
        printf("Reader %d is reading\n", *((int*)arg));

        // Exit critical section
        sem_wait(&read_count_mutex);
        read_count--;
        if (read_count == 0) {
            sem_post(&rw_mutex); // Last reader unlocks the resource
        }
        sem_post(&read_count_mutex);

        sleep(1); // Simulate reading time
    }
    return NULL;
}
```

```
void* writer(void* arg) {
    while (1) {
        // Wait to get exclusive access to the resource
        sem_wait(&rw_mutex);

        // Writing to the resource
        printf("Writer %d is writing\n", *((int*)arg));

        // Release the resource
        sem_post(&rw_mutex);

        sleep(1); // Simulate writing time
    }
    return NULL;
}

int main() {
    pthread_t readers[5], writers[2];
    int reader_ids[5] = {1, 2, 3, 4, 5};
    int writer_ids[2] = {1, 2};

    // Initialize semaphores
    sem_init(&rw_mutex, 0, 1); // Initialize rw_mutex to 1 (available)
    sem_init(&read_count_mutex, 0, 1); // Initialize read_count_mutex
to 1

    // Create reader and writer threads
    for (int i = 0; i < 5; i++) {
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
```

```
    }
    for (int i = 0; i < 2; i++) {
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }

    // Join threads (they will never actually terminate)
    for (int i = 0; i < 5; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(writers[i], NULL);
    }

    // Destroy the semaphores
    sem_destroy(&rw_mutex);
    sem_destroy(&read_count_mutex);

    return 0;
}
```

Steps to Compile and Run the Code

1. **Write the Code**:
    - Save the above code in a file named reader_writer.c.
2. **Compile the Code**:
    - Use the GCC compiler to compile the program. The -lpthread flag is necessary to link the pthread library.
      gcc -o reader_writer reader_writer.c -lpthread

3. This command compiles the code and creates an executable named reader_writer.
4. **Run the Executable**:
   ○ Run the compiled program.

```
./reader_writer
```

You should see output similar to:

```
Reader 1 is reading
Writer 1 is writing
Reader 2 is reading
Writer 2 is writing
...
```

5.

Explanation of the Code

1. **Shared Variables**:
   ○ read_count: Tracks the number of active readers.
   ○ rw_mutex: Semaphore ensuring exclusive access to the shared resource.
   ○ read_count_mutex: Semaphore protecting the read_count variable.
2. **Reader Function**:
   ○ Waits on read_count_mutex to safely update read_count.
   ○ If the reader is the first, it locks rw_mutex to prevent writers from accessing the resource.
   ○ After reading, it decrements read_count and, if it is the last reader, releases rw_mutex.

3. **Writer Function**:
    - Waits on rw_mutex to gain exclusive access to the resource.
    - Writes to the resource and then releases rw_mutex.
4. **Main Function**:
    - Initializes semaphores and creates reader and writer threads.
    - Threads are created using pthread_create and joined using pthread_join, although they will run indefinitely due to the infinite loops in the reader and writer functions.

Output and Behavior

The output will show alternating messages from readers and writers indicating their operations. Due to the reader-priority approach, multiple readers may access the resource simultaneously, but writers must wait for all readers to finish before gaining access. This demonstrates the synchronization mechanism ensuring safe and prioritized access to the shared resource.

In an educational context, this example helps students understand how to implement synchronization mechanisms, handle concurrent access to shared resources, and the importance of prioritization in resource management.

Assignment No. 5

Aim: Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

The **Banker's Algorithm** is a classic algorithm in operating systems used to avoid deadlocks by managing resource allocation. It ensures that a system will remain in a safe state after resource allocation.

Below is a complete C program that implements the Banker's Algorithm. The program prompts the user to enter the number of processes, resources, the Available resources, Maximum resource demand per process, and the Allocation per process. It then computes the Need matrix and determines if the system is in a safe state. If it is, it outputs a safe sequence; otherwise, it indicates that no safe sequence exists.

Code:

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int main() {
   int n_processes, n_resources;
   int Allocation[MAX_PROCESSES][MAX_RESOURCES];
   int Maximum[MAX_PROCESSES][MAX_RESOURCES];
   int Need[MAX_PROCESSES][MAX_RESOURCES];
   int Available[MAX_RESOURCES];
   bool Finish[MAX_PROCESSES] = {false};
   int SafeSequence[MAX_PROCESSES];
   int count = 0;
```

```
    // Input number of processes and resources
    printf("Enter the number of processes (<= %d): ",
MAX_PROCESSES);
    scanf("%d", &n_processes);
    printf("Enter the number of resources (<= %d): ",
MAX_RESOURCES);
    scanf("%d", &n_resources);

    // Input Available vector
    printf("Enter the Available Resources:\n");
    for (int i = 0; i < n_resources; i++) {
        printf("Resource %d: ", i);
        scanf("%d", &Available[i]);
    }

    // Input Maximum matrix
    printf("Enter the Maximum Resource Matrix:\n");
    for (int i = 0; i < n_processes; i++) {
        printf("Process %d:\n", i);
        for (int j = 0; j < n_resources; j++) {
            printf("Resource %d: ", j);
            scanf("%d", &Maximum[i][j]);
        }
    }

    // Input Allocation matrix
    printf("Enter the Allocation Resource Matrix:\n");
    for (int i = 0; i < n_processes; i++) {
        printf("Process %d:\n", i);
        for (int j = 0; j < n_resources; j++) {
            printf("Resource %d: ", j);
            scanf("%d", &Allocation[i][j]);
        }
```

```
}

// Calculate Need matrix
for (int i = 0; i < n_processes; i++) {
    for (int j = 0; j < n_resources; j++) {
        Need[i][j] = Maximum[i][j] - Allocation[i][j];
    }
}

// Display Need matrix
printf("\nNeed Matrix:\n");
for (int i = 0; i < n_processes; i++) {
    printf("Process %d: ", i);
    for (int j = 0; j < n_resources; j++) {
        printf("%d ", Need[i][j]);
    }
    printf("\n");
}

// Banker's Algorithm
while (count < n_processes) {
    bool found = false;
    for (int i = 0; i < n_processes; i++) {
        if (!Finish[i]) {
            int j;
            for (j = 0; j < n_resources; j++) {
                if (Need[i][j] > Available[j])
                    break;
            }
            if (j == n_resources) {
                // The process's needs can be satisfied
                for (int k = 0; k < n_resources; k++)
                    Available[k] += Allocation[i][k];
                SafeSequence[count++] = i;
```

```
                Finish[i] = true;
                found = true;
            }
        }
    }
    if (!found) {
        printf("\nSystem is not in a safe state.\n");
        return 1;
    }
}

// Display Safe Sequence
printf("\nSystem is in a safe state.\nSafe Sequence is: ");
for (int i = 0; i < n_processes; i++) {
    printf("P%d", SafeSequence[i]);
    if (i != n_processes -1)
        printf(" -> ");
}
printf("\n");

return 0;
}
```

Detailed Explanation

1. **Definitions and Declarations:**
    - **Constants:**
        - MAX_PROCESSES: Maximum number of processes supported (set to 10).
        - MAX_RESOURCES: Maximum number of resources supported (set to 10).
    - **Variables:**
        - n_processes: Actual number of processes entered by the user.

        - n_resources: Actual number of resources entered by the user.
        - Allocation: 2D array representing resources currently allocated to each process.
        - Maximum: 2D array representing the maximum demand of each process.
        - Need: 2D array representing remaining resource needs of each process (Need = Maximum - Allocation).
        - Available: Array representing the number of available instances for each resource.
        - Finish: Boolean array to indicate if a process has completed (initially all false).
        - SafeSequence: Array to store the safe sequence of process execution.
        - count: Counter to track number of processes included in the safe sequence.
2. **User Input:**
    - **Number of Processes and Resources:**
        - The user is prompted to input the number of processes and resources, ensuring they don't exceed MAX_PROCESSES and MAX_RESOURCES.
    - **Available Resources Vector:**
        - The user inputs the total number of available instances for each resource type.
    - **Maximum Resource Matrix:**
        - For each process, the user inputs the maximum number of resources it may request.
    - **Allocation Matrix:**
        - For each process, the user inputs the number of resources currently allocated to it.
3. **Computing the Need Matrix:**
    - For each process and each resource, compute the Need as Maximum - Allocation.

4. **Displaying the Need Matrix:**
   - The program outputs the computed Need matrix for user verification.
5. **Banker's Algorithm Execution:**
   - **Initialization:**
     - count is set to 0, indicating no processes have been sequenced yet.
   - **Main Loop:**
     - The loop continues until all processes are sequenced (count < n_processes).
     - **Process Selection:**
       - For each process not yet finished, check if its needs can be satisfied with the current Available resources.
       - If yes, simulate allocation:
         - Update Available by adding back the allocated resources of the process (since it's assumed to finish).
         - Mark the process as finished in Finish.
         - Add the process to the SafeSequence.
         - Increment count.
       - If no such process is found in an iteration (i.e., no process can proceed), the system is in an unsafe state, and the program exits indicating so.
6. **Outputting the Safe Sequence:**
   - If the system is in a safe state, the program outputs the safe sequence of process execution.

Execution Steps in Linux Operating System

1. **Saving the Program:**
   - Open a text editor (like gedit, vim, or nano) and paste the

   above C program code.
   - Save the file as bankers_algorithm.c.
2. **Compiling the Program:**
   - Open the terminal and navigate to the directory containing bankers_algorithm.c.
   - Compile the program using gcc (GNU Compiler Collection) with the following command:

```
gcc bankers_algorithm.c -o bankers_algorithm
```

3. Running the Program:

- Execute the program using the following command:

```
./bankers_algorithm
```

Sample Input and Output:

- **User Input:**

```
Enter the number of processes (<= 10): 5
Enter the number of resources (<= 10): 3
Enter the Available Resources:
Resource 0: 3
Resource 1: 3
Resource 2: 2
Enter the Maximum Resource Matrix:
Process 0:
Resource 0: 7
Resource 1: 5
Resource 2: 3
Process 1:
Resource 0: 3
```

Resource 1: 2
Resource 2: 2
Process 2:
Resource 0: 9
Resource 1: 0
Resource 2: 2
Process 3:
Resource 0: 2
Resource 1: 2
Resource 2: 2
Process 4:
Resource 0: 4
Resource 1: 3
Resource 2: 3
Enter the Allocation Resource Matrix:
Process 0:
Resource 0: 0
Resource 1: 1
Resource 2: 0
Process 1:
Resource 0: 2
Resource 1: 0
Resource 2: 0
Process 2:
Resource 0: 3
Resource 1: 0
Resource 2: 2
Process 3:
Resource 0: 2
Resource 1: 1
Resource 2: 1
Process 4:
Resource 0: 0
Resource 1: 0

Resource 2: 2

Program Output:

Need Matrix:
Process 0: 7 4 3
Process 1: 1 2 2
Process 2: 6 0 0
Process 3: 0 1 1
Process 4: 4 3 1

System is in a safe state.
Safe Sequence is: P1 -> P3 -> P4 -> P0 -> P2

Assignment No-6

Aim: Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

Explanation of Page Replacement Algorithms

1. **First-Come-First-Serve (FCFS)**
   - **Concept: FCFS is the simplest page replacement algorithm. Pages are replaced in the order they arrived. If a page fault occurs and there is no free frame, the page at the oldest position (first inserted) is replaced.**
   - **Implementation: Maintain a queue where pages are added when they are referenced. If a page is not found in the queue and all frames are full, the oldest page (first-in) is removed and replaced with the new page.**
2. **Least Recently Used (LRU)**
   - **Concept: LRU replaces the page that has not been used for the longest period of time. It uses the principle that the most recently used pages will likely be used again soon.**
   - **Implementation: Keep track of the time of last use for each page. When a page fault occurs and all frames are full, replace the page with the smallest last-used time.**
3. **Optimal**
   - **Concept: The Optimal page replacement algorithm replaces the page that will not be used for the longest period of time in the future. It is considered the best page replacement algorithm but is not implementable in practice as it requires future knowledge of page references.**

- Implementation: For each page fault, find the page in the frame that will not be used for the longest time in the future and replace it with the new page.

Below is a detailed C program that demonstrates the three page replacement algorithms: First-Come-First-Serve (FCFS), Least Recently Used (LRU), and Optimal. Each algorithm is implemented with a frame size of at least three. Following the code, I'll provide explanations for each algorithm.

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

#define FRAME_SIZE 3  // Define the frame size for the
algorithms

// Function to print the current page frame
void printFrame(int frame[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", frame[i]);
   }
   printf("\n");
}

// FCFS Page Replacement Algorithm
void FCFS(int pages[], int page_count) {
   int frame[FRAME_SIZE];
   for (int i = 0; i < FRAME_SIZE; i++) {
      frame[i] = -1;  // Initialize frame with -1 (indicating empty)
   }
   int page_faults = 0;
```

```
  int index = 0;

  for (int i = 0; i < page_count; i++) {
    bool page_found = false;
    for (int j = 0; j < FRAME_SIZE; j++) {
      if (frame[j] == pages[i]) {
        page_found = true;
        break;
      }
    }
    if (!page_found) {
      frame[index] = pages[i];
      index = (index + 1) % FRAME_SIZE;  // Move to next index
cyclically
      page_faults++;
    }
    printFrame(frame, FRAME_SIZE);
  }
  printf("Total Page Faults (FCFS): %d\n", page_faults);
}

// LRU Page Replacement Algorithm
void LRU(int pages[], int page_count) {
  int frame[FRAME_SIZE];
  int last_used[FRAME_SIZE];
  for (int i = 0; i < FRAME_SIZE; i++) {
    frame[i] = -1;  // Initialize frame with -1 (indicating empty)
    last_used[i] = -1;  // Initialize last used time
  }
  int page_faults = 0;
  int time = 0;

  for (int i = 0; i < page_count; i++) {
    bool page_found = false;
```

```
        for (int j = 0; j < FRAME_SIZE; j++) {
            if (frame[j] == pages[i]) {
                page_found = true;
                last_used[j] = time++;
                break;
            }
        }
        if (!page_found) {
            int lru_index = 0;
            for (int j = 1; j < FRAME_SIZE; j++) {
                if (last_used[j] < last_used[lru_index]) {
                    lru_index = j;
                }
            }
            frame[lru_index] = pages[i];
            last_used[lru_index] = time++;
            page_faults++;
        }
        printFrame(frame, FRAME_SIZE);
    }
    printf("Total Page Faults (LRU): %d\n", page_faults);
}

// Optimal Page Replacement Algorithm
void Optimal(int pages[], int page_count) {
    int frame[FRAME_SIZE];
    for (int i = 0; i < FRAME_SIZE; i++) {
        frame[i] = -1;  // Initialize frame with -1 (indicating empty)
    }
    int page_faults = 0;

    for (int i = 0; i < page_count; i++) {
        bool page_found = false;
        for (int j = 0; j < FRAME_SIZE; j++) {
```

```
            if (frame[j] == pages[i]) {
                page_found = true;
                break;
            }
        }
        if (!page_found) {
            int farthest_index = 0;
            int farthest_dist = -1;

            for (int j = 0; j < FRAME_SIZE; j++) {
                int k;
                for (k = i + 1; k < page_count; k++) {
                    if (frame[j] == pages[k]) {
                        if (k > farthest_dist) {
                            farthest_dist = k;
                            farthest_index = j;
                        }
                        break;
                    }
                }
                if (k == page_count) {
                    farthest_index = j;
                    break;
                }
            }
            frame[farthest_index] = pages[i];
            page_faults++;
        }
        printFrame(frame, FRAME_SIZE);
    }
    printf("Total Page Faults (Optimal): %d\n", page_faults);
}

int main() {
```

```c
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 7, 0, 1};
    int page_count = sizeof(pages) / sizeof(pages[0]);

    printf("Page Replacement using FCFS:\n");
    FCFS(pages, page_count);

    printf("\nPage Replacement using LRU:\n");
    LRU(pages, page_count);

    printf("\nPage Replacement using Optimal:\n");
    Optimal(pages, page_count);

    return 0;
}
```

Steps to Compile and Run the C Program

1. **Save the Program**: Save the C program code to a file. For example, name the file `page_replacement.c`.
2. **Open Terminal or Command Prompt**: Open a terminal (Linux or macOS) or Command Prompt (Windows). Navigate to the directory where you saved `page_replacement.c`.

Compile the Program: Use a C compiler like `gcc` (GNU Compiler Collection) to compile the program. Run the following command:

```
gcc page_replacement.c -o page_replacement
```

3. This command will compile the code and produce an

executable named `page_replacement`.

Run the Program: Execute the compiled program using the following command:

```
./page_replacement
```

On Windows, the command would be:

```
page_replacement.exe
```

Understanding the Output

When you run the program, it will display the results of page replacement using FCFS, LRU, and Optimal algorithms. Here's how to interpret the output:

1. **FCFS (First-Come-First-Serve):**
    - **The program will print the state of the frame after each page reference.**
    - **It will also display the total number of page faults that occurred during the execution.**
2. **LRU (Least Recently Used):**
    - **Similarly, the state of the frame will be printed after each page reference.**
    - **The total number of page faults for the LRU algorithm will be shown.**
3. **Optimal:**
    - **The state of the frame will be printed after each page reference.**
    - **The total number of page faults for the Optimal algorithm will be displayed.**

Example Output

Here's an example of what the output might look like for the provided program:

Page Replacement using FCFS:
7 -1 -1
7 0 -1
7 0 1
2 0 1
2 0 3
0 3 7
0 3 0
4 3 0
4 2 0
4 2 3
4 2 0
4 3 0
4 7 0
1 7 0
1 7 0
Total Page Faults (FCFS): 10

Page Replacement using LRU:
7 -1 -1
7 0 -1
7 0 1
2 0 1
2 0 3
2 3 7
0 3 7

0 4 7
0 4 2
0 2 3
0 2 7
0 2 1
Total Page Faults (LRU): 9

Page Replacement using Optimal:
7 -1 -1
7 0 -1
7 0 1
2 0 1
2 0 3
2 3 7
0 3 7
0 4 7
0 4 2
0 2 3
0 2 1
Total Page Faults (Optimal): 8

Explanation of Example Output

- **For FCFS, the output shows the sequence of page frames after each page reference. Each frame is replaced in the order pages are accessed, leading to a total of 10 page faults.**
- **For LRU, the output demonstrates how the least recently used page is replaced. In this example, it results in 9 page faults.**
- **For Optimal, the output shows the page replacement based on future page references. This results in the fewest page faults (8 in this case) since it makes the most optimal choice based on future knowledge.**

Assignment No. 7A

Aim:FIFOS- Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays onstandard output.

Theory:

in computing, a named pipe (also known as a FIFO) is one of the methods for inter-process communication.

It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.

A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.

A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Creating a FIFO file: In order to create a FIFO file, a function calls i.e. mkfifo is used.

mkfifo() makes a FIFO special file with name *pathname*. Here *mode* specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).
Using FIFO: As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. *open*, *read*, *write*, *close*.
Example Programs to illustrate the named pipe: There are two programs that use the same FIFO.

FIFO Special File:
FIFO special file is similar to a pipe, except that it is created in a different way. FIFO special file is entered into the file system by calling mkfifo function.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file.

However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa

How To Create FIFO

Fifo are created using the function mkfifo() which takes as arguments

•€€€€€€€€ The name of the fifo that has to be created

•€€€€€€€€ The permissions for the file.

Once the file is created, it needs to be opened using the system call open() and the data can be read and written from the file using read() and write system calls.

The mkfifo function is declared in the header file sys/stat.h.

Function:

int mkfifo (const char *filename, mode_t mode)

The mkfifo function makes a FIFO special file with name filename. The mode argument is used to set the file's permissions;

The normal, successful return value from mkfifo is 0. In the case of an error, -1 is returned

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it

for reading. Either low-level I/O functions like open, write, read, close or C library I/O functions (fopen, fprintf, fscanf, fclose, and soon) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
intfd = open (fifo_path, O_WRONLY);

        write (fd, data, data_length);

close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
```

```
fscanf (fifo, "%s", buffer);

fclose (fifo);
```

In this problem, we will use two independent processes to communicate using FIFOs (named pipes). The first process will accept sentences and write them to a pipe. The second process will read the sentences, count the number of characters, words, and lines, and write this data to a text file. Then, the second process will write the file's contents back to a second pipe for the first process to read and display the output.

Steps for Implementation

1. Create FIFOs:

- **Use the `mkfifo` command to create two FIFOs (pipes).**
- **One pipe will be used for communication from Process 1 to Process 2, and the other for Process 2 to Process 1.**

Command to create FIFOs:

mkfifo pipe1 pipe2

2.Process 1:

- **Accept sentences from the user.**
- **Write the sentences to `pipe1`.**
- **Read the processed data from `pipe2` and display it.**

3. Process 2:

- **Read the sentences from `pipe1`.**
- **Count the number of characters, words, and lines.**
- **Write the results to a text file.**
- **Write the file's content to `pipe2`.**

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAX 1024

// Function to count characters, words, and lines in a string
void count(char *str, int *charCount, int *wordCount, int
*lineCount) {
   int i = 0;
   *charCount = 0;
   *wordCount = 0;
   *lineCount = 0;

   int inWord = 0;

   while (str[i] != '\0') {
      // Ignore spaces and newlines for character count
      if (str[i] != ' ' && str[i] != '\n') {
         (*charCount)++;
      }

      // Count words: a word starts when we see a non-space
after a space/newline
      if (str[i] != ' ' && str[i] != '\n' && !inWord) {
         (*wordCount)++;
         inWord = 1;
      }
```

```
        // Word ends when we encounter a space or newline
        if (str[i] == ' ' || str[i] == '\n') {
            inWord = 0;
        }

        // Count lines: increment when we see a newline
        if (str[i] == '\n') {
            (*lineCount)++;
        }

        i++;
    }

    // If the last character isn't a newline, we still count one line
    if (i > 0 && str[i-1] != '\n') {
        (*lineCount)++;
    }
}

int main() {
    char input[MAX];
    char full_input[MAX] = ""; // Buffer to store all input lines
    int fd1, fd2;
    int charCount, wordCount, lineCount;
    FILE *fp;

    // Create two FIFOs
    mkfifo("pipe1", 0666);
    mkfifo("pipe2", 0666);

    pid_t pid = fork();

    if (pid > 0) {  // Parent Process (Process 1)
```

```
    // Continuously accept input from the user
    printf("Enter sentences (end input with CTRL+D):\n");
    while (fgets(input, MAX, stdin) != NULL) {
      strcat(full_input, input); // Append each line of input to
full_input buffer
    }

    // Write the combined input to pipe1
    fd1 = open("pipe1", O_WRONLY);
    write(fd1, full_input, strlen(full_input) + 1);
    close(fd1);

    // Read processed data from pipe2
    fd2 = open("pipe2", O_RDONLY);
    char result[MAX];
    read(fd2, result, sizeof(result));
    close(fd2);

    // Display the result
    printf("Processed result from Process 2:\n%s\n", result);

    // Remove the FIFOs
    unlink("pipe1");
    unlink("pipe2");
  } else if (pid == 0) {  // Child Process (Process 2)
    // Read data from pipe1
    fd1 = open("pipe1", O_RDONLY);
    read(fd1, full_input, sizeof(full_input));
    close(fd1);

    // Count characters, words, and lines
    count(full_input, &charCount, &wordCount, &lineCount);

    // Write the counts to a file
```

```
        fp = fopen("output.txt", "w");
        fprintf(fp, "Characters: %d\nWords: %d\nLines: %d\n",
charCount, wordCount, lineCount);
        fclose(fp);

        // Write the contents of the file to pipe2
        fd2 = open("pipe2", O_WRONLY);
        fp = fopen("output.txt", "r");
        char fileContents[MAX];
        fread(fileContents, sizeof(char), MAX, fp);
        write(fd2, fileContents, strlen(fileContents) + 1);
        close(fd2);
        fclose(fp);
    } else {
        perror("Fork failed");
        exit(1);
    }

    return 0;
}
```

Detailed Explanation

1. **FIFOs Creation:** We use `mkfifo()` to create two named pipes: `pipe1` and `pipe2`. This is done to establish a communication channel between the two processes.

```
mkfifo("pipe1", 0666);
mkfifo("pipe2", 0666);
```

1. **Process 1 (Parent):**
   - **This process accepts sentences from the user using `fgets()`.**
   - **The input is written to `pipe1` using the `write()` system call.**
   - **After the second process processes the data, it reads the result from `pipe2` and displays it to the standard output.**
2. **Process 2 (Child):**
   - **This process reads the input from `pipe1`.**
   - **It counts the number of characters, words, and lines in the input string using the `count()` function.**
   - **The results are written to a file called `output.txt`.**
   - **The contents of the file are then written to `pipe2` for Process 1 to read.**
3. **File Operations:**
   - **Process 2 writes the results to a file (`output.txt`) using standard `fopen()`, `fprintf()`, and `fclose()` functions.**
   - **The file is then opened again to read its contents and send them through the pipe to Process 1.**
4. **Forking:**
   - **The program uses `fork()` to create a child process. The parent (Process 1) handles user input and output, while the child (Process 2) handles counting and file operations.**

Commands to Compile and Run the Program

gcc -o fifo_communication fifo_communication.c

Run the program:

./fifo_communication

Process 1 (User Input): The program will prompt the user to enter a sentence:

Enter a sentence (end with CTRL+D):

Let's input the following sentence:

Hello world
How are you?

Press `CTRL+D` to signal the end of the input.
Process 2 (Count and File Output): The second process will read the input from `pipe1` and perform the character, word, and line count. Then, it will write the result to a text file (`output.txt`) and send it back to `pipe2`.
Process 1 (Displaying Result): After reading the processed data from `pipe2`, Process 1 will display the following output on the screen:

Processed result from Process 2:
Characters: 25
Words: 6
Lines: 2

1. **This indicates that:**
    - **Characters: 25 characters (including spaces and newline).**
    - **Words: 6 words.**

    - **Lines: 2 lines.**

Explanation of Output

- **The sentence `"Hello world\nHow are you?\n"` has:**
    - **25 characters: This includes the letters, spaces, and newline characters.**
    - **6 words: The words are "Hello", "world", "How", "are", "you".**
    - **2 lines: The sentence is divided into two lines by the newline character (`\n`).**

Content of `output.txt`

In addition to the console output, Process 2 writes the results to a file (`output.txt`). The file will contain:

Characters: 25
Words: 6
Lines: 2

Remove the FIFOs: After running the program, make sure to remove the created pipes (if not done programmatically):

rm pipe1 pipe2

Also, you can remove the `output.txt` file if needed:

rm output.txt

Key Concepts

- **mkfifo(): Creates a named pipe (FIFO).**
- **fork(): Creates a child process.**
- **open(), read(), write(), close(): System calls for reading and writing data through pipes.**
- **File handling: Writing results to a file and reading from it.**

Assignment No. 7B

Aim: Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

Theory:

Detailed Explanation: Inter-Process Communication (IPC) using Shared Memory in System V

Inter-Process Communication (IPC) is a mechanism that allows processes (running programs) to exchange data and synchronize their actions. One of the most efficient forms of IPC is Shared Memory, which allows multiple processes to access the same memory segment.

What is Shared Memory?

Shared memory is a segment of memory that is accessible by multiple processes simultaneously. It is the fastest form of IPC because data is directly accessible without the overhead of kernel intervention, as in other IPC methods like pipes or message queues.

When a shared memory segment is created, it allows processes to:

1. **Write data to the memory.**
2. **Read data from the memory.**

The shared memory is useful when processes need to exchange a large amount of data. In System V IPC, shared memory segments are identified by a unique key, and once created, any process with access to that key can attach to the shared memory segment and read/write data.

IPC Using Shared Memory - System V

In System V IPC, shared memory is managed using these three system calls:

1. **shmget() - Creates or accesses a shared memory segment.**
2. **shmat() - Attaches a shared memory segment to the process's address space.**
3. **shmdt() - Detaches a shared memory segment from the process's address space.**

Additional system calls include:

- **shmctl() - Controls and performs operations such as deleting the shared memory segment.**

The Scenario: Client-Server Communication using Shared Memory

We will demonstrate the Client-Server model where:

- **The server process creates a shared memory segment and writes a message to it.**
- **The client process reads that message from the shared memory.**

Here's how this works step-by-step:

Step 1: Server Creates Shared Memory and Writes a Message

The server process does the following:

1. **Create the Shared Memory Segment using `shmget()`:**
   - **This allocates a portion of memory that both processes can share.**
   - **The key provided (`SHM_KEY`) is used to uniquely identify the shared memory.**

2. **Attach the Shared Memory using `shmat()`:**
    - **Once the shared memory is created, the process needs to attach it to its own address space to read/write data into it.**
3. **Write a Message:**
    - **The server writes a message to the shared memory, which is stored as a character string.**
4. **Detach from the Shared Memory using `shmdt()`:**
    - **After writing the message, the server detaches the shared memory. However, the shared memory remains available until explicitly removed.**

Step 2: Client Attaches to the Shared Memory and Reads the Message

The client process performs the following actions:

1. **Locate the Shared Memory using `shmget()`:**
    - **The client uses the same key (`SHM_KEY`) to locate the already created shared memory segment.**
    - **The client doesn't create a new segment, it simply accesses the one created by the server.**
2. **Attach to the Shared Memory using `shmat()`:**
    - **The client process attaches to the shared memory segment so that it can read the data.**
3. **Read the Message:**
    - **The client reads the message that was written by the server.**
4. **Detach from the Shared Memory using `shmdt()`:**
    - **After reading the data, the client detaches the shared memory segment from its address space.**

System V Shared Memory Functions in Detail

1. `shmget()`:
    - Purpose: Creates or locates a shared memory segment.
    - Prototype: `int shmget(key_t key, size_t size, int shmflg);`
    - Parameters:
        - `key`: Unique identifier for the shared memory.
        - `size`: Size of the shared memory segment (in bytes).
        - `shmflg`: Flags, such as `IPC_CREAT` (create if not exists), `IPC_EXCL` (fail if exists), and permission bits (e.g., `0666` for read/write permission).
    - Return: Returns a shared memory ID (shmid), or -1 on failure.
2. `shmat()`:
    - Purpose: Attaches the shared memory segment to the process's address space.
    - Prototype: `void *shmat(int shmid, const void *shmaddr, int shmflg);`
    - Parameters:
        - `shmid`: Shared memory ID returned by `shmget()`.
        - `shmaddr`: Usually NULL, allowing the system to choose the address.
        - `shmflg`: Flags (0 for default behavior).
    - Return: Pointer to the shared memory, or `(void *) -1` on failure.
3. `shmdt()`:
    - Purpose: Detaches the shared memory segment from the process.
    - Prototype: `int shmdt(const void *shmaddr);`
    - Parameters:

        - `shmaddr`: Pointer to the shared memory segment returned by `shmat()`.
4. `shmctl()`:
    - Purpose: Performs control operations on the shared memory, such as deleting the segment.
    - Prototype: `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
    - Common Commands:
        - `IPC_RMID`: Removes the shared memory segment.

Visual Representation

```
+------------------------+              +------------------------+
  |      Server         |                 |      Client         |
|------------------------|               |------------------------|
| Create Shared Memory    |          | Attach to Shared Memory |
| Write to Shared Memory   |         | Read from Shared Memory |
| Detach from Shared Memory|         | Detach from Shared Memory|
+------------------------+                +------------------------+

        |                              |
        |                              |
        +------------------------------+
           Shared Memory Segment
           (e.g., "Hello from Server!")
```

Advantages of Shared Memory

1. Efficiency: Shared memory is the fastest form of IPC because processes directly access the same memory.
2. No Kernel Intervention: Once the shared memory is set up, the operating system does not need to be involved in

subsequent reads/writes.

3. **Large Data Exchange: It is especially useful for sharing large amounts of data because it avoids the overhead of copying data multiple times between processes.**

Potential Issues

- **Synchronization: Since multiple processes can read/write to the same shared memory segment, you must handle synchronization (e.g., using semaphores or mutexes) to avoid race conditions.**
- **Persistence: The shared memory remains available until explicitly deleted, even if the processes using it terminate.**

Step 1: Setup Shared Memory (Server Program)

The server process will:

1. **Create a shared memory segment.**
2. **Attach itself to the shared memory.**
3. **Write data to the shared memory.**
4. **Detach from the shared memory.**

Step 2: Client Reads Shared Memory

The client process will:

1. **Attach to the shared memory segment.**
2. **Read the message from the shared memory.**
3. **Detach from the shared memory.**

Code Implementation:

1. Server Program

```c
// Server Program: server.c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <stdlib.h>

#define SHM_KEY 12345  // Key for shared memory
#define SHM_SIZE 1024  // Size of shared memory segment

int main() {
   int shmid;
   char *shm_ptr;

   // Create a shared memory segment
   shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
   if (shmid == -1) {
      perror("shmget failed");
      exit(1);
   }

   // Attach the shared memory segment to the server's address space
   shm_ptr = (char *) shmat(shmid, NULL, 0);
   if (shm_ptr == (char *) -1) {
      perror("shmat failed");
      exit(1);
   }

   // Write a message to shared memory
   printf("Writing to shared memory: Hello from Server!\n");
   strcpy(shm_ptr, "Hello from Server!");
```

```c
    // Detach from the shared memory
    shmdt(shm_ptr);

    return 0;
}
```

2. Client Program

```c
// Client Program: client.c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>

#define SHM_KEY 12345  // Same key as in the server
#define SHM_SIZE 1024  // Size of shared memory segment

int main() {
    int shmid;
    char *shm_ptr;

    // Locate the shared memory segment
    shmid = shmget(SHM_KEY, SHM_SIZE, 0666);
    if (shmid == -1) {
        perror("shmget failed");
        exit(1);
    }

    // Attach to the shared memory segment
    shm_ptr = (char *) shmat(shmid, NULL, 0);
    if (shm_ptr == (char *) -1) {
```

```
        perror("shmat failed");
        exit(1);
    }

    // Read the message from shared memory
    printf("Message from server: %s\n", shm_ptr);

    // Detach from the shared memory
    shmdt(shm_ptr);

    return 0;
}
```

Compilation and Execution:

1. Compile the server and client programs separately:

```
gcc server.c -o server
gcc client.c -o client
```

2. Run the server first to create the shared memory and write the message:

```
./server
```

3. Then, run the client to read the message from shared memory:

```
./client
```

Explanation:

- **`shmget`: Creates or accesses a shared memory segment.**
- **`shmat`: Attaches the shared memory segment to the process's address space.**
- **`shmdt`: Detaches the shared memory segment from the process.**
- **The server writes the message "Hello from Server!" to shared memory.**
- **The client reads and displays the message from shared memory.**

Clean-up:

To remove the shared memory segment after both processes have finished:

ipcrm -m <shmid>

You can find the `shmid` by running `ipcs` to list active shared memory segments.

Assignment No 8

Aim: Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look

considering the initial head position moving away from the spindle.

Theory:

Disk Scheduling Algorithms in C: SSTF, SCAN, and C-LOOK

Disk scheduling algorithms manage the order in which I/O requests for reading/writing sectors on the disk are handled. They aim to minimize the total seek time (the time the read/write head takes to move from one disk sector to another).

Here, we will implement three commonly used disk scheduling algorithms:

1. **SSTF (Shortest Seek Time First)**
2. **SCAN**
3. **C-LOOK**

Problem Consideration:

- **We have a disk with a specified number of cylinders.**
- **We know the initial head position (where the read/write head is currently located).**
- **We have a sequence of requests for various cylinder positions.**

Inputs:

- **Initial head position: The starting cylinder.**
- **Requests: The list of cylinder requests to process.**

1. SSTF (Shortest Seek Time First) Algorithm

The SSTF algorithm selects the request with the shortest seek time (i.e., the closest to the current head position).

SSTF Algorithm Steps:

1. **Find the request closest to the current head position.**

2. **Move the head to this position and serve the request.**
3. **Repeat until all requests have been served.**

2. SCAN (Elevator Algorithm)

The SCAN algorithm moves the head towards one end of the disk, servicing all requests in its path, and then reverses direction at the end, serving requests in the opposite direction.

SCAN Algorithm Steps:

1. **Start from the initial head position and move in a fixed direction (e.g., towards the highest cylinder).**
2. **Service all requests in the current direction.**
3. **Once the end is reached, reverse direction and service the remaining requests.**

3. C-LOOK Algorithm

The C-LOOK algorithm is similar to SCAN, but instead of reversing direction, the head jumps back to the lowest request and continues moving in the same direction.

C-LOOK Algorithm Steps:

1. **Move the head towards the highest request and serve requests in that direction.**
2. **When the head reaches the highest request, jump back to the lowest request and continue in the same direction.**

Code:

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

#define MAX_REQUESTS 100
```

```
// Function prototypes
void sstf(int requests[], int n, int head);
void scan(int requests[], int n, int head, int disk_size);
void c_look(int requests[], int n, int head);

int main() {
   int n, head, choice, disk_size;
   int requests[MAX_REQUESTS];

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

   printf("Enter the request queue: ");
   for (int i = 0; i < n; i++) {
      scanf("%d", &requests[i]);
   }

   printf("Enter the initial head position: ");
   scanf("%d", &head);

   printf("Enter the total number of cylinders in the disk: ");
   scanf("%d", &disk_size);

   do {
      printf("\nDisk Scheduling Algorithms\n");
      printf("1. SSTF\n2. SCAN\n3. C-LOOK\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            sstf(requests, n, head);
            break;
```

```
        case 2:
          scan(requests, n, head, disk_size);
          break;
        case 3:
          c_look(requests, n, head);
          break;
        case 4:
          printf("Exiting...\n");
          break;
        default:
          printf("Invalid choice!\n");
      }
  } while (choice != 4);

  return 0;
}

// Helper function to sort the array
void sort(int arr[], int n) {
  int temp;
  for (int i = 0; i < n-1; i++) {
    for (int j = i+1; j < n; j++) {
      if (arr[i] > arr[j]) {
        temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    }
  }
}

// SSTF Algorithm
void sstf(int requests[], int n, int head) {
  int total_seek_time = 0, completed = 0, min_index, min_seek;
```

```
    int completed_requests[MAX_REQUESTS] = {0}; // Keep track
of completed requests
    int current_head = head;

    printf("\nSSTF Disk Scheduling\n");
    printf("Order of service: ");

    while (completed < n) {
        min_seek = 9999; // Large value for comparison
        for (int i = 0; i < n; i++) {
            if (!completed_requests[i]) {
                int seek_time = abs(requests[i] - current_head);
                if (seek_time < min_seek) {
                    min_seek = seek_time;
                    min_index = i;
                }
            }
        }
        total_seek_time += min_seek;
        current_head = requests[min_index];
        completed_requests[min_index] = 1;
        completed++;
        printf("%d ", current_head);
    }

    printf("\nTotal Seek Time: %d\n", total_seek_time);
    printf("Average Seek Time: %.2f\n", (float)total_seek_time /
n);
}

// SCAN Algorithm
void scan(int requests[], int n, int head, int disk_size) {
    int total_seek_time = 0, current_head = head;
    int requests_with_limits[MAX_REQUESTS + 2]; // Including 0
```

```
and disk_size
    int i, j;

    printf("\nSCAN Disk Scheduling\n");
    printf("Order of service: ");

    // Include 0 and disk_size as limits
    requests_with_limits[0] = 0;
    for (i = 0; i < n; i++) {
        requests_with_limits[i + 1] = requests[i];
    }
    requests_with_limits[n + 1] = disk_size - 1;

    // Sort the requests along with 0 and disk_size
    sort(requests_with_limits, n + 2);

    // Move towards the right from head
    for (i = 0; i < n + 2; i++) {
        if (requests_with_limits[i] >= head) {
            break;
        }
    }

    // Serve the right side
    for (j = i; j < n + 2; j++) {
        printf("%d ", requests_with_limits[j]);
        total_seek_time += abs(current_head -
requests_with_limits[j]);
        current_head = requests_with_limits[j];
    }

    // Serve the left side after reaching the end
    for (j = i - 1; j >= 0; j--) {
        printf("%d ", requests_with_limits[j]);
```

```
        total_seek_time += abs(current_head -
requests_with_limits[j]);
        current_head = requests_with_limits[j];
    }

    printf("\nTotal Seek Time: %d\n", total_seek_time);
    printf("Average Seek Time: %.2f\n", (float)total_seek_time /
n);
}

// C-LOOK Algorithm
void c_look(int requests[], int n, int head) {
    int total_seek_time = 0, current_head = head;
    int i, j;

    printf("\nC-LOOK Disk Scheduling\n");
    printf("Order of service: ");

    // Sort the requests
    sort(requests, n);

    // Move towards the right from head
    for (i = 0; i < n; i++) {
        if (requests[i] >= head) {
            break;
        }
    }

    // Serve the right side
    for (j = i; j < n; j++) {
        printf("%d ", requests[j]);
        total_seek_time += abs(current_head - requests[j]);
        current_head = requests[j];
    }
```

```
  // Jump to the beginning (lowest request) and serve the left
side
  for (j = 0; j < i; j++) {
    printf("%d ", requests[j]);
    total_seek_time += abs(current_head - requests[j]);
    current_head = requests[j];
  }

  printf("\nTotal Seek Time: %d\n", total_seek_time);
  printf("Average Seek Time: %.2f\n", (float)total_seek_time /
n);
}
```

Explanation of the Code:

1. **Common Structure:**
    - **The program accepts the number of requests, the list of requests (cylinders), and the initial head position as input.**
    - **It also asks for the total disk size to handle the SCAN and C-LOOK algorithms properly.**
    - **Each algorithm calculates the total seek time and average seek time for the provided requests.**
2. **Helper Function:**
    - `sort()`**: This function sorts the request array (used in SCAN and C-LOOK to process requests in order).**
3. **SSTF Algorithm:**
    - **The SSTF algorithm repeatedly finds the request closest to the current head position and serves it.**
    - **A** `completed_requests[]` **array is used to track which requests have already been served.**

    - **The seek time is calculated as the absolute difference between the current head position and the request being served.**
4. **SCAN Algorithm:**
    - **The head moves towards the end of the disk and serves requests in one direction, then reverses and serves the remaining requests.**
    - **Requests are sorted along with 0 (the start) and `disk_size - 1` (the end).**
5. **C-LOOK Algorithm:**
    - **The head moves towards the highest request, and when it reaches the end, it jumps to the lowest request and continues in the same direction.**
    - **No reversing of the head is done.**

Sample Input:

Enter the number of requests: 8
Enter the request queue: 95 180 34 119 11 123 62 64
Enter the initial head position: 50
Enter the total number of cylinders in the disk: 200

Output:
Disk Scheduling Algorithms
1. SSTF
2. SCAN
3. C-LOOK
Enter your choice: 1

SSTF Disk Scheduling
Order of service: 62 64 34 11 95 119 123 180
Total Seek Time: 236
Average Seek Time: 29.50