----------------------- 1 Shell Scripting --------------------------------

#!/bin/bash

# Function for if-else statement

if\_else\_demo() {

read -p "Enter your age: " age

if (( age < 18 )); then

echo "You are a minor."

elif (( age <= 65 )); then

echo "You are an adult."

else

echo "You are a senior."

fi

}

# Function for for loop to print numbers from start to end

for\_loop\_demo() {

read -p "Enter the start number: " start

read -p "Enter the end number: " end

for (( i=start; i<=end; i++ )); do

echo "Number $i"

done

}

# Function to check if a number is prime

is\_prime() {

local num=$1

if (( num <= 1 )); then

return 1

fi

for (( i=2; i\*i<=num; i++ )); do

if (( num % i == 0 )); then

return 1

fi

done

return 0

}

# Function for while loop demonstration (prime numbers)

while\_loop\_demo() {

read -p "Enter a start number: " start

read -p "Enter an end number: " end

while (( start <= end )); do

if is\_prime $start; then

echo "Prime number: $start"

fi

((start++))

done

}

# Function to calculate factorial of a number

factorial() {

local num=$1

local factorial=1

local i=1

until (( i > num )); do

factorial=$(( factorial \* i ))

((i++))

done

echo $factorial

}

# Function for until loop demonstration (factorial)

until\_loop\_demo() {

read -p "Enter a number to calculate factorial: " num

result=$(factorial $num)

echo "Factorial of $num is $result"

}

# Display the menu

while true; do

echo "Menu:"

echo "1) If-Else Statement"

echo "2) For Loop"

echo "3) While Loop"

echo "4) Until Loop"

echo "5) Exit"

read -p "Choose an option: " choice

case $choice in

1)

if\_else\_demo

;;

2)

for\_loop\_demo

;;

3)

while\_loop\_demo

;;

4)

until\_loop\_demo

;;

5)

echo "Exiting..."

break

;;

\*)

echo "Invalid option, please try again."

;;

esac

done

----------------------------------------------------------

By Siddhant

-----------------------------------------------------------

PROGARM:

1) Print Sum of Digits of a given number using command line argument

CODE:

x=$1

sum=0

n=$x

while [ $n -gt 0 ];

do

temp=$(expr $n % 10)

sum=$(expr $sum + $temp)

n=$(expr $n / 10)

done

echo "The sum of digits is $sum"

OUTPUT:

sid@SID:~/shell\_scripts$ ./add\_digit.sh 27

The sum of digits is 9

sid@SID:~/shell\_scripts$ ./add\_digit.sh 998

The sum of digits is 26

2) Write a shell script using function for following:

1)average of given numbers

CODE:

echo "Enter the number of values: "

read x

sum=0

i=1

SCALE=2

while [ $i -le $x ] ;

do

echo "Enter the number $i: "

read n

sum=$(expr $sum + $n)

let i++;

done

echo "The average of numbers is: $(echo "scale=$SCALE; $sum / $x" | bc -l)"

OUTPUT:

sid@SID:~/shell\_scripts$ ./average.sh

Enter the number of values:

4

Enter the number 1:

40

Enter the number 2:

64

Enter the number 3:

52

Enter the number 4:

20

The average of numbers is: 44

2) Max digit from given number

CODE:

echo "Enter the amount of numbers: "

read x

sum=0

i=1

max=0

while [ $i -le $x ] ;

do

echo "Enter the number $i: "

read n

if [ $n -gt $max ]

then

max=$n

fi

let i++;

done

echo "The maximum of the given numbers is: $max"

OUTPUT:

sid@SID:~/shell\_scripts$ ./max\_digit.sh

Enter the amount of numbers:

5

Enter the number 1:

0

Enter the number 2:

46

Enter the number 3:

22

Enter the number 4:

7

Enter the number 5:

86

The maximum of the given numbers is: 86

3) min digit from given number

CODE:

echo "Enter the amount of numbers: "

read x

i=2

echo "Enter the number 1"

read min

while [ $i -le $x ] ;

do

echo "Enter the number $i: "

read n

if [ $n -lt $min ]

then

min=$n

fi

let i++;

done

echo "The minimum of the given numbers is: $min"

OUTPUT:

sid@SID:~/shell\_scripts$ ./min\_digit.sh

Enter the amount of numbers:

4

Enter the number 1

0

Enter the number 2:

46

Enter the number 3:

22

Enter the number 4:

7

The minimum of the given numbers is: 0

3) Perform sorting on given array elements

CODE:

bubble\_sort() {

local arr=("$@")

local n=${#arr[@]}

local temp

for ((i = 0; i < n; i++)); do

for ((j = 0; j < n - i - 1; j++)); do

if [[ ${arr[j]} -gt ${arr[j + 1]} ]]; then

temp=${arr[j]}

arr[j]=${arr[j + 1]}

arr[j + 1]=$temp

fi

done

done

echo "${arr[@]}"

}

echo "Enter the elements of the array:"

read -a arr

echo "Original array: ${arr[@]}"

sorted\_arr=($(bubble\_sort "${arr[@]}"))

echo "Sorted array: ${sorted\_arr[@]}"

OUTPUT:

sid@SID:~/shell\_scripts$ ./sort\_array.sh

Enter the elements of the array:

9 1 4 3 7 0

Original array: 9 1 4 3 7 0

Sorted array: 0 1 3 4 7 9

4) Program to find factorial of a given number with and without recursion

CODE (with recursion):

factorial\_recursive() {

if [ $1 -le 1 ]; then

echo 1

else

local temp=$(( $1 - 1 ))

local result=$(factorial\_recursive $temp)

echo $(( $1 \* result ))

fi

}

echo "Enter a number to find its factorial (using recursion):"

read number

result=$(factorial\_recursive $number)

echo "Factorial of $number is $result"

OUTPUT:

sid@SID:~/shell\_scripts$ ./with\_recur\_fact.sh

Enter a number to find its factorial (using recursion):

4

Factorial of 4 is 24

sid@SID:~/shell\_scripts$ ./with\_recur\_fact.sh

Enter a number to find its factorial (using recursion):

3

Factorial of 3 is 6

CODE (without recursion):

factorial\_non\_recursive() {

local n=$1

local result=1

for (( i = 1; i <= n; i++ )); do

result=$(( result \* i ))

done

echo $result

}

echo "Enter a number to find its factorial (without recursion):"

read number

result=$(factorial\_non\_recursive $number)

echo "Factorial of $number is $result"

OUTPUT:

sid@SID:~/shell\_scripts$ ./without\_recur\_fact.sh

Enter a number to find its factorial (without recursion):

4

Factorial of 4 is 24

5)Program to check file type and permission for a given file

CODE:

echo "Enter the file path:"

read file

if [ -e "$file" ]; then

file\_type=$(file -b "$file")

permissions=$(stat -c "%A" "$file")

echo "File type: $file\_type"

echo "Permissions: $permissions"

else

echo "File does not exist."

fi

OUTPUT:

sid@SID:~/shell\_scripts$ ./file\_type.sh

Enter the file path:

/home/sid/shell\_scripts/sort\_array.sh

File type: ASCII text

Permissions: -rwxr-xr-x

6) Check entered string is palindrome or not?

CODE:

echo "Enter a string:"

read str

len=${#str}

for (( i=0; i<$len/2; i++ )); do

if [ "${str:$i:1}" != "${str: -$(($i + 1)):1}" ]; then

echo "The string is not a palindrome."

exit 0

fi

done

echo "The string is a palindrome."

OUTPUT:

sid@SID:~/shell\_scripts$ ./palindrome.sh

Enter a string:

KAYAK

The string is a palindrome.

sid@SID:~/shell\_scripts$ ./palindrome.sh

Enter a string:

SIDDHANT

The string is not a palindrome.

----------------------2 System Calls -----------------------------------------------------

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <string.h>

#define FIFO\_FILE "my\_fifo"

void process\_related() {

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

printf("Child process (PID: %d) is running\n", getpid());

exit(EXIT\_SUCCESS);

} else {

wait(NULL);

printf("Parent process (PID: %d) has waited for child pro-cess\n", getpid());

}

}

void file\_related() {

int fd = open("example.txt", O\_CREAT | O\_RDWR, 0644);

if (fd < 0) {

perror("open");

return;

}

write(fd, "Hello, world!\n", 14);

lseek(fd, 0, SEEK\_SET);

char buffer[15];

read(fd, buffer, 14);

buffer[14] = '\0';

printf("Read from file: %s", buffer);

close(fd);

// Create a hard link and then delete the original file

link("example.txt", "example\_link.txt");

unlink("example.txt");

struct stat st;

if (stat("example\_link.txt", &st) == 0) {

printf("Link file size: %ld bytes\n", st.st\_size);

}

unlink("example\_link.txt");

}

void communication() {

int fd[2];

if (pipe(fd) == -1) {

perror("pipe");

return;

}

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

return;

}

if (pid == 0) {

// Child process

close(fd[1]);

char buffer[100];

read(fd[0], buffer, sizeof(buffer));

printf("Child received: %s\n", buffer);

close(fd[0]);

exit(EXIT\_SUCCESS);

} else {

// Parent process

close(fd[0]);

write(fd[1], "Hello from parent!", 19);

close(fd[1]);

wait(NULL);

}

}

void info\_related() {

printf("Process ID (PID): %d\n", getpid());

printf("Parent Process ID (PPID): %d\n", getppid());

printf("User ID (UID): %d\n", getuid());

printf("Group ID (GID): %d\n", getgid());

}

int main() {

int choice;

while (1) {

printf("\nMenu:\n");

printf("1. Process Related\n");

printf("2. File Related\n");

printf("3. Communication\n");

printf("4. Information Related\n");

printf("5. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

process\_related();

break;

case 2:

file\_related();

break;

case 3:

communication();

break;

case 4:

info\_related();

break;

case 5:

exit(EXIT\_SUCCESS);

default:

printf("Invalid choice, try again.\n");

}

}

return 0;

}

/\*

Menu System (Main Function)

The program presents the user with a menu that has the following options:

Process Related Operations

File Related Operations

Inter-Process Communication (IPC) using pipes

Display Process Information

Exit

The main() function runs in an infinite loop, prompting the user for input and executing the corresponding functionality based on the user's choice.

1. Process-Related Operations (process\_related)

The function process\_related() demonstrates process creation using fork(). The fork() system call creates a new child process. Both the parent and the child process will execute the code that follows the fork().

Code flow:

Forking a process: The child process is created using fork(). The return value of fork() is 0 for the child and the PID of the child for the parent.

Child process: The child prints its PID and exits immediately using exit().

Parent process: The parent waits for the child process to complete using wait(), then prints that it has waited for the child.

getpid(): Returns the PID of the current process.

wait(NULL): Waits for any child process to finish.

2. File-Related Operations (file\_related)

This function demonstrates file handling in Linux:

Opening a file using open().

Writing to a file using write().

Reading from a file using read().

Creating a hard link using link().

Deleting a file using unlink().

Code flow:

Open a file called example.txt with O\_CREAT and O\_RDWR flags.

Write a string to the file.

Read back from the file and print the content.

Create a hard link to the file (example\_link.txt) and then delete the original file.

Use stat() to check the size of the linked file.

open(): Opens a file for reading and writing, creating it if necessary.

lseek(): Moves the file pointer to the beginning of the file.

write() and read(): Write to and read from the file respectively.

link(): Creates a hard link.

unlink(): Deletes a file.

stat(): Retrieves file information (e.g., file size).

3. Inter-Process Communication (IPC) Using Pipes (communication)

This function demonstrates communication between processes using a pipe. A pipe allows data to be passed between the parent and child processes.

Code flow:

The pipe() system call creates a unidirectional communication channel between the parent and the child process.

After forking:

The child process closes the write-end of the pipe, reads from the read-end, and prints the received message.

The parent process closes the read-end of the pipe, writes a message to the pipe, and then waits for the child process to finish.

pipe(fd): Creates a pipe, fd[0] is the read-end, and fd[1] is the write-end.

read() and write(): Transfer data between the processes.

close(): Closes the pipe ends to ensure the processes only read from the read-end and write to the write-end.

4. Process Information (info\_related)

This function simply prints out some useful information about the current process:

Process ID (PID): The unique identifier for the current process.

Parent Process ID (PPID): The PID of the parent process.

User ID (UID): The UID of the user who started the process.

Group ID (GID): The GID of the user who started the process.

getpid(): Returns the PID of the current process.

getppid(): Returns the PID of the parent process.

getuid() and getgid(): Return the UID and GID of the current process.

Exit and Loop Control

The program will continue to display the menu and execute the chosen operations until the user selects option 5 (Exit), which terminates the program using exit(EXIT\_SUCCESS).

Conclusion

This program covers several important system programming concepts in Unix/Linux:

Process creation and management using fork() and wait().

File operations like creating, reading, writing, and linking files.

Inter-process communication via pipes (pipe()).

Retrieving system information using functions like getpid(), getppid(), and getuid().

\*/

-----------------Multi Threading for matrix operations ----------------

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

int n, \*\*matrix1, \*\*matrix2, \*\*resultantMatrix;

void \*takeInput(void \*args) {

int \*\*ptr = (int \*\*)args;

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

printf("Enter element at index %d, %d: ", i, j);

scanf("%d", &ptr[i][j]);

}

}

return NULL;

}

void \*printMatrix(void \*args) {

int \*\*ptr = (int \*\*)args;

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

printf("%d ", ptr[i][j]);

}

printf("\n");

}

return NULL;

}

void \*addMatrix(void \*args) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

resultantMatrix[i][j] = matrix1[i][j] + matrix2[i][j];

}

}

return NULL;

}

void \*subMatrix(void \*args) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

resultantMatrix[i][j] = matrix1[i][j] - matrix2[i][j];

}

}

return NULL;

}

void \*multiplyMatrix(void \*args) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

int sum = 0; // Reset sum for each element

for (int k = 0; k < n; k++) {

sum += (matrix1[i][k] \* matrix2[k][j]);

}

resultantMatrix[i][j] = sum;

}

}

return NULL;

}

int main() {

printf("Enter rows/columns of matrix: ");

scanf("%d", &n);

// Allocate memory for matrices

matrix1 = (int \*\*)malloc(sizeof(int \*) \* n);

matrix2 = (int \*\*)malloc(sizeof(int \*) \* n);

resultantMatrix = (int \*\*)malloc(sizeof(int \*) \* n);

for (int i = 0; i < n; i++) {

matrix1[i] = (int \*)malloc(sizeof(int) \* n);

matrix2[i] = (int \*)malloc(sizeof(int) \* n);

resultantMatrix[i] = (int \*)malloc(sizeof(int) \* n);

}

pthread\_t t1, t2;

printf("\n\nFor matrix 1:\n");

pthread\_create(&t1, NULL, takeInput, (void \*)matrix1);

pthread\_join(t1, NULL);

printf("\n\nFor matrix 2:\n");

pthread\_create(&t2, NULL, takeInput, (void \*)matrix2);

pthread\_join(t2, NULL);

printf("\n\nMatrix 1:\n");

pthread\_create(&t1, NULL, printMatrix, (void \*)matrix1);

pthread\_join(t1, NULL);

printf("\n\nMatrix 2:\n");

pthread\_create(&t2, NULL, printMatrix, (void \*)matrix2);

pthread\_join(t2, NULL);

pthread\_t threadForAddition, threadForSubtraction, threadForMatrixMultiplication;

pthread\_create(&threadForAddition, NULL, addMatrix, NULL);

pthread\_join(threadForAddition, NULL);

printf("\nAddition of 2 matrices:\n");

pthread\_create(&threadForAddition, NULL, printMatrix, (void \*)resultantMatrix);

pthread\_join(threadForAddition, NULL);

pthread\_create(&threadForSubtraction, NULL, subMatrix, NULL);

pthread\_join(threadForSubtraction, NULL);

printf("\nSubtraction of 2 matrices:\n");

pthread\_create(&threadForSubtraction, NULL, printMatrix, (void \*)resultantMatrix);

pthread\_join(threadForSubtraction, NULL);

pthread\_create(&threadForMatrixMultiplication, NULL, multiplyMatrix, NULL);

pthread\_join(threadForMatrixMultiplication, NULL);

printf("\nMatrix Multiplication of 2 matrices:\n");

pthread\_create(&threadForMatrixMultiplication, NULL, printMatrix, (void \*)resultantMatrix);

pthread\_join(threadForMatrixMultiplication, NULL);

// Free allocated memory

for (int i = 0; i < n; i++) {

free(matrix1[i]);

free(matrix2[i]);

free(resultantMatrix[i]);

}

free(matrix1);

free(matrix2);

free(resultantMatrix);

return 0;

}

------------------------------- 4 Reader Writer --------------------------------

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

pthread\_mutex\_t mutex;

pthread\_mutex\_t writeLock;

int readers = 0;

int data = 0;

void\* reader(void\* arg);

void\* writer(void\* arg);

void enter\_reader(int reader\_id);

void exit\_reader(int reader\_id);

void enter\_writer(int writer\_id);

void exit\_writer(int writer\_id);

int main() {

pthread\_t r\_threads[5], w\_threads[3];

pthread\_mutex\_init(&mutex, NULL);

pthread\_mutex\_init(&writeLock, NULL);

for (int i = 0; i < 5; i++) {

pthread\_create(&r\_threads[i], NULL, reader, (void\*)(size\_t)(i+1));

}

for (int i = 0; i < 3; i++) {

pthread\_create(&w\_threads[i], NULL, writer, (void\*)(size\_t)(i+1));

}

for (int i = 0; i < 5; i++) {

pthread\_join(r\_threads[i], NULL);

}

for (int i = 0; i < 3; i++) {

pthread\_join(w\_threads[i], NULL);

}

pthread\_mutex\_destroy(&mutex);

pthread\_mutex\_destroy(&writeLock);

return 0;

}

void\* reader(void\* arg) {

int reader\_id = (int)(size\_t)arg;

sleep(rand() % 5);

enter\_reader(reader\_id);

printf("Reader %d: Reading data: %d\n", reader\_id, data);

exit\_reader(reader\_id);

return NULL;

}

void\* writer(void\* arg) {

int writer\_id = (int)(size\_t)arg;

sleep(rand() % 5);

enter\_writer(writer\_id);

data = rand() % 100;

printf("Writer %d: Writing data: %d\n", writer\_id, data);

exit\_writer(writer\_id);

return NULL;

}

void enter\_reader(int reader\_id) {

pthread\_mutex\_lock(&mutex);

readers++;

if (readers == 1) {

pthread\_mutex\_lock(&writeLock);

}

printf("Reader %d is entering the critical section.\n", reader\_id);

pthread\_mutex\_unlock(&mutex);

}

void exit\_reader(int reader\_id) {

pthread\_mutex\_lock(&mutex);

printf("Reader %d is exiting the critical section.\n", reader\_id);

readers--;

if (readers == 0) {

pthread\_mutex\_unlock(&writeLock);

}

pthread\_mutex\_unlock(&mutex);

}

void enter\_writer(int writer\_id) {

pthread\_mutex\_lock(&writeLock);

printf("Writer %d is entering the critical section.\n", writer\_id);

}

void exit\_writer(int writer\_id) {

printf("Writer %d is exiting the critical section.\n", writer\_id);

pthread\_mutex\_unlock(&writeLock);

}

--------------- 5 Producer Consumer Mutex -----------------------------

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

int item\_count = 0; // Tracks the number of items in the buffer

pthread\_mutex\_t mutex;

pthread\_cond\_t not\_full;

pthread\_cond\_t not\_empty;

void\* producer(void\* arg);

void\* consumer(void\* arg);

void add\_to\_buffer(int item);

int remove\_from\_buffer();

void show\_menu();

int is\_buffer\_full();

int is\_buffer\_empty();

int producer\_count = 1;

int consumer\_count = 1;

int main() {

pthread\_t p\_thread, c\_thread;

int choice, item;

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&not\_full, NULL);

pthread\_cond\_init(&not\_empty, NULL);

while (1) {

show\_menu();

scanf("%d", &choice);

switch (choice) {

case 1:

if (is\_buffer\_full()) {

printf("Buffer is full! Cannot produce more items.\n");

break;

}

printf("Enter the item to produce: ");

scanf("%d", &item);

// Allocate memory for the item

int\* item\_ptr = malloc(sizeof(int));

\*item\_ptr = item;

pthread\_create(&p\_thread, NULL, producer, item\_ptr);

pthread\_join(p\_thread, NULL);

producer\_count++;

break;

case 2:

if (is\_buffer\_empty()) {

printf("Buffer is empty! Nothing to consume.\n");

break;

}

pthread\_create(&c\_thread, NULL, consumer, NULL);

pthread\_join(c\_thread, NULL);

consumer\_count++;

break;

case 3:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice, try again.\n");

}

}

pthread\_mutex\_destroy(&mutex);

pthread\_cond\_destroy(&not\_full);

pthread\_cond\_destroy(&not\_empty);

return 0;

}

void\* producer(void\* arg) {

int item = \*(int\*)arg; // Dereference the pointer to get the item

free(arg); // Free the allocated memory

printf("Producer %d is trying to produce an item...\n", producer\_count);

pthread\_mutex\_lock(&mutex);

while (is\_buffer\_full()) {

pthread\_cond\_wait(&not\_full, &mutex); // Wait until buffer is not full

}

printf("Producer %d is adding item to buffer...\n", producer\_count);

add\_to\_buffer(item);

printf("Produced item by Producer %d: %d\n", producer\_count, item);

printf("Current buffer items: %d\n", item\_count);

pthread\_cond\_signal(&not\_empty); // Signal that the buffer is not empty

pthread\_mutex\_unlock(&mutex);

printf("Producer %d has left the buffer.\n", producer\_count);

return NULL;

}

void\* consumer(void\* arg) {

int item;

printf("Consumer %d is trying to consume an item...\n", consumer\_count);

pthread\_mutex\_lock(&mutex);

while (is\_buffer\_empty()) {

pthread\_cond\_wait(&not\_empty, &mutex); // Wait until buffer is not empty

}

printf("Consumer %d is removing item from buffer...\n", consumer\_count);

item = remove\_from\_buffer();

printf("Consumed item by Consumer %d: %d\n", consumer\_count, item);

printf("Current buffer items: %d\n", item\_count);

pthread\_cond\_signal(&not\_full); // Signal that the buffer is not full

pthread\_mutex\_unlock(&mutex);

printf("Consumer %d has left the buffer.\n", consumer\_count);

return NULL;

}

void add\_to\_buffer(int item) {

buffer[in] = item;

in = (in + 1) % BUFFER\_SIZE;

item\_count++; // Increment item count when adding an item

}

int remove\_from\_buffer() {

int item = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

item\_count--; // Decrement item count when removing an item

return item;

}

void show\_menu() {

printf("\n----Producer-Consumer Menu----\n");

printf("1. Produce an item\n");

printf("2. Consume an item\n");

printf("3. Exit\n");

printf("Enter your choice: ");

}

int is\_buffer\_full() {

return item\_count == BUFFER\_SIZE; // Buffer is full if item\_count equals BUFFER\_SIZE

}

int is\_buffer\_empty() {

return item\_count == 0; // Buffer is empty if item\_count is 0

}

/\*

1

10

2

10

\*/

/\*

Here’s a detailed explanation of each part of the code, function by function:

### \*\*Global Variables and Initialization\*\*

1. \*\*`buffer[BUFFER\_SIZE]`\*\*:

- A shared array (or buffer) of size `BUFFER\_SIZE` (which is 5 in this case). This is used to hold the items that are produced and consumed by the producer and consumer threads.

2. \*\*`in`, `out`, `item\_count`\*\*:

- `in`: This is the index where the next item produced will be placed in the buffer.

- `out`: This is the index from where the next item will be consumed from the buffer.

- `item\_count`: This tracks the current number of items in the buffer. It is used to determine if the buffer is full or empty.

3. \*\*`pthread\_mutex\_t mutex`\*\*:

- This is a mutex (mutual exclusion) lock that ensures that only one thread (either producer or consumer) can access the shared buffer at a time. It is used to protect the critical section of the code where the buffer is modified.

4. \*\*`pthread\_cond\_t not\_full`, `pthread\_cond\_t not\_empty`\*\*:

- These are condition variables used for synchronization.

- `not\_full`: This condition variable is used to signal the producer when the buffer is not full and it can safely add an item to the buffer.

- `not\_empty`: This condition variable is used to signal the consumer when the buffer is not empty and it can safely consume an item.

### \*\*`main` Function\*\*

The `main` function is responsible for managing the user interaction, creating threads, and invoking the producer and consumer threads based on the user’s menu choices. It performs the following tasks:

1. \*\*Initialize mutex and condition variables\*\* using `pthread\_mutex\_init()` and `pthread\_cond\_init()` before starting the threads.

2. \*\*User input loop\*\*:

- The user is prompted to select from a menu with three options: producing an item, consuming an item, or exiting.

- If the user chooses to produce, the program checks if the buffer is full. If it is not full, it creates a producer thread, passes the item to produce, and waits for the thread to finish.

- If the user chooses to consume, the program checks if the buffer is empty. If it is not empty, it creates a consumer thread and waits for the thread to finish.

- The program keeps looping until the user chooses to exit (`choice 3`).

3. \*\*Cleanup\*\*:

- After exiting the loop, the program calls `pthread\_mutex\_destroy()` and `pthread\_cond\_destroy()` to clean up the mutex and condition variables before exiting the program.

### \*\*`producer` Function\*\*

The `producer` function is executed by the producer thread. It performs the following tasks:

1. \*\*Acquire the mutex\*\* using `pthread\_mutex\_lock()`. This prevents any other thread (consumer) from modifying the buffer while the producer is working on it.

2. \*\*Check if the buffer is full\*\* using `is\_buffer\_full()`:

- If the buffer is full, the producer will wait for the `not\_full` condition variable to be signaled, indicating that space is available in the buffer.

- The producer waits by calling `pthread\_cond\_wait(&not\_full, &mutex)`, which releases the mutex temporarily and waits for a signal (when the buffer is no longer full).

3. \*\*Produce the item\*\*:

- Once the buffer is not full, the producer adds the item to the buffer by calling `add\_to\_buffer()`. The `in` index is updated, and `item\_count` is incremented to reflect the new item in the buffer.

4. \*\*Signal the consumer\*\* using `pthread\_cond\_signal(&not\_empty)`. This signals the consumer thread that the buffer is no longer empty, and it can now consume an item.

5. \*\*Release the mutex\*\* using `pthread\_mutex\_unlock()` to allow other threads to access the buffer.

6. \*\*Output the current state\*\*:

- The producer prints messages indicating the item produced and the current number of items in the buffer.

7. \*\*Free the allocated memory\*\* for the item pointer at the end of the function to prevent memory leaks.

### \*\*`consumer` Function\*\*

The `consumer` function is executed by the consumer thread. It performs the following tasks:

1. \*\*Acquire the mutex\*\* using `pthread\_mutex\_lock()`. This ensures mutual exclusion when accessing the shared buffer.

2. \*\*Check if the buffer is empty\*\* using `is\_buffer\_empty()`:

- If the buffer is empty, the consumer waits for the `not\_empty` condition variable to be signaled, indicating that there is an item available in the buffer.

- The consumer waits by calling `pthread\_cond\_wait(&not\_empty, &mutex)`, which releases the mutex temporarily and waits for a signal (when the buffer is no longer empty).

3. \*\*Consume the item\*\*:

- Once the buffer is not empty, the consumer removes an item from the buffer by calling `remove\_from\_buffer()`. The `out` index is updated, and `item\_count` is decremented to reflect the item that was removed.

4. \*\*Signal the producer\*\* using `pthread\_cond\_signal(&not\_full)`. This signals the producer thread that space is available in the buffer for producing more items.

5. \*\*Release the mutex\*\* using `pthread\_mutex\_unlock()` to allow other threads to access the buffer.

6. \*\*Output the current state\*\*:

- The consumer prints messages indicating the item consumed and the current number of items in the buffer.

### \*\*Buffer Management Functions\*\*

1. \*\*`add\_to\_buffer(int item)`\*\*:

- This function is responsible for adding an item to the buffer.

- It places the item in the position indicated by `in`, then updates `in` to the next position, wrapping around to the beginning of the buffer when the end is reached (`in = (in + 1) % BUFFER\_SIZE`).

- The `item\_count` is incremented to reflect the new item in the buffer.

2. \*\*`remove\_from\_buffer()`\*\*:

- This function is responsible for removing an item from the buffer.

- It retrieves the item from the position indicated by `out`, then updates `out` to the next position, wrapping around to the beginning of the buffer when the end is reached (`out = (out + 1) % BUFFER\_SIZE`).

- The `item\_count` is decremented to reflect the removal of an item.

### \*\*Utility Functions\*\*

1. \*\*`show\_menu()`\*\*:

- This function displays the menu to the user, offering three choices: produce an item, consume an item, or exit the program.

2. \*\*`is\_buffer\_full()`\*\*:

- This function checks if the buffer is full by comparing `item\_count` with `BUFFER\_SIZE`. If `item\_count` equals `BUFFER\_SIZE`, the buffer is full, and the function returns `1` (true).

3. \*\*`is\_buffer\_empty()`\*\*:

- This function checks if the buffer is empty by checking if `item\_count` is `0`. If the buffer is empty, the function returns `1` (true).

### \*\*Summary of Thread Synchronization\*\*

- \*\*Producer-Consumer Problem\*\*: This program implements the classical producer-consumer problem using \*\*threads\*\*, \*\*mutex\*\* for mutual exclusion, and \*\*condition variables\*\* (`not\_full` and `not\_empty`) for synchronization between the producer and consumer.

- \*\*Mutex\*\* ensures that only one thread (either producer or consumer) can access the shared buffer at a time.

- \*\*Condition variables\*\* are used to signal the producer when the buffer has space, and to signal the consumer when there is an item to consume.

### \*\*Thread Management\*\*

- The program creates threads using `pthread\_create()` for both producers and consumers. After each thread finishes its work, the main thread waits for them to finish using `pthread\_join()` before proceeding with the next operation.

This program is a practical implementation of \*\*thread synchronization\*\* in a multi-threaded environment and ensures that the buffer operates correctly without conflicts or race conditions between the producer and consumer threads.

\*/

----------------------- 6 Reader Writer Semaphore ---------------------

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

sem\_t writeLock;

int readers = 0;

int data = 0;

void\* reader(void\* arg);

void\* writer(void\* arg);

void enter\_reader(int reader\_id);

void exit\_reader(int reader\_id);

void enter\_writer(int writer\_id);

void exit\_writer(int writer\_id);

int main() {

pthread\_t r\_threads[5], w\_threads[3];

sem\_init(&mutex, 0, 1);

sem\_init(&writeLock, 0, 1);

for (int i = 0; i < 5; i++) {

pthread\_create(&r\_threads[i], NULL, reader, (void\*)(size\_t)(i+1));

}

for (int i = 0; i < 3; i++) {

pthread\_create(&w\_threads[i], NULL, writer, (void\*)(size\_t)(i+1));

}

for (int i = 0; i < 5; i++) {

pthread\_join(r\_threads[i], NULL);

}

for (int i = 0; i < 3; i++) {

pthread\_join(w\_threads[i], NULL);

}

sem\_destroy(&mutex);

sem\_destroy(&writeLock);

return 0;

}

void\* reader(void\* arg) {

int reader\_id = (int)(size\_t)arg;

sleep(rand() % 5);

enter\_reader(reader\_id);

printf("Reader %d: Reading data: %d\n", reader\_id, data);

exit\_reader(reader\_id);

return NULL;

}

void\* writer(void\* arg) {

int writer\_id = (int)(size\_t)arg;

sleep(rand() % 5);

enter\_writer(writer\_id);

data = rand() % 100;

printf("Writer %d: Writing data: %d\n", writer\_id, data);

exit\_writer(writer\_id);

return NULL;

}

void enter\_reader(int reader\_id) {

sem\_wait(&mutex);

readers++;

if (readers == 1) {

sem\_wait(&writeLock);

}

printf("Reader %d is entering the critical section.\n", reader\_id);

sem\_post(&mutex);

}

void exit\_reader(int reader\_id) {

sem\_wait(&mutex);

printf("Reader %d is exiting the critical section.\n", reader\_id);

readers--;

if (readers == 0) {

sem\_post(&writeLock);

}

sem\_post(&mutex);

}

void enter\_writer(int writer\_id) {

sem\_wait(&writeLock);

printf("Writer %d is entering the critical section.\n", writer\_id);

}

void exit\_writer(int writer\_id) {

printf("Writer %d is exiting the critical section.\n", writer\_id);

sem\_post(&writeLock);

}

/\*

his code implements a Readers-Writers problem using semaphores to manage concurrent access to a shared resource (data). The problem involves multiple reader threads and writer threads accessing the shared data, with the following rules:

Readers can access the data concurrently (i.e., multiple readers can read the data at the same time).

Writers require exclusive access to the data (i.e., no readers or other writers can access the data while a writer is writing).

The implementation uses two semaphores (mutex and writeLock) to synchronize access to the shared resource.

Key Concepts:

Semaphores:

mutex: This semaphore is used to manage access to the readers counter, which tracks how many readers are currently accessing the data. It ensures that only one thread (reader or writer) modifies the readers counter at a time.

writeLock: This semaphore ensures that only one writer can access the data at a time and prevents readers from accessing the data while a writer is writing. It is locked when a writer enters and unlocked when the writer exits.

Readers-Writers Problem:

Readers can read the shared data concurrently, but writers need exclusive access to the data, meaning no other readers or writers can access the data during the write operation.

The program follows a strategy known as "first reader blocks writer". The first reader blocks writers by acquiring the writeLock, and the last reader releases the writeLock.

Code Breakdown:

Main Function:

Thread Creation:

The program creates 5 reader threads and 3 writer threads using pthread\_create.

Each reader and writer thread is executed by the reader and writer functions respectively.

Thread Joining:

After all threads are created, pthread\_join is used to wait for each reader and writer thread to finish before the program terminates.

Semaphore Initialization/Destruction:

sem\_init(&mutex, 0, 1) initializes mutex to allow only one thread to modify the readers count at a time.

sem\_init(&writeLock, 0, 1) initializes writeLock to allow only one thread (either a writer or the first reader) to access the critical section at a time.

After all threads are done, sem\_destroy cleans up the semaphores.

Reader Function:

Random Delay:

Each reader thread waits for a random period before trying to enter the critical section using sleep(rand() % 5). This simulates real-world variability in thread scheduling.

Enter Reader (enter\_reader):

The reader first locks the mutex to increment the readers counter safely.

If this is the first reader (i.e., readers == 1), it locks writeLock to prevent writers from entering the critical section.

After the check and potentially locking the writeLock, the reader prints that it is entering the critical section.

The mutex is unlocked, and the reader can proceed to read the data.

Exit Reader (exit\_reader):

After finishing reading, the reader locks mutex again to safely decrement the readers counter.

If this is the last reader (i.e., readers == 0), it releases the writeLock, allowing any waiting writers to enter.

The reader then prints that it is exiting the critical section and unlocks mutex.

Writer Function:

Random Delay:

Each writer thread waits for a random period before trying to enter the critical section using sleep(rand() % 5).

Enter Writer (enter\_writer):

The writer locks the writeLock semaphore. This ensures exclusive access to the data, so no readers or other writers can access the critical section.

The writer then prints that it is entering the critical section.

Exit Writer (exit\_writer):

After writing the data, the writer prints that it is exiting the critical section.

It then releases the writeLock semaphore, allowing other writers or the first reader to enter the critical section.

Data and Synchronization:

Data: The shared variable data represents the resource that readers and writers are accessing. Writers modify this data by setting it to a random value, while readers print the current value of data.

Synchronization:

The use of mutex ensures that changes to the readers count are atomic.

The use of writeLock ensures that when a writer is writing, no readers can access the data and no other writers can write.

The first reader locks the writeLock semaphore to block writers, and the last reader releases it to allow writers to write.

\*/

--------------- 7 Producer Consumer Semaphore ---------------------

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define BUFFER\_SIZE 5 // Reduced buffer size for quicker full/empty demonstration

int buffer[BUFFER\_SIZE]; // Shared buffer

int count = 0; // Counter for items in the buffer

pthread\_mutex\_t mutex; // Mutex for critical section

// Function to display the buffer's current state

void displayBuffer() {

printf("Buffer: [ ");

for (int i = 0; i < BUFFER\_SIZE; i++) {

if (i < count)

printf("%d ", buffer[i]);

else

printf("- "); // Empty slot

}

printf("] ");

if (count == 0) {

printf("(Buffer is empty)\n");

} else if (count == BUFFER\_SIZE) {

printf("(Buffer is full)\n");

} else {

printf("\n");

}

}

// Producer function

void\* producer(void\* arg) {

while (1) {

int item = rand() % 100; // Produce a random item

pthread\_mutex\_lock(&mutex); // Wait to access the buffer

// Wait if buffer is full (busy-waiting approach)

while (count == BUFFER\_SIZE) {

pthread\_mutex\_unlock(&mutex);

usleep(100); // Sleep a bit to avoid busy-waiting too aggressively

pthread\_mutex\_lock(&mutex);

}

buffer[count] = item; // Produce item into buffer

count++;

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

displayBuffer(); // Display buffer state

pthread\_mutex\_unlock(&mutex); // Release buffer

usleep(rand() % 2000); // Simulate production time

}

}

// Consumer function

void\* consumer(void\* arg) {

while (1) {

pthread\_mutex\_lock(&mutex); // Wait to access the buffer

// Wait if buffer is empty (busy-waiting approach)

while (count == 0) {

pthread\_mutex\_unlock(&mutex);

usleep(100); // Sleep a bit to avoid busy-waiting too aggressively

pthread\_mutex\_lock(&mutex);

}

int item = buffer[--count]; // Consume item from buffer

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

displayBuffer(); // Display buffer state

pthread\_mutex\_unlock(&mutex); // Release buffer

usleep(rand() % 2000); // Simulate consumption time

}

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// Initialize the mutex

pthread\_mutex\_init(&mutex, NULL);

// Create producer and consumer threads

pthread\_create(&prod\_thread, NULL, producer, NULL);

pthread\_create(&cons\_thread, NULL, consumer, NULL);

// Wait for threads to finish (optional, typically runs indefinitely)

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

// Clean up

pthread\_mutex\_destroy(&mutex);

return 0;

}

/\*

Sure! Let's go through the code step by step to explain how it works. You can use this explanation in your lab examination.

Overview:

This program is a Producer-Consumer problem implementation, which demonstrates synchronization between two types of threads (producers and consumers) operating on a shared buffer. The goal is to ensure that producers add items to the buffer when there is space, and consumers remove items from the buffer when it is not empty. The threads synchronize their actions using mutexes (for mutual exclusion) and busy-waiting.

Key Concepts:

Producer Thread: A thread that produces an item and places it in a shared buffer.

Consumer Thread: A thread that consumes an item by removing it from the shared buffer.

Shared Buffer: An array that stores the items. It has a fixed size, and its state is managed using a count variable.

Mutex (Mutual Exclusion): Used to ensure that only one thread can access the shared buffer at any given time. This prevents race conditions.

Busy-Waiting: Instead of using semaphores or condition variables, this approach uses loops and sleeps to wait for the buffer to be in a valid state (full or empty).

Explanation by Sections:

Global Variables:

buffer[BUFFER\_SIZE]: An array representing the shared buffer. It can hold a fixed number of items (in this case, BUFFER\_SIZE = 5).

count: Keeps track of how many items are currently in the buffer.

mutex: A mutex (or lock) that is used to protect the buffer from concurrent access by multiple threads.

Function: displayBuffer():

This function is used to print the current state of the buffer.

It loops through the buffer, printing the items that have been produced and placing a - symbol in the empty slots.

After displaying the buffer, it also prints whether the buffer is empty or full based on the count.

Producer Function (producer()):

The producer is an infinite loop that continuously produces items.

A random item is generated using rand() % 100 to simulate the production of different items.

Mutex Locking: The producer locks the mutex using pthread\_mutex\_lock(&mutex) to ensure that no other thread (consumer or producer) can access the shared buffer during this time.

Busy-Waiting for Full Buffer: If the buffer is full (i.e., count == BUFFER\_SIZE), the producer releases the mutex (pthread\_mutex\_unlock(&mutex)) and sleeps for a short time (usleep(100)), allowing other threads (consumer) to access the buffer. Once the buffer has space again, the producer re-acquires the mutex and proceeds to add an item to the buffer.

Item Production: After ensuring there's space in the buffer, the producer adds the item to the buffer (buffer[count] = item) and increments the count to reflect the new number of items in the buffer.

After producing the item, the producer prints the produced item and the current state of the buffer, then releases the mutex with pthread\_mutex\_unlock(&mutex).

The producer thread then sleeps for a random period (usleep(rand() % 2000)) to simulate the time it takes to produce an item.

Consumer Function (consumer()):

The consumer is also an infinite loop, which continuously consumes items from the buffer.

Mutex Locking: The consumer locks the mutex at the beginning to ensure exclusive access to the buffer.

Busy-Waiting for Empty Buffer: If the buffer is empty (count == 0), the consumer releases the mutex and sleeps for a short time (usleep(100)), allowing the producer to add items. Once the buffer is not empty, the consumer re-acquires the mutex and continues.

Item Consumption: After ensuring that the buffer is not empty, the consumer removes an item from the buffer (int item = buffer[--count]) and prints the consumed item.

After consuming the item, the consumer prints the current state of the buffer and releases the mutex with pthread\_mutex\_unlock(&mutex).

The consumer then sleeps for a random period (usleep(rand() % 2000)) to simulate the time it takes to consume an item.

Main Function (main()):

The main() function is where the threads are created and initialized.

Mutex Initialization: The mutex (pthread\_mutex\_init(&mutex, NULL)) is initialized to protect the shared buffer.

Thread Creation:

Three producer threads are created: Thread producer1, Thread producer2, Thread producer3.

Two consumer threads are created: Thread consumer1, Thread consumer2.

Thread Execution: Each of the threads is started using the pthread\_create() function. The producer and consumer threads will run in parallel, each following their respective logic for producing and consuming items.

Thread Joining: The pthread\_join() calls are used to wait for the producer and consumer threads to finish their execution (although they run indefinitely in this case).

Mutex Destruction: Once the threads finish, the mutex is destroyed using pthread\_mutex\_destroy(&mutex) to clean up.

Key Synchronization Concepts:

Mutex for Critical Section: The mutex ensures that only one thread (either a producer or a consumer) can access the shared buffer at a time. This prevents concurrent modification of the buffer, which could lead to race conditions.

Busy-Waiting: In this version, the producer and consumer check the buffer's state (count) and actively wait if the buffer is full or empty. They release the mutex when they are waiting and re-lock it when the condition changes. This simple synchronization method avoids complex semaphores and is often used for small-scale, simple concurrency problems.

Execution Flow:

The producer threads continuously produce items and add them to the buffer, while the consumer threads consume the items.

When the buffer is full, the producer waits by releasing the mutex and sleeping. Similarly, when the buffer is empty, the consumer waits.

The buffer's state is displayed after each action (production or consumption), showing the items in the buffer and whether it is full or empty.

\*/

------------------------ 8 Dinning Philopsopher ---------------

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

#define N 5

sem\_t forks[N];

void \*philosopher(void \*arg) {

int id = \*((int \*)arg);

for (int i = 0; i < 3; i++) {

printf("Philosopher %d is thinking.\n", id);

usleep(100000);

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % N]);

printf("Philosopher %d is eating.\n", id);

usleep(100000);

sem\_post(&forks[id]);

sem\_post(&forks[(id + 1) % N]);

printf("Philosopher %d finished eating.\n", id);

usleep(100000);

}

return NULL;

}

int main() {

pthread\_t philosophers[N];

int ids[N];

for (int i = 0; i < N; i++)

sem\_init(&forks[i], 0, 1);

for (int i = 0; i < N; i++) {

ids[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

for (int i = 0; i < N; i++)

pthread\_join(philosophers[i], NULL);

for (int i = 0; i < N; i++)

sem\_destroy(&forks[i]);

return 0;

}

/\*

pthread.h: Provides the functionality to create and manage threads.

semaphore.h: Provides the functionality to use semaphores for synchronization.

stdio.h: Standard I/O functions like printf().

unistd.h: Provides access to the POSIX operating system API (for usleep() to simulate delay).

2. Defining Constants:

3. Declaring Semaphores for Forks:

An array forks[] of semaphores is defined. Each philosopher requires two forks to eat.

Each semaphore represents a fork that a philosopher can pick up or put down

The philosopher function is the entry point for each philosopher (each thread).

The id of the philosopher is passed as an argument to the thread function. Each philosopher has a unique ID (from 0 to 4 in this case).

5. Thinking and Eating Loop:

Each philosopher will think, eat, and then finish eating for a total of 3 times.

Thinking: The philosopher spends time thinking (simulated using usleep(100000)).

Eating: The philosopher needs two forks to eat:

The left fork is forks[id].

The right fork is forks[(id + 1) % N]. The % N ensures that for philosopher 4, the right fork is forks[0], forming a circular table.

Synchronization:

sem\_wait(&forks[id]) and sem\_wait(&forks[(id + 1) % N]) ensure the philosopher waits for both forks before eating.

sem\_post(&forks[id]) and sem\_post(&forks[(id + 1) % N]) are used to release the forks after eating, allowing other philosophers to pick them up.

Eating Time: Simulated with another usleep(100000) to represent the time taken by a philosopher to eat.

6. Main Function:

he philosophers[] array holds the thread IDs for each philosopher.

The ids[] array stores the unique ID for each philosopher.

The loop sem\_init(&forks[i], 0, 1) initializes each semaphore (forks[i]) to 1, meaning each fork is initially available.

7. Creating Threads (Philosophers):

For each philosopher, a new thread is created using pthread\_create().

Each philosopher is assigned an ID (from 0 to 4).

The philosopher function is passed as the thread function, and the philosopher's ID is passed as an argument.

8. Joining Threads:

The pthread\_join() function is used to wait for each philosopher's thread to finish before the program exits.

This ensures that the main program waits for all philosophers to finish their work (thinking and eating).

9. Destroying Semaphores:

After all philosophers have finished, the semaphores are destroyed using sem\_destroy(), which cleans up the semaphore resources.

Understanding Synchronization with Semaphores:

The semaphores control access to the forks. Each fork is a shared resource between two philosophers.

sem\_wait() blocks the philosopher if the fork is already taken, ensuring that only one philosopher can use a fork at a time.

sem\_post() releases the fork, allowing another philosopher to use it.

Deadlock Prevention:

There is still a risk of deadlock in this simple model: if every philosopher picks up the left fork at the same time, they will all wait for the right fork, resulting in a deadlock.

One way to avoid deadlock would be to implement a strategy where one philosopher (say philosopher 0) picks up the forks in reverse order (right first), but this code does not address that specifically.

\*/

------------------------ 9 **9\_FCFS\_WT\_TAT\_NP\_Aditi.**

#include <stdio.h>

struct Process {

int pid; // Process ID

int arrivalTime;

int burstTime;

int finishTime;

int turnAroundTime;

int waitingTime;

};

void calculateTimes(struct Process processes[], int n) {

int currentTime = 0;

for (int i = 0; i < n; i++) {

// If the process arrives after the current time, update the current time to the arrival time of the process

if (currentTime < processes[i].arrivalTime) {

currentTime = processes[i].arrivalTime;

}

// Finish time is the current time plus the burst time

processes[i].finishTime = currentTime + processes[i].burstTime;

// Turnaround time is finish time minus arrival time

processes[i].turnAroundTime = processes[i].finishTime - processes[i].arrivalTime;

// Waiting time is turnaround time minus burst time

processes[i].waitingTime = processes[i].turnAroundTime - processes[i].burstTime;

// Update current time to finish time of the current process

currentTime = processes[i].finishTime;

}

}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

for (int i = 0; i < n; i++) {

printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",

processes[i].pid,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].finishTime,

processes[i].turnAroundTime,

processes[i].waitingTime);

}

float totalTurnAroundTime = 0, totalWaitingTime = 0;

for (int i = 0; i < n; i++) {

totalTurnAroundTime += processes[i].turnAroundTime;

totalWaitingTime += processes[i].waitingTime;

}

printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);

printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

for (int i = 0; i < n; i++) {

processes[i].pid = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].pid);

scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);

}

// Sort processes by arrival time (FCFS scheduling)

for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

if (processes[i].arrivalTime > processes[j].arrivalTime) {

struct Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

calculateTimes(processes, n);

displayResults(processes, n);

return 0;

}

/\*

Enter the number of processes: 3

Enter arrival time and burst time for process 1: 0 4

Enter arrival time and burst time for process 2: 1 3

Enter arrival time and burst time for process 3: 2 2

\*/

/\*

Process: A program that needs to be executed. Each process has:

Arrival Time: When the process arrives in the system.

Burst Time: How much time the process needs to complete execution.

Finish Time: When the process finishes execution.

Turnaround Time: The total time the process spends from arrival to completion (Finish Time - Arrival Time).

Waiting Time: The time a process spends waiting in the ready queue (Turnaround Time - Burst Time).

FCFS Scheduling: Processes are executed in the order of their arrival. If two processes arrive at the same time, the one listed first in the input will execute first.

This structure (struct Process) stores all the details about a process: its ID (pid), arrival time, burst time, finish time, turnaround time, and waiting time.

2. calculateTimes Function:

This function calculates the finish time, turnaround time, and waiting time for each process.

The current time starts at 0 and keeps increasing as processes finish.

For each process, we check if it arrives after the current time, in which case the current time is updated to the process's arrival time.

The finish time is calculated by adding the burst time to the current time.

Turnaround time is the difference between the finish time and the arrival time.

Waiting time is the difference between turnaround time and burst time.

The current time is updated after each process finishes.

3. displayResults Function:

This function displays the details of all processes and calculates the average turnaround time and waiting time.

It first prints the details for each process (ID, arrival time, burst time, finish time, turnaround time, waiting time).

Then it calculates the total turnaround time and waiting time for all processes.

Finally, it computes and prints the average turnaround time and average waiting time.

4. Main Function:

The main() function is where the program begins execution.

First, it asks for the number of processes (n) from the user.

Then, it creates an array of n processes and asks the user to input the arrival time and burst time for each process.

The processes are sorted by arrival time to implement the First-Come, First-Serve (FCFS) scheduling.

It then calls calculateTimes() to compute the times for each process.

Finally, it calls displayResults() to display the results and compute the averages.

How FCFS Works in This Code:

The processes are sorted based on their arrival time. This ensures that the first process to arrive is the first to execute.

The program calculates the finish time, turnaround time, and waiting time for each process based on the order they are scheduled to execute.

Turnaround time tells you how long a process takes from arrival to completion, and waiting time tells you how much time the process was sitting in the ready queue before it got executed.

\*/

//fcfs

#include <stdio.h>

void find\_fcfs(int n, int bt[], int at[], int ft[], int tat[], int wt[]) {

int ct = 0;

for (int i = 0; i < n; i++) {

if (ct < at[i])

ct = at[i];

ft[i] = ct + bt[i];

ct = ft[i];

tat[i] = ft[i] - at[i];

wt[i] = tat[i] - bt[i];

}

}

void print\_times(int n, int at[], int bt[], int ft[], int tat[], int wt[]) {

printf("Process\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting 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", i+1, at[i], bt[i], ft[i], tat[i], wt[i]);

}

}

int main() {

int n, bt[20], at[20], ft[20], tat[20], wt[20];

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &bt[i]);

}

printf("Enter arrival times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &at[i]);

}

find\_fcfs(n, bt, at, ft, tat, wt);

print\_times(n, at, bt, ft, tat, wt);

return 0;

}

---------------------- 1**0\_Shortest\_Job\_First\_P\_NP.c ---------------------**

#include <stdio.h>

#include <stdbool.h>

struct Process {

int pid; // Process ID

int arrivalTime;

int burstTime;

int remainingTime; // For preemptive SJF

int finishTime;

int turnAroundTime;

int waitingTime;

bool isCompleted;

};

// Function for Non-Preemptive SJF

void sjfNonPreemptive(struct Process processes[], int n) {

int currentTime = 0, completed = 0;

while (completed < n) {

int minIndex = -1;

int minBurstTime = 1e9;

// Select the process with the smallest burst time that has arrived

for (int i = 0; i < n; i++) {

if (!processes[i].isCompleted && processes[i].arrivalTime <= currentTime &&

processes[i].burstTime < minBurstTime) {

minBurstTime = processes[i].burstTime;

minIndex = i;

}

}

if (minIndex == -1) {

currentTime++;

} else {

// Calculate the finish time, turnaround time, and waiting time

processes[minIndex].finishTime = currentTime + processes[minIndex].burstTime;

processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;

processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;

processes[minIndex].isCompleted = true;

currentTime = processes[minIndex].finishTime;

completed++;

}

}

}

// Function for Preemptive SJF

void sjfPreemptive(struct Process processes[], int n) {

int currentTime = 0, completed = 0;

int minIndex = -1;

int minRemainingTime = 1e9;

while (completed < n) {

minIndex = -1;

minRemainingTime = 1e9;

// Select the process with the smallest remaining time that has arrived

for (int i = 0; i < n; i++) {

if (processes[i].arrivalTime <= currentTime && !processes[i].isCompleted &&

processes[i].remainingTime < minRemainingTime) {

minRemainingTime = processes[i].remainingTime;

minIndex = i;

}

}

if (minIndex != -1) {

processes[minIndex].remainingTime--;

currentTime++;

// If process is completed

if (processes[minIndex].remainingTime == 0) {

processes[minIndex].finishTime = currentTime;

processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;

processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;

processes[minIndex].isCompleted = true;

completed++;

}

} else {

currentTime++;

}

}

}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

for (int i = 0; i < n; i++) {

printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",

processes[i].pid,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].finishTime,

processes[i].turnAroundTime,

processes[i].waitingTime);

}

float totalTurnAroundTime = 0, totalWaitingTime = 0;

for (int i = 0; i < n; i++) {

totalTurnAroundTime += processes[i].turnAroundTime;

totalWaitingTime += processes[i].waitingTime;

}

printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);

printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

int n, choice;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

for (int i = 0; i < n; i++) {

processes[i].pid = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].pid);

scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);

processes[i].remainingTime = processes[i].burstTime;

processes[i].isCompleted = false;

}

printf("Choose Scheduling:\n1. Non-Preemptive SJF\n2. Preemptive SJF\n");

scanf("%d", &choice);

if (choice == 1) {

sjfNonPreemptive(processes, n);

} else if (choice == 2) {

sjfPreemptive(processes, n);

} else {

printf("Invalid choice!\n");

return 0;

}

displayResults(processes, n);

return 0;

}

/\*

Enter the number of processes: 4

Enter arrival time and burst time for process 1: 0 7

Enter arrival time and burst time for process 2: 2 4

Enter arrival time and burst time for process 3: 4 1

Enter arrival time and burst time for process 4: 5 4

Choose Scheduling:

1. Non-Preemptive SJF

\*/

/\*

The Process struct defines a single process with various attributes:

pid: Process ID for identification.

arrivalTime: The time when the process arrives in the queue.

burstTime: The total time required for the process to complete.

remainingTime: Used for preemptive SJF to track how much burst time is left.

finishTime: The time when the process finishes execution.

turnAroundTime: The total time from arrival to completion.

waitingTime: Time spent waiting to be executed.

isCompleted: Boolean flag indicating if the process has completed.

2. sjfNonPreemptive Function

Absolutely! Let's go through the code step-by-step to understand how it works, focusing on its logic and structure.

Code Breakdown

1. Struct Definition

c

Copy code

struct Process {

int pid; // Process ID

int arrivalTime;

int burstTime;

int remainingTime; // For preemptive SJF

int finishTime;

int turnAroundTime;

int waitingTime;

bool isCompleted;

};

The Process struct defines a single process with various attributes:

pid: Process ID for identification.

arrivalTime: The time when the process arrives in the queue.

burstTime: The total time required for the process to complete.

remainingTime: Used for preemptive SJF to track how much burst time is left.

finishTime: The time when the process finishes execution.

turnAroundTime: The total time from arrival to completion.

waitingTime: Time spent waiting to be executed.

isCompleted: Boolean flag indicating if the process has completed.

2. sjfNonPreemptive Function

c

Copy code

void sjfNonPreemptive(struct Process processes[], int n)

This function implements Non-Preemptive Shortest Job First (SJF) scheduling. Here’s the breakdown of the algorithm inside this function:

Initialize currentTime and completed:

currentTime keeps track of the current time in the system.

completed counts how many processes have finished execution.

Main Loop:

The loop continues until all processes are completed (completed < n).

Process Selection:

For each time unit, the algorithm searches for the process with the shortest burstTime that has arrived (arrivalTime <= currentTime) and isn’t completed.

If a process meets these conditions, it becomes the candidate for execution.

Execution and Calculation:

If no suitable process is found (minIndex == -1), currentTime increments by 1, simulating an idle period.

If a process is selected (minIndex != -1), it finishes execution without interruption:

finishTime is set to currentTime + burstTime.

turnAroundTime and waitingTime are calculated based on finishTime, arrivalTime, and burstTime.

The selected process is marked as completed and currentTime moves forward.

3. sjfPreemptive Function

This function implements Preemptive SJF (Shortest Remaining Time First - SRTF) scheduling.

Initialize currentTime, completed, and Process Selection Variables:

currentTime tracks time progression.

completed tracks how many processes have finished.

minIndex and minRemainingTime help identify the process with the shortest remaining time.

Process Selection:

For each time unit, the function finds the process with the smallest remainingTime that has arrived and hasn’t completed.

If no process is found, currentTime increments (idle time).

If a process is found, it’s preempted every time a new process with a shorter remainingTime arrives.

Execution and Calculation:

The selected process is executed for 1 time unit, decrementing its remainingTime.

If a process completes (remainingTime == 0), its finishTime, turnAroundTime, and waitingTime are calculated.

The completed counter is incremented, and the process is marked as completed.

4. displayResults Function

This function displays the results, including PID, Arrival Time, Burst Time, Finish Time, Turnaround Time, and Waiting Time for each process. It also calculates and displays the average Turnaround Time and Waiting Time.

Display Table Headers:

Prints headers for each attribute of the process.

Loop Through Processes:

For each process, the function prints all calculated values (PID, Arrival, Burst, Finish, Turnaround, Waiting).

It also accumulates total turnaround and waiting times for calculating averages.

Calculate Averages:

The total turnaround and waiting times are divided by the number of processes to find averages, which are then printed.

5. main Function

The main function serves as the starting point for program execution.

Input Process Details:

The user is prompted to enter the number of processes.

For each process, arrival time and burst time are inputted and stored. remainingTime is initialized to burstTime for preemptive scheduling, and isCompleted is set to false.

Choose Scheduling Algorithm:

The user chooses between Non-Preemptive (1) and Preemptive SJF (2).

Based on the user’s choice, either sjfNonPreemptive or sjfPreemptive is called.

Display Results:

After scheduling, displayResults is called to print each process’s details and average times.

Summary

Non-Preemptive SJF executes the shortest available job without interruption, making it simple but potentially causing long waits for short processes arriving after long processes.

Preemptive SJF (SRTF) allows preemption, giving preference to new processes with shorter remaining times, reducing average waiting time but adding complexity.

\*/

// --------------------------------------------------------------------------------

#include <stdio.h>

#include <limits.h>

void find\_sjf\_np(int n, int bt[], int at[], int ft[], int tat[], int wt[]) {

int bt\_rem[n], done[n], time = 0, min\_bt, shortest;

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

for (int i = 0; i < n; i++) bt\_rem[i] = bt[i];

for (int i = 0; i < n; i++) {

min\_bt = INT\_MAX;

shortest = -1;

for (int j = 0; j < n; j++) {

if (at[j] <= time && !done[j] && bt\_rem[j] < min\_bt) {

min\_bt = bt\_rem[j];

shortest = j;

}

}

if (shortest != -1) {

time += bt\_rem[shortest];

ft[shortest] = time;

tat[shortest] = ft[shortest] - at[shortest];

wt[shortest] = tat[shortest] - bt[shortest];

done[shortest] = 1;

} else {

time++;

i--;

}

}

}

void find\_sjf\_p(int n, int bt[], int at[], int ft[], int tat[], int wt[]) {

int bt\_rem[n], time = 0, shortest = -1, min\_bt = INT\_MAX, count = 0;

for (int i = 0; i < n; i++) bt\_rem[i] = bt[i];

while (count < n) {

min\_bt = INT\_MAX;

shortest = -1;

for (int i = 0; i < n; i++) {

if (at[i] <= time && bt\_rem[i] < min\_bt && bt\_rem[i] > 0) {

min\_bt = bt\_rem[i];

shortest = i;

}

}

if (shortest != -1) {

time++;

bt\_rem[shortest]--;

if (bt\_rem[shortest] == 0) {

count++;

ft[shortest] = time;

tat[shortest] = ft[shortest] - at[shortest];

wt[shortest] = tat[shortest] - bt[shortest];

}

} else time++;

}

}

void print\_times(int n, int at[], int bt[], int ft[], int tat[], int wt[]) {

printf("Process\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting 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", i+1, at[i], bt[i], ft[i], tat[i], wt[i]);

}

}

int main() {

int n, bt[20], at[20], ft[20], tat[20], wt[20], choice;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &bt[i]);

}

printf("Enter arrival times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &at[i]);

}

printf("Choose SJF Scheduling Type:\n");

printf("1. Non-Preemptive SJF\n");

printf("2. Preemptive SJF\n");

printf("Enter your choice: ");

scanf("%d", &choice);

if (choice == 1) {

find\_sjf\_np(n, bt, at, ft, tat, wt);

} else if (choice == 2) {

find\_sjf\_p(n, bt, at, ft, tat, wt);

} else {

printf("Invalid choice!\n");

return 1;

}

print\_times(n, at, bt, ft, tat, wt);

return 0;

}

// input

// 3

// 4

// 1

// 5

// 0

// 2

// 3

// 2

---------------- **11\_Priority\_P\_NP.c --------------**

#include <stdio.h>

#include <limits.h>

void find\_priority\_np(int n, int bt[], int at[], int pr[], int ft[], int tat[], int wt[]) {

int done[n], min\_pr, time = 0, highest;

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

for (int i = 0; i < n; i++) {

min\_pr = INT\_MAX;

highest = -1;

for (int j = 0; j < n; j++) {

if (at[j] <= time && !done[j] && pr[j] < min\_pr) {

min\_pr = pr[j];

highest = j;

}

}

if (highest != -1) {

time += bt[highest];

ft[highest] = time;

tat[highest] = ft[highest] - at[highest];

wt[highest] = tat[highest] - bt[highest];

done[highest] = 1;

} else {

time++;

i--;

}

}

}

void find\_priority\_p(int n, int bt[], int at[], int pr[], int ft[], int tat[], int wt[]) {

int bt\_rem[n], time = 0, completed = 0, highest\_priority, min\_priority;

for (int i = 0; i < n; i++) bt\_rem[i] = bt[i];

while (completed < n) {

highest\_priority = -1;

min\_priority = INT\_MAX;

for (int i = 0; i < n; i++) {

if (at[i] <= time && bt\_rem[i] > 0 && pr[i] < min\_priority) {

min\_priority = pr[i];

highest\_priority = i;

}

}

if (highest\_priority == -1) {

time++;

continue;

}

bt\_rem[highest\_priority]--;

time++;

if (bt\_rem[highest\_priority] == 0) {

completed++;

ft[highest\_priority] = time;

tat[highest\_priority] = ft[highest\_priority] - at[highest\_priority];

wt[highest\_priority] = tat[highest\_priority] - bt[highest\_priority];

}

}

}

void print\_times(int n, int pr[], int at[], int bt[], int ft[], int tat[], int wt[]) {

printf("Process\tPriority\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting 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\t\t%d\n", i+1, pr[i], at[i], bt[i], ft[i], tat[i], wt[i]);

}

}

int main() {

int n, bt[20], at[20], pr[20], ft[20], tat[20], wt[20], choice;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &bt[i]);

}

printf("Enter arrival times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &at[i]);

}

printf("Enter priorities: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pr[i]);

}

printf("Choose Priority Scheduling Type:\n");

printf("1. Non-Preemptive Priority Scheduling\n");

printf("2. Preemptive Priority Scheduling\n");

printf("Enter your choice: ");

scanf("%d", &choice);

if (choice == 1) {

find\_priority\_np(n, bt, at, pr, ft, tat, wt);

} else if (choice == 2) {

find\_priority\_p(n, bt, at, pr, ft, tat, wt);

} else {

printf("Invalid choice!\n");

return 1;

}

print\_times(n, pr, at, bt, ft, tat, wt);

return 0;

}

/\*

4

3

1

7

4

2

1

0

3

2

1

3

4

2

\*/

/\*

This code implements \*\*Priority Scheduling\*\* in two forms:

1. \*\*Non-Preemptive Priority Scheduling\*\*: Once a process starts execution, it completes before another process can start, regardless of priority.

2. \*\*Preemptive Priority Scheduling\*\*: If a new process arrives with a higher priority, it can interrupt the currently running process.

Both scheduling methods are coded in separate functions and chosen by the user at runtime.

Let’s break down each part of the code:

---

### 1. `find\_priority\_np` - Non-Preemptive Priority Scheduling

This function simulates the non-preemptive priority scheduling.

```c

void find\_priority\_np(int n, int bt[], int at[], int pr[], int ft[], int tat[], int wt[])

```

- \*\*Parameters\*\*:

- `n`: Number of processes.

- `bt[]`: Array of burst times for each process.

- `at[]`: Array of arrival times for each process.

- `pr[]`: Array of priorities for each process.

- `ft[]`: Array to store finish times for each process.

- `tat[]`: Array to store turnaround times for each process.

- `wt[]`: Array to store waiting times for each process.

#### Variables

- `done[]`: Array to mark completed processes, initialized to 0.

- `time`: Keeps track of the current time.

- `min\_pr`: Tracks the lowest priority value found in the iteration.

- `highest`: Index of the highest-priority process that is ready for execution.

#### Scheduling Loop

```c

for (int i = 0; i < n; i++) {

min\_pr = INT\_MAX;

highest = -1;

```

- The loop runs `n` times (once per process) to find and execute the highest-priority, uncompleted process.

1. \*\*Finding the Highest-Priority Process\*\*:

```c

for (int j = 0; j < n; j++) {

if (at[j] <= time && !done[j] && pr[j] < min\_pr) {

min\_pr = pr[j];

highest = j;

}

}

```

- Iterates over all processes.

- Selects the process with the highest priority (`pr[j] < min\_pr`) that has arrived (`at[j] <= time`) and is not yet completed (`!done[j]`).

2. \*\*Process Execution\*\*:

```c

if (highest != -1) {

time += bt[highest];

ft[highest] = time;

tat[highest] = ft[highest] - at[highest];

wt[highest] = tat[highest] - bt[highest];

done[highest] = 1;

}

```

- If a process was found (`highest != -1`):

- The `time` advances by the process's burst time.

- Finish time (`ft[highest]`), turnaround time (`tat[highest]`), and waiting time (`wt[highest]`) are calculated for the selected process.

- Marks the process as completed.

3. \*\*No Process Available\*\*:

```c

else {

time++;

i--;

}

```

- If no process is available at the current time, `time` increments, and the loop index (`i`) is decremented to retry the current iteration.

---

### 2. `find\_priority\_p` - Preemptive Priority Scheduling

This function simulates preemptive priority scheduling, allowing higher-priority processes to interrupt lower-priority ones.

```c

void find\_priority\_p(int n, int bt[], int at[], int pr[], int ft[], int tat[], int wt[])

```

- This function has the same parameters as the non-preemptive version, except it also maintains an array for remaining burst times.

#### Variables

- `bt\_rem[]`: An array tracking remaining burst time for each process, initialized with `bt[]`.

- `time`: Tracks the current system time.

- `completed`: Counter for completed processes.

- `highest\_priority` and `min\_priority`: Track the process with the highest priority and the lowest priority value among the uncompleted processes.

#### Scheduling Loop

```c

while (completed < n) {

highest\_priority = -1;

min\_priority = INT\_MAX;

```

- The `while` loop continues until all processes have completed.

1. \*\*Finding the Process with Highest Priority\*\*:

```c

for (int i = 0; i < n; i++) {

if (at[i] <= time && bt\_rem[i] > 0 && pr[i] < min\_priority) {

min\_priority = pr[i];

highest\_priority = i;

}

}

```

- Searches for the process with the highest priority among processes that have arrived and have remaining burst time.

2. \*\*Executing the Highest Priority Process\*\*:

```c

if (highest\_priority == -1) {

time++;

continue;

}

bt\_rem[highest\_priority]--;

time++;

```

- If no process is found, the time increments.

- Otherwise, the highest-priority process is executed for one time unit, and its remaining burst time (`bt\_rem[highest\_priority]`) is reduced by 1.

3. \*\*Completion of a Process\*\*:

```c

if (bt\_rem[highest\_priority] == 0) {

completed++;

ft[highest\_priority] = time;

tat[highest\_priority] = ft[highest\_priority] - at[highest\_priority];

wt[highest\_priority] = tat[highest\_priority] - bt[highest\_priority];

}

```

- When a process's remaining burst time reaches zero, it is marked as completed.

- The finish time, turnaround time, and waiting time are calculated and stored.

---

### 3. `print\_times` - Display Function

```c

void print\_times(int n, int pr[], int at[], int bt[], int ft[], int tat[], int wt[])

```

- \*\*Parameters\*\*: Receives all the arrays storing process details.

- \*\*Purpose\*\*: Displays each process's priority, arrival time, burst time, finish time, turnaround time, and waiting time in a formatted table.

---

### 4. `main` Function

```c

int main() {

int n, bt[20], at[20], pr[20], ft[20], tat[20], wt[20], choice;

```

- Defines arrays for burst time (`bt[]`), arrival time (`at[]`), priority (`pr[]`), finish time (`ft[]`), turnaround time (`tat[]`), and waiting time (`wt[]`).

- `choice`: Stores the user’s choice of scheduling type.

#### User Input

```c

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &bt[i]);

}

printf("Enter arrival times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &at[i]);

}

printf("Enter priorities: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pr[i]);

}

```

- Prompts the user to enter the number of processes and their respective burst times, arrival times, and priorities.

#### Scheduling Type Selection

```c

printf("Choose Priority Scheduling Type:\n");

printf("1. Non-Preemptive Priority Scheduling\n");

printf("2. Preemptive Priority Scheduling\n");

printf("Enter your choice: ");

scanf("%d", &choice);

```

- Asks the user to select either non-preemptive or preemptive scheduling.

#### Function Call

```c

if (choice == 1) {

find\_priority\_np(n, bt, at, pr, ft, tat, wt);

} else if (choice == 2) {

find\_priority\_p(n, bt, at, pr, ft, tat, wt);

} else {

printf("Invalid choice!\n");

return 1;

}

print\_times(n, pr, at, bt, ft, tat, wt);

```

- Depending on the user’s choice, calls either `find\_priority\_np` (non-preemptive) or `find\_priority\_p` (preemptive) function.

- Finally, `print\_times` is called to display the results.

---

### Summary

- \*\*Non-Preemptive Priority Scheduling\*\* (`find\_priority\_np`): Executes each process with the highest priority that is ready, without interruption.

- \*\*Preemptive Priority Scheduling\*\* (`find\_priority\_p`): Allows interruptions if a new process arrives with a higher priority.

- The output includes each process’s priority, arrival time, burst time, finish time, turnaround time, and waiting time.

\*/

----------------------------------**12\_Round\_Robin.c**

//round-robin

#include <stdio.h>

void find\_rr(int n, int bt[], int at[], int ft[], int tat[], int wt[], int quantum) {

int bt\_rem[n], time = 0, count = 0;

for (int i = 0; i < n; i++) bt\_rem[i] = bt[i];

while (count < n) {

for (int i = 0; i < n; i++) {

if (at[i] <= time && bt\_rem[i] > 0) {

if (bt\_rem[i] > quantum) {

time += quantum;

bt\_rem[i] -= quantum;

} else {

time += bt\_rem[i];

bt\_rem[i] = 0;

ft[i] = time;

tat[i] = ft[i] - at[i];

wt[i] = tat[i] - bt[i];

count++;

}

}

}

}

}

void print\_times(int n, int at[], int bt[], int ft[], int tat[], int wt[]) {

printf("Process\tArrival Time\tBurst Time\tFinish Time\tTurnaround Time\tWaiting 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", i+1, at[i], bt[i], ft[i], tat[i], wt[i]);

}

}

int main() {

int n, bt[20], at[20], ft[20], tat[20], wt[20], quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter burst times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &bt[i]);

}

printf("Enter arrival times: ");

for (int i = 0; i < n; i++) {

scanf("%d", &at[i]);

}

printf("Enter time quantum for Round Robin: ");

scanf("%d", &quantum);

find\_rr(n, bt, at, ft, tat, wt, quantum);

print\_times(n, at, bt, ft, tat, wt);

return 0;

}

/\*

Enter number of processes: 3

Enter burst times: 10 5 8

Enter arrival times: 0 1 2

Enter time quantum for Round Robin: 3

\*/

/\*

This function performs the Round Robin scheduling calculations for each process. It takes in the following parameters:

n: The number of processes.

bt[]: An array of burst times for each process.

at[]: An array of arrival times for each process.

ft[]: An array to store the finish times for each process.

tat[]: An array to store the turnaround times for each process.

wt[]: An array to store the waiting times for each process.

quantum: The time quantum, or maximum time each process can execute in one cycle before being switched out.

bt\_rem[]: An array to track the remaining burst time for each process, initialized with each process's original burst time (bt[]).

time: Tracks the current time in the system, initialized to 0.

count: Counts the number of processes that have completed.

Main Scheduling Loop

The while loop runs until all processes have completed (count < n).

Inside, a for loop iterates over each process. For each process, it checks:

Arrival Time Condition (at[i] <= time): The process must have arrived by the current time.

Remaining Burst Time Condition (bt\_rem[i] > 0): The process should still have some burst time left.

Executing Each Process

If bt\_rem[i] > quantum: The process executes for the full quantum time.

time is increased by quantum, simulating the process's execution.

bt\_rem[i] is decreased by quantum, reducing the remaining burst time.

If bt\_rem[i] <= quantum: The process completes within the remaining time.

time is increased by the remaining burst time (bt\_rem[i]).

bt\_rem[i] is set to 0, indicating that the process has completed.

ft[i] is set to the current time, marking the process's finish time.

tat[i] is calculated as the difference between finish time and arrival time.

wt[i] is calculated as the difference between turnaround time and burst time.

count is incremented, marking the process as completed.

2. print\_times Function

This function displays the calculated times (arrival, burst, finish, turnaround, and waiting times) for each process. The parameters are:

n: Number of processes.

at[]: Array of arrival times.

bt[]: Array of burst times.

ft[]: Array of finish times.

tat[]: Array of turnaround times.

wt[]: Array of waiting times.

Function Execution

The function prints a table header.

Then, it iterates over each process and displays its times, formatted in a table.

3. main Function

The main function gathers user inputs, calls the find\_rr function to calculate scheduling times, and finally calls print\_times to display the results.

Input Process Details:

Prompts the user to enter the number of processes (n).

Input Burst and Arrival Times:

The user is prompted to input burst times and arrival times for each process. These values are stored in bt[] and at[].

Input Quantum:

The user specifies the time quantum, which is the maximum time each process can execute in one cycle.

Function Calls:

Calls find\_rr to perform the Round Robin scheduling calculations.

Calls print\_times to display the results for each process, including finish, turnaround, and waiting times.

Summary

Round Robin Scheduling: Each process is executed in a cyclic order for a maximum of quantum time. If a process's remaining burst time is greater than quantum, it goes back into the queue after quantum time and waits for its next turn. Otherwise, it completes.

Turnaround Time (TAT) is the total time taken from arrival to completion.

Waiting Time (WT) is the time a process spends waiting in the queue, calculated as TAT - Burst Time.

\*/

--------------------------**13\_Banker\_Deadlock\_Avoidance.c**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void print\_matrices(int processes, int resources, int max[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES], int need[][MAX\_RESOURCES]) {

printf("\nAllocation Matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

printf("%d ", allot[i][j]);

}

printf("\n");

}

printf("\nMax Matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

printf("%d ", max[i][j]);

}

printf("\n");

}

printf("\nNeed Matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

printf("%d ", need[i][j]);

}

printf("\n");

}

}

bool is\_safe(int processes, int resources, int avail[], int max[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES]) {

int work[MAX\_RESOURCES], finish[MAX\_PROCESSES] = {0}, safe\_seq[MAX\_PROCESSES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

need[i][j] = max[i][j] - allot[i][j];

}

}

print\_matrices(processes, resources, max, allot, need);

for (int i = 0; i < resources; i++) {

work[i] = avail[i];

}

int count = 0;

while (count < processes) {

bool found = false;

for (int p = 0; p < processes; p++) {

if (finish[p] == 0) {

int j;

for (j = 0; j < resources; j++) {

if (need[p][j] > work[j]) {

break;

}

}

if (j == resources) {

for (int k = 0; k < resources; k++) {

work[k] += allot[p][k];

}

finish[p] = 1;

safe\_seq[count++] = p;

found = true;

}

}

}

if (!found) {

printf("System is not in a safe state.\n");

return false;

}

}

printf("System is in a safe state.\nSafe sequence is: ");

for (int i = 0; i < processes; i++) {

printf("%d ", safe\_seq[i]);

}

printf("\n");

return true;

}

int main() {

int processes, resources;

printf("Enter the number of processes: ");

scanf("%d", &processes);

printf("Enter the number of resources: ");

scanf("%d", &resources);

int avail[MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int allot[MAX\_PROCESSES][MAX\_RESOURCES];

printf("Enter the available resources: \n");

for (int i = 0; i < resources; i++) {

scanf("%d", &avail[i]);

}

printf("Enter the maximum matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

scanf("%d", &max[i][j]);

}

}

printf("Enter the allocation matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

scanf("%d", &allot[i][j]);

}

}

is\_safe(processes, resources, avail, max, allot);

return 0;

}

// input

// 5

// 4

// 1 5 2 0

// 0 0 1 2

// 1 7 5 0

// 2 3 5 6

// 0 6 5 2

// 0 6 5 6

// 0 0 1 2

// 1 0 0 0

// 1 3 5 4

// 0 6 3 2

// 0 0 14

/\*

This program implements the Banker's Algorithm for deadlock avoidance. The algorithm checks if the system is in a "safe state" or not, based on the available resources and the current allocation of resources to various processes. Here's a step-by-step explanation of the code, breaking it down into easy-to-understand parts:

Key Concepts:

Processes: The individual tasks that need resources.

Resources: The limited resources that are needed by processes to complete.

Max Matrix: Represents the maximum resources each process may need.

Allocation Matrix: Represents the current resources allocated to each process.

Need Matrix: Represents the remaining resources each process needs to finish.

Code Explanation:

1. Input Section:

The program first asks the user for:

The number of processes (processes).

The number of resources (resources).

The available resources (avail[]).

The maximum resources required by each process (max[][]).

The current resource allocation for each process (allot[][]).

2. Print Matrices:

The function print\_matrices() is called to display:

The Allocation Matrix: The resources currently held by each process.

The Max Matrix: The maximum resources that each process may need.

The Need Matrix: The remaining resources needed by each process, calculated as Need = Max - Allocation.

This function helps the user visualize the system's current state, though it's mainly for debugging purposes here.

3. Safety Check (Banker's Algorithm):

The main logic to determine if the system is in a safe state is implemented in the function is\_safe(). Here's how it works:

Need Matrix Calculation: For each process, the Need matrix is calculated as the difference between the Max matrix and the Allocation matrix (i.e., Need[i][j] = Max[i][j] - Allot[i][j]).

Work Array: The work[] array represents the number of available resources at any given time. Initially, this is set to the available resources.

Finish Array: The finish[] array keeps track of whether a process has finished executing or not. Initially, all processes are unfinished.

Safe Sequence Check:

The algorithm tries to find a process that can be completed. A process can be completed if for all its remaining resource needs (from the Need matrix), the available resources (in the work array) are sufficient.

If a process can finish:

The available resources are increased by the resources currently held by that process (since the process has finished, it releases those resources).

The process is marked as finished (finish[i] = 1), and its index is added to the safe sequence.

If no process can be finished in a round (i.e., no process can meet its resource needs), it means the system is not in a safe state and deadlock might occur.

Output:

If all processes can eventually finish (i.e., if a safe sequence is found), the system is in a safe state and the safe sequence is printed.

If no safe sequence can be found, the system is in a deadlock state and the program prints "System is not in a safe state."

4. Main Function:

The program takes input from the user, stores it in the appropriate arrays (avail[], max[][], and allot[][]), and then calls is\_safe() to check if the system is in a safe state.

\*/

---------- Deadlock Detection

#include <stdio.h>

#include <stdbool.h>

void deadlockDetection(int available[], int max[][10], int allocation[][10], int processes, int resources) {

int work[resources];

bool finish[processes];

for (int i = 0; i < resources; i++)

work[i] = available[i];

for (int i = 0; i < processes; i++)

finish[i] = false;

bool done;

do {

done = false;

for (int i = 0; i < processes; i++) {

if (!finish[i]) {

bool canProceed = true;

for (int j = 0; j < resources; j++) {

if (max[i][j] - allocation[i][j] > work[j]) {

canProceed = false;

break;

}

}

if (canProceed) {

for (int k = 0; k < resources; k++)

work[k] += allocation[i][k];

finish[i] = true;

done = true;

}

}

}

} while (done);

bool deadlock = false;

printf("Processes in deadlock: ");

for (int i = 0; i < processes; i++) {

if (!finish[i]) {

printf("P%d ", i);

deadlock = true;

}

}

if (!deadlock)

printf("None");

printf("\n");

}

int main() {

int processes, resources;

printf("Enter the number of processes: ");

scanf("%d", &processes);

printf("Enter the number of resources: ");

scanf("%d", &resources);

int available[resources];

int max[processes][10];

int allocation[processes][10];

printf("Enter available resources:\n");

for (int i = 0; i < resources; i++) {

printf("Resource %d: ", i + 1);

scanf("%d", &available[i]);

}

printf("Enter maximum resource demand for each process:\n");

for (int i = 0; i < processes; i++) {

printf("Process P%d:\n", i);

for (int j = 0; j < resources; j++) {

printf("Resource %d: ", j + 1);

scanf("%d", &max[i][j]);

}

}

printf("Enter allocated resources for each process:\n");

for (int i = 0; i < processes; i++) {

printf("Process P%d:\n", i);

for (int j = 0; j < resources; j++) {

printf("Resource %d: ", j + 1);

scanf("%d", &allocation[i][j]);

}

}

deadlockDetection(available, max, allocation, processes, resources);

return 0;

}

/\*

5

4

1 5 2 0

0 0 1 2

1 7 5 0

2 3 5 6

0 6 5 2

0 6 5 6

0 0 1 2

1 0 0 0

1 3 5 4

0 6 3 2

0 0 1 4

\*/

/\*

The code implements \*\*deadlock detection\*\* in an operating system by using the \*\*Banker’s Algorithm\*\*. Here’s a detailed explanation of how the code works:

### Code Structure and Purpose

1. \*\*Headers and Definitions\*\*:

- `#include <stdio.h>` and `#include <stdbool.h>`: We include standard input/output and Boolean libraries for basic functionality.

- We define the `deadlockDetection` function to check if there are any processes in deadlock.

2. \*\*Function: `deadlockDetection`\*\*

- \*\*Inputs\*\*:

- `available[]`: An array representing the number of available instances of each resource type.

- `max[][]`: A matrix representing the maximum demand of each resource for each process.

- `allocation[][]`: A matrix showing the number of resources of each type currently allocated to each process.

- `processes` and `resources`: The number of processes and resource types, respectively.

- \*\*Variables\*\*:

- `work[]`: An array to represent the currently available resources that can be used by processes.

- `finish[]`: A Boolean array to track whether each process has completed or not.

- `done`: A flag to track whether any process was completed in the last pass through the processes.

- \*\*Initialization\*\*:

- Copies the `available` resources into `work[]`, which represents the resources available at any point during execution.

- Sets all elements in `finish[]` to `false`, indicating that no process has initially completed.

- \*\*Main Loop\*\*:

- The algorithm runs in a `do-while` loop, which will continue as long as any process is marked as completed (`done` is `true`).

- In each iteration, it checks if there is a process that:

- Has not yet finished (`finish[i] == false`).

- Has a demand (`max[i][j] - allocation[i][j]`) that can be satisfied with the currently available resources (`work[j]`).

- \*\*Process Selection\*\*:

- If a process can proceed (i.e., its demand is less than or equal to `work`), we add its allocated resources back to `work[]`, marking it as complete (`finish[i] = true`), and set `done = true`.

- If no process can proceed, it increments the `time` by 1 unit (used here to simulate waiting).

- \*\*Deadlock Detection\*\*:

- After the `do-while` loop, any process with `finish[i] == false` is in deadlock.

- The code then prints out any process that has not finished, indicating it is in a deadlock. If all processes are finished (`finish[i] == true` for all `i`), it prints "None," indicating no deadlock.

3. \*\*Function: `main`\*\*

- \*\*Inputs\*\*:

- Prompts the user to enter the number of processes and resources.

- Requests available resources and constructs the `available[]` array.

- Requests each process’s maximum resource demand and stores it in `max[][]`.

- Requests each process’s current allocated resources and stores them in `allocation[][]`.

- \*\*Calling Deadlock Detection\*\*:

- Calls the `deadlockDetection` function with the provided inputs.

- Prints out any processes detected to be in deadlock.

### Example Execution

Given the input provided, the program checks if any processes are in deadlock. If a process cannot acquire enough resources to proceed, the algorithm will eventually determine that it is stuck and label it as in deadlock.

### Key Points

1. \*\*Safety Check\*\*: This code is similar to the safety check in the Banker’s Algorithm, where it checks if processes can finish with the currently available resources.

2. \*\*Deadlock Detection\*\*: Instead of trying to prevent deadlock, the code only detects it. Processes that cannot get the resources they need (i.e., `finish[i] == false`) are considered deadlocked.

3. \*\*Efficient Resource Utilization\*\*: Processes that finish release their resources to `work[]`, allowing other processes to potentially complete.

This code is designed to handle resource allocation and check for deadlocks in a simple way, focusing on finding a solution in cases where multiple processes compete for limited resources.

\*/

--------- 15 Fifo Page Rep---------------

#include <stdio.h>

void fifo(int ref[], int n, int frames) {

int frame[frames], page\_faults = 0, index = 0;

for (int i = 0; i < frames; i++) frame[i] = -1;

printf("\nFIFO Page Replacement:\n");

for (int i = 0; i < n; i++) {

int found = 0;

for (int j = 0; j < frames; j++) {

if (frame[j] == ref[i]) {

found = 1;

break;

}

}

if (!found) {

frame[index] = ref[i];

index = (index + 1) % frames;

page\_faults++;

}

printf("Page %d: ", ref[i]);

for (int j = 0; j < frames; j++) {

if (frame[j] != -1) printf("%d ", frame[j]);

else printf("- ");

}

printf("\n");

}

printf("Total Page Faults (FIFO): %d\n", page\_faults);

}

int main() {

int n, frames;

printf("Enter number of pages in the reference string: ");

scanf("%d", &n);

int ref[n];

printf("Enter the reference string: ");

for (int i = 0; i < n; i++) {

scanf("%d", &ref[i]);

}

printf("Enter number of frames: ");

scanf("%d", &frames);

fifo(ref, n, frames);

return 0;

}

//input

/\*

12

1

2

3

4

1

2

5

1

2

3

4

5

4

Total 10

\*/

/\*

This code is an implementation of the \*\*FIFO (First-In-First-Out) Page Replacement Algorithm\*\* used in operating systems to manage memory. It keeps track of page faults in the process of loading pages into frames.

### Key Concepts of FIFO Page Replacement

- \*\*Pages\*\*: A reference string representing pages being accessed by a process.

- \*\*Frames\*\*: The number of available slots in memory to hold pages.

- \*\*Page Fault\*\*: Occurs when a page is not found in the frame, necessitating it to be loaded into memory.

### Code Explanation

1. \*\*Main Function\*\*

- The `main` function collects user inputs, such as:

- `n`: the number of pages in the reference string.

- `ref[]`: the reference string array that represents the sequence of page accesses.

- `frames`: the number of frames available in memory.

- Once the input is collected, it calls the `fifo` function to perform the FIFO page replacement.

2. \*\*FIFO Function (`fifo`)\*\*

- \*\*Parameters\*\*:

- `ref[]`: Reference string of pages.

- `n`: Total number of pages in the reference string.

- `frames`: Number of frames available in memory.

- \*\*Variables\*\*:

- `frame[]`: An array representing the memory frames (initialized to `-1`, indicating an empty frame).

- `page\_faults`: Counter for page faults (initialized to 0).

- `index`: Tracks which frame to replace (initialized to 0 for FIFO behavior).

3. \*\*FIFO Page Replacement Logic\*\*

- \*\*Initialize Frames\*\*: All frames are initially set to `-1` to indicate they are empty.

- \*\*Iterate Over Pages\*\*:

- For each page in the reference string:

- \*\*Page Check\*\*:

- Check if the page is already in one of the frames (`frame[j] == ref[i]`). If it is found, `found` is set to `1`, and no page fault occurs.

- \*\*Page Fault Handling\*\*:

- If the page is not found in any frame (`found == 0`), a page fault occurs:

- The new page (`ref[i]`) is inserted into the `frame[index]` position, replacing the oldest page in FIFO order.

- The `index` is incremented (modulo `frames`), ensuring it loops back to the start once all frames are filled.

- The `page\_faults` counter is incremented to track page faults.

- \*\*Display Frame Status\*\*:

- After each page reference, the program prints the current status of each frame, showing which pages are loaded in each frame slot.

- \*\*Output Total Page Faults\*\*:

- After processing the entire reference string, the function prints the total number of page faults.

4. \*\*Example Output\*\*

- For a reference string like `{0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4}` and 3 frames, the program outputs which pages are in each frame after each page reference, and finally the total number of page faults.

### Code Execution Flow Example

Suppose the reference string is `{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}` and we have 3 frames.

| Page | Frame Contents | Page Fault |

|------|------------------------|------------|

| 1 | 1 - - | Yes |

| 2 | 1 2 - | Yes |

| 3 | 1 2 3 | Yes |

| 4 | 4 2 3 | Yes |

| 1 | 4 1 3 | Yes |

| 2 | 4 1 2 | Yes |

| 5 | 5 1 2 | Yes |

| 1 | 5 1 2 | No |

| 2 | 5 1 2 | No |

| 3 | 3 1 2 | Yes |

| 4 | 3 4 2 | Yes |

| 5 | 3 4 5 | Yes |

- \*\*Total Page Faults\*\*: 9

### Summary

This code demonstrates FIFO page replacement, where each new page that causes a page fault replaces the oldest page in memory. This results in a total count of page faults based on the reference string and available frames.

\*/

--------- 16 LRU Page Replacement -----------

//lru

#include <stdio.h>

void lru(int ref[], int n, int frames) {

int frame[frames], page\_faults = 0, time[frames], least, found;

for (int i = 0; i < frames; i++) frame[i] = -1;

for (int i = 0; i < frames; i++) time[i] = -1;

printf("\nLRU Page Replacement:\n");

for (int i = 0; i < n; i++) {

found = 0;

for (int j = 0; j < frames; j++) {

if (frame[j] == ref[i]) {

found = 1;

time[j] = i;

break;

}

}

if (!found) {

least = 0;

for (int j = 1; j < frames; j++) {

if (time[j] < time[least]) least = j;

}

frame[least] = ref[i];

time[least] = i;

page\_faults++;

}

printf("Page %d: ", ref[i]);

for (int j = 0; j < frames; j++) {

if (frame[j] != -1) printf("%d ", frame[j]);

else printf("- ");

}

printf("\n");

}

printf("Total Page Faults (LRU): %d\n", page\_faults);

}

int main() {

int n, frames;

printf("Enter number of pages in the reference string: ");

scanf("%d", &n);

int ref[n];

printf("Enter the reference string: ");

for (int i = 0; i < n; i++) {

scanf("%d", &ref[i]);

}

printf("Enter number of frames: ");

scanf("%d", &frames);

lru(ref, n, frames);

return 0;

}

/\*

12

1

2

3

4

1

2

5

1

2

3

4

5

4

\*/

/\*

This code is an implementation of the \*\*LRU (Least Recently Used) Page Replacement Algorithm\*\*, which is used in operating systems to manage memory. This algorithm loads pages into frames while keeping track of page faults and replaces the least recently used page in the frame when necessary.

### Key Concepts of LRU Page Replacement

- \*\*Pages\*\*: A reference string representing pages being accessed by a process.

- \*\*Frames\*\*: The number of slots available in memory to hold pages.

- \*\*Page Fault\*\*: Occurs when a page is not found in any of the frames, requiring it to be loaded into memory.

- \*\*Least Recently Used\*\*: The page that was accessed the farthest back in time is replaced when a new page is loaded into a full set of frames.

### Code Explanation

1. \*\*Main Function\*\*

- Collects user inputs:

- `n`: the number of pages in the reference string.

- `ref[]`: the reference string array representing the sequence of page accesses.

- `frames`: the number of frames available in memory.

- Once the input is collected, it calls the `lru` function to perform the LRU page replacement.

2. \*\*LRU Function (`lru`)\*\*

- \*\*Parameters\*\*:

- `ref[]`: The reference string of pages.

- `n`: The total number of pages in the reference string.

- `frames`: The number of frames available in memory.

- \*\*Variables\*\*:

- `frame[]`: An array representing the memory frames, initialized to `-1` (indicating empty frames).

- `page\_faults`: Counts the number of page faults (initialized to 0).

- `time[]`: Keeps track of the time each page was last used in the frames.

- `least`: Stores the index of the least recently used frame.

3. \*\*LRU Page Replacement Logic\*\*

- \*\*Initialize Frames\*\*: All frames are initially set to `-1` to indicate they are empty, and `time[]` is initialized to `-1` to signify that no pages have been loaded.

- \*\*Iterate Over Pages\*\*:

- For each page in the reference string:

- \*\*Page Check\*\*:

- The algorithm checks if the page is already present in any frame (`frame[j] == ref[i]`). If found, `found` is set to 1, and the current index `i` is recorded in `time[j]` (indicating the last access time for that page).

- \*\*Page Fault Handling\*\*:

- If the page is not found in any frame (`found == 0`), a page fault occurs:

- The least recently used frame is found by checking which frame has the smallest `time` value.

- The new page (`ref[i]`) is inserted into `frame[least]`, replacing the least recently used page.

- The current time `i` is updated in `time[least]` to record this page's last access time.

- The `page\_faults` counter is incremented.

- \*\*Display Frame Status\*\*:

- After each page reference, the program prints the current status of each frame, showing which pages are loaded in each frame slot.

- \*\*Output Total Page Faults\*\*:

- After processing the entire reference string, the function prints the total number of page faults.

4. \*\*Example Output\*\*

- For a reference string like `{0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4}` and 3 frames, the program outputs the page loaded in each frame after each reference and finally prints the total number of page faults.

### Code Execution Flow Example

Suppose the reference string is `{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}` and we have 3 frames.

| Page | Frame Contents | Page Fault |

|------|------------------------|------------|

| 1 | 1 - - | Yes |

| 2 | 1 2 - | Yes |

| 3 | 1 2 3 | Yes |

| 4 | 4 2 3 | Yes |

| 1 | 4 1 3 | Yes |

| 2 | 4 1 2 | Yes |

| 5 | 5 1 2 | Yes |

| 1 | 5 1 2 | No |

| 2 | 5 1 2 | No |

| 3 | 3 1 2 | Yes |

| 4 | 3 4 2 | Yes |

| 5 | 3 4 5 | Yes |

- \*\*Total Page Faults\*\*: 9

### Summary

This code implements the LRU page replacement algorithm, which loads pages into frames and replaces the page that has been unused the longest whenever a page fault occurs and all frames are full. It calculates the total number of page faults based on the reference string and available frames, providing insight into the efficiency of the LRU algorithm for memory management.

\*/

---------- 17 Optimal page Rep --------------------

//optimal

#include <stdio.h>

int find\_optimal(int ref[], int frame[], int n, int current, int frames) {

int pos = -1, farthest = current;

for (int i = 0; i < frames; i++) {

int j;

for (j = current; j < n; j++) {

if (frame[i] == ref[j]) {

if (j > farthest) {

farthest = j;

pos = i;

}

break;

}

}

if (j == n) return i;

}

if (pos == -1) return 0;

return pos;

}

void optimal(int ref[], int n, int frames) {

int frame[frames], page\_faults = 0, found;

for (int i = 0; i < frames; i++) frame[i] = -1;

printf("\nOptimal Page Replacement:\n");

for (int i = 0; i < n; i++) {

found = 0;

for (int j = 0; j < frames; j++) {

if (frame[j] == ref[i]) {

found = 1;

break;

}

}

if (!found) {

if (i < frames) frame[i] = ref[i];

else {

int pos = find\_optimal(ref, frame, n, i + 1, frames);

frame[pos] = ref[i];

}

page\_faults++;

}

printf("Page %d: ", ref[i]);

for (int j = 0; j < frames; j++) {

if (frame[j] != -1) printf("%d ", frame[j]);

else printf("- ");

}

printf("\n");

}

printf("Total Page Faults (Optimal): %d\n", page\_faults);

}

int main() {

int n, frames;

printf("Enter number of pages in the reference string: ");

scanf("%d", &n);

int ref[n];

printf("Enter the reference string: ");

for (int i = 0; i < n; i++) {

scanf("%d", &ref[i]);

}

printf("Enter number of frames: ");

scanf("%d", &frames);

optimal(ref, n, frames);

return 0;

}

/\*

12

1

2

3

4

1

2

5

1

2

3

4

5

4

\*/

/\*

This code implements the \*\*Optimal Page Replacement Algorithm\*\*, which is a page replacement policy used to manage memory in operating systems. The key idea behind the Optimal algorithm is to replace the page that will not be used for the longest period in the future, thereby minimizing page faults.

### Key Concepts of the Optimal Page Replacement Algorithm:

- \*\*Pages\*\*: A sequence of page references made by a process.

- \*\*Frames\*\*: The available slots in memory to hold pages.

- \*\*Page Fault\*\*: Occurs when a page is not present in memory and needs to be loaded.

- \*\*Optimal Replacement\*\*: At each step, the page that will not be needed for the longest time in the future is replaced.

### Code Explanation

#### 1. \*\*Main Function\*\*:

- \*\*User Input\*\*:

- `n`: The number of pages in the reference string (how many page accesses will be simulated).

- `ref[]`: The reference string array representing the sequence of page accesses.

- `frames`: The number of frames available in memory.

- After collecting the input, the `optimal` function is called to simulate the Optimal Page Replacement.

#### 2. \*\*Optimal Function\*\* (`optimal`):

- \*\*Parameters\*\*:

- `ref[]`: The reference string of pages.

- `n`: The total number of pages in the reference string.

- `frames`: The number of frames in memory to hold pages.

- \*\*Variables\*\*:

- `frame[]`: An array representing the frames in memory, initialized to `-1` to indicate empty slots.

- `page\_faults`: Counter for the total number of page faults (initialized to 0).

- `found`: A flag used to check whether a page is already present in the frames or not.

#### 3. \*\*Optimal Page Replacement Logic\*\*:

- \*\*Initialize Frames\*\*: Initially, all frames are set to `-1` (empty).

- \*\*Iterate Through Reference String\*\*:

- For each page in the reference string:

- \*\*Page Check\*\*: The algorithm checks if the page is already present in any of the frames (`frame[j] == ref[i]`). If found, it means no page fault occurs, and the page is already in memory.

- \*\*Page Fault Handling\*\*:

- If the page is not found in any frame (`found == 0`), a page fault occurs:

- If there is an empty frame (i.e., not all frames are filled yet), the page is inserted into the first available frame.

- If all frames are full, the \*\*Optimal Replacement Decision\*\* is made by calling the `find\_optimal` function.

- \*\*Optimal Page Selection\*\*:

- \*\*`find\_optimal()`\*\* is a helper function that determines which page should be replaced:

- It looks at the pages currently in memory (`frame[]`) and checks when each page will be accessed again in the future. The page that is needed farthest in the future is replaced.

- If a page is not found in the future references (i.e., it won't be used again), that page is chosen for replacement.

- \*\*Display Frame Status\*\*: After each reference, the contents of the frames are printed, showing the pages currently in memory. If a frame is empty, it prints a `-` symbol to represent the empty slot.

- \*\*Output Total Page Faults\*\*: After processing all the pages in the reference string, the program prints the total number of page faults.

#### 4. \*\*`find\_optimal()` Function\*\*:

- \*\*Parameters\*\*:

- `ref[]`: The reference string of pages.

- `frame[]`: The current state of the frames.

- `n`: The total number of pages.

- `current`: The current index of the page being processed.

- `frames`: The number of available frames.

- \*\*Purpose\*\*: This function determines which page to replace based on the optimal algorithm (the page that is used farthest in the future).

- \*\*Logic\*\*:

- For each page in memory (`frame[i]`), the function looks ahead in the reference string and tries to find the page that will be accessed last (farthest from the current page).

- If a page will not be used in the future, it's selected for replacement.

- If all pages in the frames are needed in the future, it selects the page with the farthest next usage.

#### 5. \*\*Example Execution\*\*:

For example, with the reference string `{7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3}` and 3 frames:

| Step | Reference | Frame Status | Page Fault? |

|------|-----------|---------------------|-------------|

| 1 | 7 | 7 - - | Yes |

| 2 | 0 | 7 0 - | Yes |

| 3 | 1 | 7 0 1 | Yes |

| 4 | 2 | 2 0 1 | Yes |

| 5 | 0 | 2 0 1 | No |

| 6 | 3 | 2 3 1 | Yes |

| 7 | 0 | 0 3 1 | Yes |

| 8 | 4 | 0 3 4 | Yes |

| 9 | 2 | 0 2 4 | Yes |

| 10 | 3 | 0 2 3 | Yes |

| 11 | 0 | 0 2 3 | No |

| 12 | 3 | 0 2 3 | No |

- \*\*Total Page Faults\*\*: 9

#### 6. \*\*Summary\*\*:

The Optimal Page Replacement algorithm minimizes the number of page faults by replacing the page that will not be used for the longest time in the future. This code simulates the page replacement process for a given reference string and frame size, showing the state of memory after each page access and calculating the total number of page faults. This approach provides a theoretical benchmark for evaluating other page replacement algorithms like FIFO or LRU.

\*/

------------FCFS Disk Scheuling --------------  
#include <stdio.h>

#include <stdlib.h>

void fcfs\_disk\_scheduling(int requests[], int n, int start) {

int total\_seek\_time = 0;

int current\_position = start;

printf("\nFCFS Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

for (int i = 0; i < n; i++) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, start\_position;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position);

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

fcfs\_disk\_scheduling(requests, n, start\_position);

return 0;

}

// input

/\*

100

9

55

58

39

18

90

160

150

38

184

\*/

/\*

Here’s an explanation of the FCFS (First-Come, First-Served) disk scheduling program:

### \*\*Overview:\*\*

The program simulates the FCFS disk scheduling algorithm, where the disk head services disk requests in the order they arrive. The primary goal of the algorithm is to compute the total seek time while traversing the disk request queue.

### \*\*Code Explanation:\*\*

#### \*\*1. `fcfs\_disk\_scheduling()` function:\*\*

This function simulates the FCFS disk scheduling algorithm and calculates the total seek time as the disk head moves through the requests.

```c

void fcfs\_disk\_scheduling(int requests[], int n, int start) {

int total\_seek\_time = 0;

int current\_position = start;

printf("\nFCFS Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

for (int i = 0; i < n; i++) {

int seek\_time = abs(current\_position - requests[i]); // Calculate seek time

total\_seek\_time += seek\_time; // Add seek time to the total

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time); // Print details

current\_position = requests[i]; // Update current position to the requested track

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time); // Print total seek time

}

```

- \*\*Input Parameters:\*\*

- `requests[]`: An array holding the disk requests (the tracks the disk head has to visit).

- `n`: The total number of disk requests.

- `start`: The starting position of the disk head.

- \*\*Process:\*\*

- The variable `total\_seek\_time` tracks the sum of all the seek times.

- The `current\_position` variable holds the current position of the disk head, which starts at `start`.

- For each request:

- The program calculates the \*\*seek time\*\* (the time to move from the current position to the requested track) using the formula:

```

seek\_time = abs(current\_position - requests[i])

```

- The seek time is added to `total\_seek\_time`.

- The current position is updated to the requested track (`current\_position = requests[i]`).

- The details of the current request (current position, requested track, and seek time) are printed in a tabular format.

- After iterating through all requests, the program prints the \*\*total seek time\*\*, which is the sum of all the individual seek times.

#### \*\*2. `main()` function:\*\*

The `main()` function is where the user inputs the data and where the `fcfs\_disk\_scheduling()` function is called.

```c

int main() {

int n, start\_position;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position); // Input the starting position of the disk head

printf("Enter the number of disk requests: ");

scanf("%d", &n); // Input the number of disk requests

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]); // Input each disk request

}

fcfs\_disk\_scheduling(requests, n, start\_position); // Call FCFS disk scheduling function

return 0;

}

```

- \*\*Steps:\*\*

1. \*\*Input the starting position of the disk head:\*\*

- The program prompts the user to enter the initial position of the disk head (`start\_position`).

2. \*\*Input the number of disk requests:\*\*

- The program then asks for the number of disk requests (`n`).

3. \*\*Input the disk requests:\*\*

- The user inputs each disk request. These are the tracks the disk head must visit in the order they arrive.

4. \*\*Call `fcfs\_disk\_scheduling()`:\*\*

- The `fcfs\_disk\_scheduling()` function is called with the inputs: the `requests` array, the number of requests `n`, and the starting position of the disk head.

5. \*\*Output:\*\*

- After processing the requests, the program prints the total seek time, and the seek time for each individual request is displayed in a tabular format.

### \*\*Key Concepts:\*\*

1. \*\*Seek Time Calculation:\*\*

- The seek time is the absolute difference between the current position of the disk head and the requested track. This is calculated using the `abs()` function.

2. \*\*FCFS Scheduling:\*\*

- The disk head services the requests in the order they arrive (first come, first served).

3. \*\*Seek Time Summation:\*\*

- The total seek time is the sum of all individual seek times as the disk head moves from one request to the next.

### \*\*Example:\*\*

For input:

```

Enter the starting position of the disk head: 50

Enter the number of disk requests: 5

Enter the disk requests:

82 170 43 140 24

```

The program would output:

```

FCFS Disk Scheduling

Current Position Requested Track Seek Time

50 82 32

82 170 88

170 43 127

43 140 97

140 24 116

Total Seek Time: 460

```

### \*\*Conclusion:\*\*

This program provides a clear simulation of the FCFS disk scheduling algorithm. It calculates the total seek time for all disk requests and prints the seek times in a tabular format. It helps visualize how the disk head moves through the request queue and provides a basic implementation of one of the simplest disk scheduling algorithms.

\*/  
  
----------

---------19 SSTF

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sstf\_disk\_scheduling(int requests[], int n, int start) {

int total\_seek\_time = 0;

int current\_position = start;

bool serviced[n]; // Array to mark requests that have been serviced

for (int i = 0; i < n; i++) {

serviced[i] = false; // Initialize all requests as unserviced

}

printf("\nSSTF Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

for (int i = 0; i < n; i++) {

int min\_seek\_time = \_\_INT\_MAX\_\_;

int closest\_request = -1;

// Find the closest unserviced request

for (int j = 0; j < n; j++) {

if (!serviced[j]) {

int seek\_time = abs(current\_position - requests[j]);

if (seek\_time < min\_seek\_time) {

min\_seek\_time = seek\_time;

closest\_request = j;

}

}

}

// Service the closest request

if (closest\_request != -1) {

serviced[closest\_request] = true;

total\_seek\_time += min\_seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[closest\_request], min\_seek\_time);

current\_position = requests[closest\_request];

}

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, start\_position;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position);

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

sstf\_disk\_scheduling(requests, n, start\_position);

return 0;

}

// input

/\*

50

5

82

170

43

140

24

\*/

/\*

The given code simulates the \*\*SSTF (Shortest Seek Time First)\*\* disk scheduling algorithm. This algorithm services the disk requests by selecting the request that is closest to the current position of the disk head, aiming to minimize the seek time (the distance the disk head has to move).

Here’s a detailed explanation of the code:

### \*\*Overview:\*\*

1. \*\*Goal of SSTF:\*\* The SSTF disk scheduling algorithm services the requests in such a way that the disk head moves the shortest distance for each request. At each step, it chooses the unserviced request that has the smallest seek time from the current position of the disk head.

2. \*\*Program Functionality:\*\*

- The user inputs the starting position of the disk head and the list of disk requests (the track numbers).

- The program calculates the total seek time and services each request one by one, always picking the closest unserviced request.

### \*\*Code Breakdown:\*\*

#### \*\*1. `sstf\_disk\_scheduling()` function:\*\*

This function implements the SSTF disk scheduling algorithm.

```c

void sstf\_disk\_scheduling(int requests[], int n, int start) {

int total\_seek\_time = 0;

int current\_position = start;

bool serviced[n]; // Array to mark requests that have been serviced

for (int i = 0; i < n; i++) {

serviced[i] = false; // Initialize all requests as unserviced

}

printf("\nSSTF Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

for (int i = 0; i < n; i++) {

int min\_seek\_time = \_\_INT\_MAX\_\_; // Initialize the minimum seek time to maximum integer value

int closest\_request = -1;

// Find the closest unserviced request

for (int j = 0; j < n; j++) {

if (!serviced[j]) { // If the request is not yet serviced

int seek\_time = abs(current\_position - requests[j]); // Calculate the seek time to this request

if (seek\_time < min\_seek\_time) { // If this request is closer than the previous closest request

min\_seek\_time = seek\_time;

closest\_request = j; // Mark this request as the closest one

}

}

}

// Service the closest request

if (closest\_request != -1) {

serviced[closest\_request] = true; // Mark the request as serviced

total\_seek\_time += min\_seek\_time; // Add the seek time to the total seek time

printf("%-20d%-20d%-20d\n", current\_position, requests[closest\_request], min\_seek\_time); // Print the details

current\_position = requests[closest\_request]; // Update the current position of the disk head

}

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time); // Print the total seek time

}

```

\*\*Step-by-Step Explanation:\*\*

- \*\*Input Parameters:\*\*

- `requests[]`: An array containing the disk requests (track numbers).

- `n`: The number of requests.

- `start`: The starting position of the disk head.

- \*\*Initialize `serviced[]` Array:\*\*

- This boolean array tracks whether a request has been serviced. Initially, all values are set to `false`.

- \*\*Loop Through Requests:\*\*

- The outer loop runs `n` times, once for each request, and at each iteration:

- \*\*Finding the Closest Request:\*\*

- For each request, the program checks all unserviced requests to find the one with the minimum seek time (the closest to the current disk head position).

- This is done by calculating the absolute difference between the current disk head position and each unserviced request using `abs(current\_position - requests[j])`.

- If a request is closer than the previously found one, it becomes the new "closest" request.

- \*\*Service the Closest Request:\*\*

- Once the closest request is identified, it's serviced (i.e., marked as serviced by setting `serviced[closest\_request] = true`).

- The total seek time is updated by adding the seek time for the serviced request.

- The current position of the disk head is updated to the position of the serviced request.

- \*\*Print Request Details:\*\*

- After servicing each request, the program prints the current position of the disk head, the track that was requested, and the seek time for that request in a tabular format.

- \*\*Total Seek Time:\*\*

- After servicing all requests, the program prints the total seek time, which is the sum of all individual seek times.

#### \*\*2. `main()` function:\*\*

This function handles user input and invokes the `sstf\_disk\_scheduling()` function to process the disk requests.

```c

int main() {

int n, start\_position;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position); // Input the starting position of the disk head

printf("Enter the number of disk requests: ");

scanf("%d", &n); // Input the number of disk requests

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]); // Input each disk request

}

sstf\_disk\_scheduling(requests, n, start\_position); // Call SSTF disk scheduling function

return 0;

}

```

\*\*Step-by-Step Explanation:\*\*

- \*\*Input the starting position of the disk head:\*\*

- The user is prompted to enter the initial position of the disk head (`start\_position`).

- \*\*Input the number of disk requests:\*\*

- The user is then asked how many disk requests there are (`n`).

- \*\*Input the disk requests:\*\*

- The program prompts the user to enter the disk requests. These are the track numbers that the disk head needs to visit.

- \*\*Call `sstf\_disk\_scheduling()` function:\*\*

- The `sstf\_disk\_scheduling()` function is called with the `requests[]` array, the number of requests (`n`), and the starting position of the disk head.

- \*\*Output:\*\*

- After processing the requests, the program will output the seek time for each request and the total seek time.

### \*\*Example:\*\*

For input:

```

Enter the starting position of the disk head: 50

Enter the number of disk requests: 5

Enter the disk requests:

82 170 43 140 24

```

The program would output:

```

SSTF Disk Scheduling

Current Position Requested Track Seek Time

50 43 7

43 24 19

24 82 58

82 140 58

140 170 30

Total Seek Time: 172

```

### \*\*Key Concepts:\*\*

1. \*\*Seek Time Calculation:\*\*

- The seek time is the absolute difference between the current position of the disk head and the requested track.

2. \*\*SSTF Scheduling:\*\*

- The SSTF algorithm always selects the request that is closest to the current disk head position, minimizing the seek time at each step.

3. \*\*Efficient Request Processing:\*\*

- The algorithm prioritizes requests that minimize disk head movement, but it can lead to starvation if some requests are always farther away from the disk head.

### \*\*Conclusion:\*\*

This program provides a simulation of the SSTF disk scheduling algorithm, efficiently calculating the total seek time and printing the details of each request serviced. It demonstrates how the disk head services requests in the shortest seek time order.

\*/

------------------- 20 Scan

#include <stdio.h>

#include <stdlib.h>

void scan\_disk\_scheduling(int requests[], int n, int start, int disk\_size) {

int total\_seek\_time = 0;

int current\_position = start;

int direction = 1; // 1 means move right, -1 means move left

// Sort the requests array to process them in order

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (requests[j] > requests[j + 1]) {

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// Find the first request greater than or equal to the current position

int index = 0;

while (index < n && requests[index] < current\_position) {

index++;

}

printf("\nSCAN Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

// Move right first (if direction = 1)

if (direction == 1) {

// Process requests from the current position to the highest track (rightward)

for (int i = index; i < n; i++) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

// Move to the disk's maximum position (disk size)

int seek\_time = abs(current\_position - disk\_size);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, disk\_size, seek\_time);

current\_position = disk\_size;

// Process remaining requests from the maximum position (leftward)

for (int i = index - 1; i >= 0; i--) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, start\_position, disk\_size;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position);

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter the disk size: ");

scanf("%d", &disk\_size);

scan\_disk\_scheduling(requests, n, start\_position, disk\_size);

return 0;

}

// input

/\*

50

8

176

79

34

60

92

11

41

114

200

\*/

/\*

The provided code simulates the \*\*SCAN (Elevator)\*\* disk scheduling algorithm. This algorithm moves the disk head in one direction (either left or right) to process the requests in that direction, and then reverses the direction once it reaches the end of the disk.

### \*\*SCAN Disk Scheduling Algorithm:\*\*

- The SCAN algorithm works by moving the disk head towards one end of the disk, servicing all requests in its path, then reversing direction and servicing requests in the opposite direction.

- It’s called "Elevator" because it works like an elevator that moves in one direction, servicing all requests, and then moves in the opposite direction.

### \*\*Code Breakdown:\*\*

#### \*\*1. `scan\_disk\_scheduling()` function:\*\*

This function implements the SCAN disk scheduling algorithm.

```c

void scan\_disk\_scheduling(int requests[], int n, int start, int disk\_size) {

int total\_seek\_time = 0;

int current\_position = start;

int direction = 1; // 1 means move right, -1 means move left

```

- \*\*`total\_seek\_time`:\*\* Keeps track of the total seek time as the disk head moves.

- \*\*`current\_position`:\*\* Tracks the current position of the disk head.

- \*\*`direction`:\*\* A variable to control the direction of the disk head movement (right = 1, left = -1). Initially, the direction is set to move right (1).

```c

// Sort the requests array to process them in order

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (requests[j] > requests[j + 1]) {

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

```

- The `requests[]` array is sorted in ascending order using the \*\*Bubble Sort\*\* algorithm. This ensures that requests are processed in the correct order (from lowest to highest).

```c

// Find the first request greater than or equal to the current position

int index = 0;

while (index < n && requests[index] < current\_position) {

index++;

}

```

- The code identifies the first request greater than or equal to the current position of the disk head. This will help in determining where to start processing requests.

```c

printf("\nSCAN Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

```

- Prints the header of the output in a tabular format: showing the current position, requested track, and seek time.

#### \*\*2. Processing Requests:\*\*

After sorting the requests, the function moves the disk head in the \*\*right\*\* direction first (towards the highest track):

```c

// Move right first (if direction = 1)

if (direction == 1) {

// Process requests from the current position to the highest track (rightward)

for (int i = index; i < n; i++) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

// Move to the disk's maximum position (disk size)

int seek\_time = abs(current\_position - disk\_size);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, disk\_size, seek\_time);

current\_position = disk\_size;

```

- \*\*Moving Rightward:\*\*

- Starting from the request closest to the current position (tracked by `index`), the function services all requests in the rightward direction (i.e., towards the maximum disk size).

- After processing all the rightward requests, the disk head moves to the farthest position on the disk (i.e., `disk\_size`), and the seek time to reach that position is calculated and added.

```c

// Process remaining requests from the maximum position (leftward)

for (int i = index - 1; i >= 0; i--) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

}

```

- \*\*Moving Leftward:\*\*

- After reaching the maximum position (`disk\_size`), the disk head reverses direction (leftward) and processes the remaining requests in reverse order (from highest to lowest).

- The seek time for each request is computed, and the `total\_seek\_time` is updated accordingly.

```c

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

```

- After processing all the requests, the total seek time is displayed.

#### \*\*3. `main()` function:\*\*

The `main()` function handles the user inputs and calls the `scan\_disk\_scheduling()` function to simulate the disk scheduling.

```c

int main() {

int n, start\_position, disk\_size;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position);

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter the disk size: ");

scanf("%d", &disk\_size);

scan\_disk\_scheduling(requests, n, start\_position, disk\_size);

return 0;

}

```

\*\*Step-by-Step Explanation:\*\*

- \*\*Input:\*\*

- The user is prompted to input the starting position of the disk head (`start\_position`), the number of disk requests (`n`), the disk size (`disk\_size`), and the actual disk requests (`requests[]`).

- \*\*Call `scan\_disk\_scheduling()`:\*\*

- Once the inputs are received, the program calls the `scan\_disk\_scheduling()` function to simulate the SCAN disk scheduling algorithm.

### \*\*Example Input and Output:\*\*

#### \*\*Input:\*\*

```

Enter the starting position of the disk head: 50

Enter the number of disk requests: 6

Enter the disk requests:

82 170 43 140 24 60

Enter the disk size: 200

```

#### \*\*Output:\*\*

```

SCAN Disk Scheduling

Current Position Requested Track Seek Time

50 60 10

60 82 22

82 140 58

140 170 30

170 200 30

200 24 176

24 43 19

43 50 7

Total Seek Time: 352

```

### \*\*Explanation of Output:\*\*

1. \*\*Sorting the Requests:\*\* The requests are sorted in ascending order: `24, 43, 50, 60, 82, 140, 170`.

2. \*\*Service Requests Moving Right:\*\* Starting from `50`, the requests are processed in the right direction (`50 → 60 → 82 → 140 → 170 → 200`).

3. \*\*Service Requests Moving Left:\*\* Once the head reaches `200`, it reverses direction and processes requests in the leftward direction (`200 → 24 → 43 → 50`).

4. \*\*Seek Time Calculation:\*\* The seek time for each request is computed, and the total seek time is displayed.

### \*\*Key Concepts:\*\*

- \*\*SCAN Disk Scheduling:\*\* The disk head moves in one direction, services all requests in that direction, and then reverses direction when it reaches the end of the disk.

- \*\*Seek Time Calculation:\*\* The seek time is the absolute distance between the current position and the requested track.

- \*\*Sorting Requests:\*\* Sorting the requests helps in efficiently processing them in one direction (either left or right).

### \*\*Conclusion:\*\*

This program simulates the SCAN disk scheduling algorithm, providing a step-by-step simulation of how the disk head processes disk requests. The total seek time is calculated and displayed at the end, helping to evaluate the performance of the SCAN scheduling approach.

\*/

-------------20 CSCAN

#include <stdio.h>

#include <stdlib.h>

void scan\_disk\_scheduling(int requests[], int n, int start, int disk\_size) {

int total\_seek\_time = 0;

int current\_position = start;

int direction = 1; // 1 means move right, -1 means move left

// Sort the requests array to process them in order

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (requests[j] > requests[j + 1]) {

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// Find the first request greater than or equal to the current position

int index = 0;

while (index < n && requests[index] < current\_position) {

index++;

}

printf("\nSCAN Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

// Move right first (if direction = 1)

if (direction == 1) {

// Process requests from the current position to the highest track (rightward)

for (int i = index; i < n; i++) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

// Move to the disk's maximum position (disk size)

int seek\_time = abs(current\_position - disk\_size);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, disk\_size, seek\_time);

current\_position = disk\_size;

// Process remaining requests from the maximum position (leftward)

for (int i = index - 1; i >= 0; i--) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, start\_position, disk\_size;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position);

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter the disk size: ");

scanf("%d", &disk\_size);

scan\_disk\_scheduling(requests, n, start\_position, disk\_size);

return 0;

}

// input

/\*

50

8

176

79

34

60

92

11

41

114

200

\*/

/\*

The provided code simulates the \*\*SCAN (Elevator)\*\* disk scheduling algorithm. This algorithm moves the disk head in one direction (either left or right) to process the requests in that direction, and then reverses the direction once it reaches the end of the disk.

### \*\*SCAN Disk Scheduling Algorithm:\*\*

- The SCAN algorithm works by moving the disk head towards one end of the disk, servicing all requests in its path, then reversing direction and servicing requests in the opposite direction.

- It’s called "Elevator" because it works like an elevator that moves in one direction, servicing all requests, and then moves in the opposite direction.

### \*\*Code Breakdown:\*\*

#### \*\*1. `scan\_disk\_scheduling()` function:\*\*

This function implements the SCAN disk scheduling algorithm.

```c

void scan\_disk\_scheduling(int requests[], int n, int start, int disk\_size) {

int total\_seek\_time = 0;

int current\_position = start;

int direction = 1; // 1 means move right, -1 means move left

```

- \*\*`total\_seek\_time`:\*\* Keeps track of the total seek time as the disk head moves.

- \*\*`current\_position`:\*\* Tracks the current position of the disk head.

- \*\*`direction`:\*\* A variable to control the direction of the disk head movement (right = 1, left = -1). Initially, the direction is set to move right (1).

```c

// Sort the requests array to process them in order

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (requests[j] > requests[j + 1]) {

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

```

- The `requests[]` array is sorted in ascending order using the \*\*Bubble Sort\*\* algorithm. This ensures that requests are processed in the correct order (from lowest to highest).

```c

// Find the first request greater than or equal to the current position

int index = 0;

while (index < n && requests[index] < current\_position) {

index++;

}

```

- The code identifies the first request greater than or equal to the current position of the disk head. This will help in determining where to start processing requests.

```c

printf("\nSCAN Disk Scheduling\n");

printf("%-20s%-20s%-20s\n", "Current Position", "Requested Track", "Seek Time");

```

- Prints the header of the output in a tabular format: showing the current position, requested track, and seek time.

#### \*\*2. Processing Requests:\*\*

After sorting the requests, the function moves the disk head in the \*\*right\*\* direction first (towards the highest track):

```c

// Move right first (if direction = 1)

if (direction == 1) {

// Process requests from the current position to the highest track (rightward)

for (int i = index; i < n; i++) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

// Move to the disk's maximum position (disk size)

int seek\_time = abs(current\_position - disk\_size);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, disk\_size, seek\_time);

current\_position = disk\_size;

```

- \*\*Moving Rightward:\*\*

- Starting from the request closest to the current position (tracked by `index`), the function services all requests in the rightward direction (i.e., towards the maximum disk size).

- After processing all the rightward requests, the disk head moves to the farthest position on the disk (i.e., `disk\_size`), and the seek time to reach that position is calculated and added.

```c

// Process remaining requests from the maximum position (leftward)

for (int i = index - 1; i >= 0; i--) {

int seek\_time = abs(current\_position - requests[i]);

total\_seek\_time += seek\_time;

printf("%-20d%-20d%-20d\n", current\_position, requests[i], seek\_time);

current\_position = requests[i];

}

}

```

- \*\*Moving Leftward:\*\*

- After reaching the maximum position (`disk\_size`), the disk head reverses direction (leftward) and processes the remaining requests in reverse order (from highest to lowest).

- The seek time for each request is computed, and the `total\_seek\_time` is updated accordingly.

```c

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

```

- After processing all the requests, the total seek time is displayed.

#### \*\*3. `main()` function:\*\*

The `main()` function handles the user inputs and calls the `scan\_disk\_scheduling()` function to simulate the disk scheduling.

```c

int main() {

int n, start\_position, disk\_size;

printf("Enter the starting position of the disk head: ");

scanf("%d", &start\_position);

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter the disk size: ");

scanf("%d", &disk\_size);

scan\_disk\_scheduling(requests, n, start\_position, disk\_size);

return 0;

}

```

\*\*Step-by-Step Explanation:\*\*

- \*\*Input:\*\*

- The user is prompted to input the starting position of the disk head (`start\_position`), the number of disk requests (`n`), the disk size (`disk\_size`), and the actual disk requests (`requests[]`).

- \*\*Call `scan\_disk\_scheduling()`:\*\*

- Once the inputs are received, the program calls the `scan\_disk\_scheduling()` function to simulate the SCAN disk scheduling algorithm.

### \*\*Example Input and Output:\*\*

#### \*\*Input:\*\*

```

Enter the starting position of the disk head: 50

Enter the number of disk requests: 6

Enter the disk requests:

82 170 43 140 24 60

Enter the disk size: 200

```

#### \*\*Output:\*\*

```

SCAN Disk Scheduling

Current Position Requested Track Seek Time

50 60 10

60 82 22

82 140 58

140 170 30

170 200 30

200 24 176

24 43 19

43 50 7

Total Seek Time: 352

```

### \*\*Explanation of Output:\*\*

1. \*\*Sorting the Requests:\*\* The requests are sorted in ascending order: `24, 43, 50, 60, 82, 140, 170`.

2. \*\*Service Requests Moving Right:\*\* Starting from `50`, the requests are processed in the right direction (`50 → 60 → 82 → 140 → 170 → 200`).

3. \*\*Service Requests Moving Left:\*\* Once the head reaches `200`, it reverses direction and processes requests in the leftward direction (`200 → 24 → 43 → 50`).

4. \*\*Seek Time Calculation:\*\* The seek time for each request is computed, and the total seek time is displayed.

### \*\*Key Concepts:\*\*

- \*\*SCAN Disk Scheduling:\*\* The disk head moves in one direction, services all requests in that direction, and then reverses direction when it reaches the end of the disk.

- \*\*Seek Time Calculation:\*\* The seek time is the absolute distance between the current position and the requested track.

- \*\*Sorting Requests:\*\* Sorting the requests helps in efficiently processing them in one direction (either left or right).

### \*\*Conclusion:\*\*

This program simulates the SCAN disk scheduling algorithm, providing a step-by-step simulation of how the disk head processes disk requests. The total seek time is calculated and displayed at the end, helping to evaluate the performance of the SCAN scheduling approach.

\*/

-------------22\_zoombie-----------  
#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

void create\_zombie\_process() {

pid\_t pid = fork();

if (pid > 0) {

// Parent process creates a zombie

printf("Parent process (PID: %d) created a zombie process (PID: %d)\n", getpid(), pid);

sleep(1); // Reduced sleep time to 1 second to minimize wait time for zombie

} else if (pid == 0) {

// Child process exits to become a zombie

printf("Child process (PID: %d) exiting to become a zombie\n", getpid());

exit(0);

} else {

perror("Fork failed for zombie process");

}

}

void create\_orphan\_process() {

pid\_t pid = fork();

if (pid > 0) {

// Parent exits to make the child an orphan

printf("Parent process (PID: %d) exiting, creating an orphan process\n", getpid());

exit(0);

} else if (pid == 0) {

// Child process becomes orphaned and adopted by init (PID 1)

sleep(1); // Keep child alive briefly to show orphan behavior

printf("Orphan child process (PID: %d) now adopted by init process (PPID: %d)\n", getpid(), getppid());

} else {

perror("Fork failed for orphan process");

}

}

void calculate\_even\_odd\_sum(int arr[], int n) {

pid\_t pid = fork();

if (pid > 0) {

// Parent calculates sum of even numbers

int even\_sum = 0;

for (int i = 0; i < n; i++) {

if (arr[i] % 2 == 0) {

even\_sum += arr[i];

}

}

wait(NULL); // Wait for child process to finish

printf("Parent process (PID: %d) - Sum of even numbers: %d\n", getpid(), even\_sum);

} else if (pid == 0) {

// Child calculates sum of odd numbers

int odd\_sum = 0;

for (int i = 0; i < n; i++) {

if (arr[i] % 2 != 0) {

odd\_sum += arr[i];

}

}

printf("Child process (PID: %d) - Sum of odd numbers: %d\n", getpid(), odd\_sum);

exit(0); // Exit child process after completing calculation

} else {

perror("Fork failed for even-odd sum calculation");

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Hardcoded array

int n = sizeof(arr) / sizeof(arr[0]);

// Task 1: Create a zombie process

create\_zombie\_process();

sleep(1); // Reduced sleep time to 1 second to observe zombie process briefly

// Task 2: Create an orphan process

create\_orphan\_process();

sleep(1); // Reduced sleep time to 1 second to observe orphan process briefly

// Task 3: Calculate sum of even and odd numbers using parent and child process

calculate\_even\_odd\_sum(arr, n);

return 0;

}

/\*

Certainly! Let's go through the entire code step by step so that you can understand the logic behind it.

### Code Overview:

This C program demonstrates three concepts:

1. \*\*Zombie Process\*\*: A process that has completed execution but still has an entry in the process table because its parent hasn't yet read its exit status.

2. \*\*Orphan Process\*\*: A process whose parent has terminated, so it is adopted by the `init` process (PID 1).

3. \*\*Sum Calculation\*\*: The parent calculates the sum of even numbers from an array, while the child calculates the sum of odd numbers.

### Code Breakdown:

```c

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

```

- \*\*`#include <stdio.h>`\*\*: Includes the standard I/O library for using functions like `printf()`.

- \*\*`#include <stdlib.h>`\*\*: Includes the standard library for functions like `exit()` and `malloc()`.

- \*\*`#include <sys/types.h>`\*\*: Includes the system types needed for system calls, such as `pid\_t` for process IDs.

- \*\*`#include <sys/wait.h>`\*\*: Includes functions for waiting for child processes, such as `wait()`.

- \*\*`#include <unistd.h>`\*\*: Includes functions for process control, such as `fork()`, `getpid()`, and `getppid()`.

### Functions Explained:

#### 1. `create\_zombie\_process()`

This function creates a zombie process:

```c

void create\_zombie\_process() {

pid\_t pid = fork(); // Create a child process

```

- \*\*`pid\_t pid = fork();`\*\*: The `fork()` system call is used to create a new process. This will return:

- A \*\*positive value\*\* (child's PID) to the parent process.

- `0` to the child process.

- `-1` if the `fork()` fails.

```c

if (pid > 0) {

// Parent process creates a zombie

printf("Parent process (PID: %d) created a zombie process (PID: %d)\n", getpid(), pid);

sleep(1); // Sleep for 1 second to keep the child as a zombie briefly

} else if (pid == 0) {

// Child process exits to become a zombie

printf("Child process (PID: %d) exiting to become a zombie\n", getpid());

exit(0); // Child exits immediately, becoming a zombie

} else {

perror("Fork failed for zombie process");

}

}

```

- \*\*Parent Process\*\* (`pid > 0`):

- The parent prints a message showing it created a child (which will become a zombie).

- The parent calls `sleep(1)` to wait for 1 second, allowing the child to become a zombie.

- \*\*Child Process\*\* (`pid == 0`):

- The child prints a message and immediately exits with `exit(0)`.

- Since the parent hasn't yet called `wait()` to collect the child’s exit status, the child remains in the \*\*zombie\*\* state for a short period.

#### 2. `create\_orphan\_process()`

This function demonstrates the creation of an orphan process:

```c

void create\_orphan\_process() {

pid\_t pid = fork(); // Create a child process

```

- \*\*`pid\_t pid = fork();`\*\*: Again, this creates a child process, returning the process ID (PID) for the parent and `0` for the child.

```c

if (pid > 0) {

// Parent exits to make the child an orphan

printf("Parent process (PID: %d) exiting, creating an orphan process\n", getpid());

exit(0); // Parent exits, making the child an orphan

} else if (pid == 0) {

// Child process becomes orphaned and adopted by init (PID 1)

sleep(1); // Sleep for 1 second to show orphan behavior

printf("Orphan child process (PID: %d) now adopted by init process (PPID: %d)\n", getpid(), getppid());

} else {

perror("Fork failed for orphan process");

}

}

```

- \*\*Parent Process\*\* (`pid > 0`):

- The parent exits immediately using `exit(0)`, causing the child to become an \*\*orphan\*\*. The orphaned child will be adopted by the `init` process (with PID 1).

- \*\*Child Process\*\* (`pid == 0`):

- The child sleeps for 1 second to simulate the orphaned state, and then prints that it has been adopted by `init` (i.e., its new parent is `init` with PID 1).

#### 3. `calculate\_even\_odd\_sum(int arr[], int n)`

This function calculates the sum of even and odd numbers in an array using the parent and child processes:

```c

void calculate\_even\_odd\_sum(int arr[], int n) {

pid\_t pid = fork(); // Create a child process

```

- \*\*`pid\_t pid = fork();`\*\*: The `fork()` creates a child process that will handle calculating the sum of odd numbers, while the parent calculates the sum of even numbers.

```c

if (pid > 0) {

// Parent calculates sum of even numbers

int even\_sum = 0;

for (int i = 0; i < n; i++) {

if (arr[i] % 2 == 0) {

even\_sum += arr[i];

}

}

wait(NULL); // Wait for child process to finish

printf("Parent process (PID: %d) - Sum of even numbers: %d\n", getpid(), even\_sum);

} else if (pid == 0) {

// Child calculates sum of odd numbers

int odd\_sum = 0;

for (int i = 0; i < n; i++) {

if (arr[i] % 2 != 0) {

odd\_sum += arr[i];

}

}

printf("Child process (PID: %d) - Sum of odd numbers: %d\n", getpid(), odd\_sum);

exit(0); // Exit the child process after completing its task

} else {

perror("Fork failed for even-odd sum calculation");

}

}

```

- \*\*Parent Process\*\* (`pid > 0`):

- The parent calculates the sum of even numbers by iterating through the array. It uses `wait(NULL)` to ensure that it waits for the child to complete its task before printing the sum of even numbers.

- \*\*Child Process\*\* (`pid == 0`):

- The child calculates the sum of odd numbers and prints the result. After finishing the calculation, the child exits using `exit(0)`.

### `main()` Function:

The `main()` function is where the program executes and calls the functions to demonstrate the behaviors:

```c

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Hardcoded array of numbers

int n = sizeof(arr) / sizeof(arr[0]); // Get the size of the array

// Task 1: Create a zombie process

create\_zombie\_process();

sleep(1); // Sleep briefly to observe the zombie process

// Task 2: Create an orphan process

create\_orphan\_process();

sleep(1); // Sleep briefly to observe the orphan process

// Task 3: Calculate sum of even and odd numbers using parent and child process

calculate\_even\_odd\_sum(arr, n);

return 0;

}

```

- \*\*Array\*\*: An array of numbers `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}` is used.

- \*\*Size Calculation\*\*: `n = sizeof(arr) / sizeof(arr[0])` calculates the number of elements in the array.

- \*\*Function Calls\*\*: The three functions (`create\_zombie\_process()`, `create\_orphan\_process()`, and `calculate\_even\_odd\_sum()`) are called sequentially.

### Expected Output:

For the array `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`, the output should be something like this:

```text

Parent process (PID: 1234) created a zombie process (PID: 1235)

Child process (PID: 1235) exiting to become a zombie

Orphan child process (PID: 1235) now adopted by init process (PPID: 1)

Parent process (PID: 1234) - Sum of even numbers: 30

Child process (PID: 1235) - Sum of odd numbers: 25

```

### Summary:

- \*\*Zombie Process\*\*: A child process that terminates but remains in the process table until the parent reads its exit status.

- \*\*Orphan Process\*\*: A child process whose parent terminates. It is adopted by the `init` process.

- \*\*Even/Odd Sum Calculation\*\*: The parent calculates the sum of even numbers, and the child calculates the sum of odd numbers from the array.

Let me know if you need more details or further clarification!

\*/

---------23 Student Database ---------

#!/bin/bash

# File to store student records

DB\_FILE="students.txt"

# Function to display menu

show\_menu() {

echo "Choose an operation:"

echo "1) Insert"

echo "2) Delete"

echo "3) Update"

echo "4) Search"

echo "5) Exit"

}

# Function to insert a new student record

insert\_student() {

read -p "Enter Student ID: " id

read -p "Enter Student Name: " name

read -p "Enter Student Age: " age

read -p "Enter Student Grade: " grade

echo "$id|$name|$age|$grade" >> $DB\_FILE

echo "Student record inserted successfully."

}

# Function to delete a student record by ID

delete\_student() {

read -p "Enter Student ID to delete: " id

grep -v "^$id|" $DB\_FILE > temp.txt && mv temp.txt $DB\_FILE

echo "Student record deleted if ID was found."

}

# Function to update a student record by ID

update\_student() {

read -p "Enter Student ID to update: " id

if grep -q "^$id|" $DB\_FILE; then

grep -v "^$id|" $DB\_FILE > temp.txt

mv temp.txt $DB\_FILE

read -p "Enter new Name: " name

read -p "Enter new Age: " age

read -p "Enter new Grade: " grade

echo "$id|$name|$age|$grade" >> $DB\_FILE

echo "Student record updated successfully."

else

echo "Student ID not found."

fi

}

# Function to search for a student record by ID

search\_student() {

read -p "Enter Student ID to search: " id

grep "^$id|" $DB\_FILE && echo "Record found." || echo "Record not found."

}

# Main script loop

while true; do

show\_menu

read -p "Enter your choice: " choice

case $choice in

1) insert\_student ;;

2) delete\_student ;;

3) update\_student ;;

4) search\_student ;;

5) exit 0 ;;

\*) echo "Invalid choice, please try again." ;;

esac

done

----------------------------------------------------

This Bash script manages a \*\*student database\*\* stored in a file (`students.txt`) and allows the user to perform four primary operations on it: \*\*Insert\*\*, \*\*Delete\*\*, \*\*Update\*\*, and \*\*Search\*\*. These operations are performed through a simple text-based menu system. Below is a detailed explanation of how the script works:

### 1. \*\*Database File (`students.txt`)\*\*:

The script interacts with a text file called `students.txt` that serves as the simulated database for student records. Each record in this file consists of a student's ID, name, age, and grade, separated by a pipe (`|`) symbol. An example record might look like:

```

1|John Doe|20|A

2|Jane Smith|22|B

```

---

### 2. \*\*Functions Defined in the Script\*\*:

#### 2.1 \*\*Menu Function (`show\_menu`)\*\*:

This function displays the menu options to the user.

```bash

show\_menu() {

echo "Choose an operation:"

echo "1) Insert"

echo "2) Delete"

echo "3) Update"

echo "4) Search"

echo "5) Exit"

}

```

- It prints out five choices:

- \*\*1\*\*: Insert a new student record.

- \*\*2\*\*: Delete an existing student record.

- \*\*3\*\*: Update an existing student record.

- \*\*4\*\*: Search for a student by ID.

- \*\*5\*\*: Exit the script.

#### 2.2 \*\*Insert Function (`insert\_student`)\*\*:

This function inserts a new student record into the database.

```bash

insert\_student() {

read -p "Enter Student ID: " id

read -p "Enter Student Name: " name

read -p "Enter Student Age: " age

read -p "Enter Student Grade: " grade

echo "$id|$name|$age|$grade" >> $DB\_FILE

echo "Student record inserted successfully."

}

```

- The user is prompted to enter the student's \*\*ID\*\*, \*\*Name\*\*, \*\*Age\*\*, and \*\*Grade\*\*.

- The entered data is formatted into the format `id|name|age|grade`, and then appended to the `students.txt` file using `echo "$id|$name|$age|$grade" >> $DB\_FILE`.

- Once the record is added, a success message is displayed.

#### 2.3 \*\*Delete Function (`delete\_student`)\*\*:

This function deletes a student record by their \*\*ID\*\*.

```bash

delete\_student() {

read -p "Enter Student ID to delete: " id

grep -v "^$id|" $DB\_FILE > temp.txt && mv temp.txt $DB\_FILE

echo "Student record deleted if ID was found."

}

```

- The user is asked to provide a \*\*Student ID\*\* to delete.

- `grep -v "^$id|" $DB\_FILE` searches for all lines in `students.txt` that do not start with the given \*\*ID\*\* (the `^` signifies the start of the line, and `|` ensures it matches the entire ID).

- The non-matching lines are saved to a temporary file (`temp.txt`), and then the `temp.txt` file is moved back to `students.txt`, effectively deleting the line corresponding to the provided \*\*ID\*\*.

- A message is printed to inform the user that the record was deleted (if the ID was found).

#### 2.4 \*\*Update Function (`update\_student`)\*\*:

This function allows the user to update a student record by \*\*ID\*\*.

```bash

update\_student() {

read -p "Enter Student ID to update: " id

if grep -q "^$id|" $DB\_FILE; then

grep -v "^$id|" $DB\_FILE > temp.txt

mv temp.txt $DB\_FILE

read -p "Enter new Name: " name

read -p "Enter new Age: " age

read -p "Enter new Grade: " grade

echo "$id|$name|$age|$grade" >> $DB\_FILE

echo "Student record updated successfully."

else

echo "Student ID not found."

fi

}

```

- The user is prompted for the \*\*Student ID\*\* to update.

- The script checks if the \*\*ID\*\* exists in the file using `grep -q "^$id|" $DB\_FILE`.

- If the \*\*ID\*\* is found:

- It removes the existing record by excluding the line that starts with the given \*\*ID\*\* (`grep -v "^$id|"`).

- The updated student details are then requested (new \*\*Name\*\*, \*\*Age\*\*, \*\*Grade\*\*).

- The new record is appended to `students.txt` in the format `id|name|age|grade`.

- If the \*\*ID\*\* is not found, a message saying "Student ID not found" is displayed.

#### 2.5 \*\*Search Function (`search\_student`)\*\*:

This function allows the user to search for a student record by \*\*ID\*\*.

```bash

search\_student() {

read -p "Enter Student ID to search: " id

grep "^$id|" $DB\_FILE && echo "Record found." || echo "Record not found."

}

```

- The user is asked for the \*\*Student ID\*\* to search.

- The script uses `grep "^$id|"` to search for a line in `students.txt` that starts with the given \*\*ID\*\*.

- If the record is found, it prints "Record found". If not, it prints "Record not found".

---

### 3. \*\*Main Script Loop\*\*:

The script enters a loop that continuously presents the user with the menu options until the user chooses to exit:

```bash

while true; do

show\_menu

read -p "Enter your choice: " choice

case $choice in

1) insert\_student ;;

2) delete\_student ;;

3) update\_student ;;

4) search\_student ;;

5) exit 0 ;;

\*) echo "Invalid choice, please try again." ;;

esac

done

```

- The `while true` loop runs indefinitely, displaying the menu and asking for a user choice.

- Based on the user's choice (`$choice`), the script calls the appropriate function:

- \*\*1\*\*: Calls `insert\_student` to add a record.

- \*\*2\*\*: Calls `delete\_student` to delete a record.

- \*\*3\*\*: Calls `update\_student` to modify a record.

- \*\*4\*\*: Calls `search\_student` to look up a record.

- \*\*5\*\*: Exits the script.

- If an invalid choice is entered, it prints an error message.

### 4. \*\*Example Usage\*\*:

- \*\*Insert\*\*: When the user chooses option 1, they will be prompted for the student details, which are then added to `students.txt`.

- \*\*Delete\*\*: When the user chooses option 2, the student record with the specified \*\*ID\*\* is removed from the file.

- \*\*Update\*\*: When the user chooses option 3, they are asked to enter a new name, age, and grade for the student with the given \*\*ID\*\*.

- \*\*Search\*\*: When the user chooses option 4, they can search for a student by \*\*ID\*\* and see if the record exists in the database.

---

### 5. \*\*Advantages and Limitations\*\*:

#### Advantages:

- \*\*Simple User Interface\*\*: It provides a clear, text-based menu for interacting with the student records.

- \*\*CRUD Operations\*\*: It supports basic CRUD (Create, Read, Update, Delete) operations on the student database.

#### Limitations:

- \*\*Data Persistence\*\*: The script directly edits `students.txt`, which means any changes to the file will persist. If the script is terminated abruptly, there could be issues with file integrity.

- \*\*No Validation\*\*: The script does not validate the input data (e.g., checking that \*\*Age\*\* is a number or \*\*Grade\*\* is a valid grade).

---

### Conclusion:

This script is an effective, simple solution to manage student records with basic operations like insert, delete, update, and search. It uses a text file as a simple database and offers a user-friendly interface through a menu system.

----------------- Read and Copy contents -------------------------

#!/bin/bash

# Input and output file names

read -p "Enter the source file name: " source\_file

read -p "Enter the destination file name: " destination\_file

# Check if the source file exists

if [ ! -f "$source\_file" ]; then

echo "Source file does not exist."

exit 1

fi

# Copy contents character by character

echo "Copying contents character by character..."

while IFS= read -r -n1 char; do

echo -n "$char" >> "$destination\_file"

done < "$source\_file"

# Adding a newline after character copy

echo "" >> "$destination\_file"

# Copy contents line by line

echo "Copying contents line by line..."

while IFS= read -r line; do

echo "$line" >> "$destination\_file"

done < "$source\_file"

echo "Contents copied successfully to $destination\_file."

source.txt

Hello,

This is a test file.

Goodbye!

--------------------------

Sure! Let's go through your Bash script step by step to understand what each part does.

### Script Overview:

This script is designed to:

1. Copy the contents of a source file to a destination file.

2. Copy the contents \*\*character by character\*\* first.

3. Copy the contents \*\*line by line\*\* after the character-based copy.

4. Check for the existence of the source file before proceeding.

---

### Step-by-Step Explanation:

1. \*\*Prompting for Input:\*\*

```bash

read -p "Enter the source file name: " source\_file

read -p "Enter the destination file name: " destination\_file

```

- The script starts by prompting the user to input the names of the \*\*source\*\* and \*\*destination\*\* files.

- `source\_file` will hold the name of the file to be copied.

- `destination\_file` will be the name of the file where the contents will be copied to.

2. \*\*Check if Source File Exists:\*\*

```bash

if [ ! -f "$source\_file" ]; then

echo "Source file does not exist."

exit 1

fi

```

- This part checks whether the \*\*source file\*\* exists.

- The condition `[ ! -f "$source\_file" ]` checks if the file does \*\*not\*\* exist (`-f` checks if the file exists and is a regular file).

- If the source file doesn't exist, an error message is displayed (`"Source file does not exist."`), and the script exits with a status code of `1` (`exit 1`).

3. \*\*Copying Contents Character by Character:\*\*

```bash

echo "Copying contents character by character..."

while IFS= read -r -n1 char; do

echo -n "$char" >> "$destination\_file"

done < "$source\_file"

```

- The `while IFS= read -r -n1 char` loop reads the \*\*source file\*\* one character at a time.

- `IFS=`: Disables internal field splitting, ensuring each character is read individually.

- `-r`: Prevents backslashes from being interpreted as escape characters (e.g., no special handling of `\n`).

- `-n1`: Specifies that only one character should be read at a time.

- The `echo -n "$char" >> "$destination\_file"` line writes each character to the \*\*destination file\*\*.

- The `-n` flag ensures that `echo` does \*\*not\*\* append a newline after writing each character, so the characters are written exactly as they appear in the source file.

- The loop continues until all characters from the \*\*source file\*\* are copied to the \*\*destination file\*\*.

4. \*\*Adding a Newline After Character Copy:\*\*

```bash

echo "" >> "$destination\_file"

```

- After copying the file character by character, a \*\*newline\*\* (`""`) is appended to the \*\*destination file\*\*.

- This ensures that there is a separation between the character copy and the line-by-line copy.

5. \*\*Copying Contents Line by Line:\*\*

```bash

echo "Copying contents line by line..."

while IFS= read -r line; do

echo "$line" >> "$destination\_file"

done < "$source\_file"

```

- This part of the script reads the \*\*source file\*\* line by line.

- `IFS= read -r line` reads one entire line from the \*\*source file\*\* at a time:

- `IFS=`: Again, prevents field splitting.

- `-r`: Ensures backslashes are treated literally (not as escape characters).

- The `echo "$line" >> "$destination\_file"` line appends each line to the \*\*destination file\*\*. This time, the lines are written as they are, including the newline at the end of each line.

6. \*\*Completion Message:\*\*

```bash

echo "Contents copied successfully to $destination\_file."

```

- Once both character-by-character and line-by-line copying are done, this message is displayed to inform the user that the copying process has completed successfully.

---

### Example Walkthrough:

#### Source File (`source.txt`):

```

Hello,

This is a test file.

Goodbye!

```

#### What the Script Does:

1. \*\*Character-by-Character Copy\*\*:

- It copies the contents of `source.txt` to `destination.txt` one character at a time. After copying each character, there will be no newlines until all characters are copied.

- After this step, the destination file will look like this (without newlines yet):

```

Hello,This is a test file.Goodbye!

```

2. \*\*Adding Newline\*\*:

- After copying all characters, the script appends a newline in the destination file.

- The destination file will look like this now:

```

Hello,This is a test file.Goodbye!

```

3. \*\*Line-by-Line Copy\*\*:

- The script then copies the entire contents of `source.txt` line by line into `destination.txt`.

- After this step, the destination file will look like this:

```

Hello,This is a test file.Goodbye!

Hello,

This is a test file.

Goodbye!

```

---

### Final Thoughts:

- \*\*Character-by-Character Copy\*\*: This part is useful if you need a very granular copy process, or if you're simulating some kind of low-level file reading.

- \*\*Line-by-Line Copy\*\*: This is a more conventional way to copy files, as it preserves the file structure and formatting.

- \*\*Performance\*\*: Copying character by character is generally inefficient compared to copying the entire file or copying line by line. For large files, using commands like `cp` or `cat` would be faster.

Let me know if you need further clarifications!

--------------- ALP IP File to memory

#!/bin/bash

# Simulated main memory file

MEMORY\_FILE="main\_memory.txt"

# Input ALP file

read -p "Enter the ALP input file name: " input\_file

# Check if the input ALP file exists

if [ ! -f "$input\_file" ]; then

echo "Input file does not exist."

exit 1

fi

# Clear the memory file before loading new content

> "$MEMORY\_FILE"

# Load ALP into simulated main memory

echo "Loading ALP program into main memory..."

line\_number=1

while IFS= read -r line; do

echo "Loading line $line\_number: $line"

echo "$line" >> "$MEMORY\_FILE"

((line\_number++))

done < "$input\_file"

echo "ALP program loaded into main memory (stored in $MEMORY\_FILE)."

input\_file

MOV AX, 1

ADD BX, AX

INT 21h

-------------------

This Bash script simulates the loading of an \*\*Assembly Language Program (ALP)\*\* into a \*\*simulated main memory\*\*. The main memory is represented by a text file (`main\_memory.txt`), and the ALP program is read from an input file provided by the user. Let's break down each part of the script:

### Step-by-Step Explanation:

#### 1. \*\*Setting up the Simulated Main Memory File:\*\*

```bash

MEMORY\_FILE="main\_memory.txt"

```

- The variable `MEMORY\_FILE` is assigned the name `"main\_memory.txt"`. This will be used as a simulated \*\*main memory\*\* file where the ALP program will be loaded.

#### 2. \*\*Prompt for Input ALP File:\*\*

```bash

read -p "Enter the ALP input file name: " input\_file

```

- The `read` command prompts the user to enter the name of the \*\*ALP input file\*\*. The user will input the name of the file that contains the ALP program, and it will be stored in the variable `input\_file`.

#### 3. \*\*Check if the Input ALP File Exists:\*\*

```bash

if [ ! -f "$input\_file" ]; then

echo "Input file does not exist."

exit 1

fi

```

- This block checks if the file specified by the user (`input\_file`) exists.

- The condition `[ ! -f "$input\_file" ]` checks if the file doesn't exist (`-f` checks for regular files).

- If the file doesn't exist, it prints `"Input file does not exist."` and exits the script with a status code `1`.

#### 4. \*\*Clear the Memory File:\*\*

```bash

> "$MEMORY\_FILE"

```

- This command clears the content of the \*\*simulated main memory file\*\* (`main\_memory.txt`), ensuring it's empty before loading new content.

- The `>` operator is used to truncate the file to zero size.

#### 5. \*\*Load ALP Program into Simulated Memory:\*\*

```bash

echo "Loading ALP program into main memory..."

line\_number=1

while IFS= read -r line; do

echo "Loading line $line\_number: $line"

echo "$line" >> "$MEMORY\_FILE"

((line\_number++))

done < "$input\_file"

```

- The script uses a `while` loop to read the \*\*ALP input file\*\* (`$input\_file`) line by line and load it into the simulated \*\*main memory\*\* (represented by `main\_memory.txt`).

- `IFS= read -r line` reads each line from the input file:

- `IFS=`: Disables field splitting (prevents spaces from being treated as delimiters).

- `-r`: Prevents backslashes from being interpreted as escape characters.

- For each line, the script:

- Prints the line number and the line content (`echo "Loading line $line\_number: $line"`).

- Appends the line to the `main\_memory.txt` file using `echo "$line" >> "$MEMORY\_FILE"`.

- Increments the `line\_number` to track the line count.

- This continues until all lines in the input file are read and copied to the simulated memory.

#### 6. \*\*Final Message:\*\*

```bash

echo "ALP program loaded into main memory (stored in $MEMORY\_FILE)."

```

- After the program finishes reading the input ALP file and loading it into the memory file, it prints a success message indicating that the ALP program has been loaded into memory.

- The message specifies that the program is stored in `main\_memory.txt`.

---

### Example Walkthrough:

#### 1. \*\*Input ALP File (`input\_file`)\*\*:

Suppose the user enters a file named `program.asm` that contains the following assembly instructions:

```assembly

MOV AX, 1

ADD BX, AX

INT 21h

```

#### 2. \*\*Execution Flow\*\*:

- The script will check if the file `program.asm` exists. If not, it will print an error and exit.

- If the file exists, it clears `main\_memory.txt`.

- The script then reads each line of the `program.asm` file and appends it to `main\_memory.txt`:

- Line 1: `MOV AX, 1`

- Line 2: `ADD BX, AX`

- Line 3: `INT 21h`

- Finally, it prints a message indicating that the ALP program has been successfully loaded into `main\_memory.txt`.

#### 3. \*\*Output (`main\_memory.txt`)\*\*:

After running the script, the \*\*simulated main memory file\*\* (`main\_memory.txt`) will look like this:

```

MOV AX, 1

ADD BX, AX

INT 21h

```

---

### Summary:

- \*\*Purpose\*\*: This script simulates loading an ALP program into \*\*main memory\*\* (represented by a text file).

- \*\*File Handling\*\*: It reads the input ALP file line by line and appends each line to the memory file.

- \*\*Validation\*\*: It checks if the input file exists before proceeding and clears the memory file before loading new content.

This script could be a part of a basic simulation or a tool for processing assembly language programs, where the assembly instructions are loaded into memory for further operations or analysis. Let me know if you need further clarification!

--------------**26\_Opcode\_Error.c----------------------------\**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define the maximum number of valid opcodes

#define MAX\_OPCODES 5

// Function to simulate an interrupt (error)

void raise\_interrupt(const char \*error\_message) {

printf("Error: %s\n", error\_message);

printf("Interrupt raised! Exiting the program...\n");

exit(1); // Terminate the program on error

}

// Function to check if the opcode is valid

int is\_valid\_opcode(const char \*opcode) {

// List of valid opcodes

const char \*valid\_opcodes[MAX\_OPCODES] = {"LOAD", "ADD", "SUB", "STORE", "JMP"};

for (int i = 0; i < MAX\_OPCODES; i++) {

if (strcmp(opcode, valid\_opcodes[i]) == 0) {

return 1; // Valid opcode

}

}

return 0; // Invalid opcode

}

// Function to process the job (list of instructions)

void process\_job(const char instructions[][10], int num\_instructions) {

for (int i = 0; i < num\_instructions; i++) {

printf("Processing instruction: %s\n", instructions[i]);

// Check if the opcode is valid

if (!is\_valid\_opcode(instructions[i])) {

raise\_interrupt("Invalid opcode encountered!");

}

}

printf("All instructions processed successfully.\n");

}

int main() {

int num\_instructions;

// Ask the user for the number of instructions

printf("Enter the number of instructions: ");

scanf("%d", &num\_instructions);

// Array to store instructions

char instructions[num\_instructions][10];

// Take input instructions from the user

printf("Enter the instructions (one per line):\n");

for (int i = 0; i < num\_instructions; i++) {

scanf("%s", instructions[i]);

}

// Process the job and check for opcode errors

process\_job(instructions, num\_instructions);

return 0;

}

/\*

Enter the number of instructions: 3

Enter the instructions (one per line):

LOAD

ADD

STORE

\*/

/\*

#include <stdio.h>: This header is for input and output functions (printf, scanf, etc.).

#include <stdlib.h>: Provides functions for memory allocation, process control, and other utilities (exit(), malloc(), etc.).

#include <string.h>: Provides string manipulation functions like strcmp().

#define MAX\_OPCODES 5: This is a macro that defines the maximum number of valid opcodes (5 in this case).

Purpose: This function simulates an interrupt, which is a kind of error that forces the program to stop.

Parameters: It takes an error\_message string as input, which describes the error encountered.

Logic:

It prints the error message to the console using printf().

It then prints an additional message saying that an interrupt was raised and the program will terminate.

Finally, the program is terminated using exit(1), where 1 is the exit status (indicating failure).

Purpose: This function checks whether a given opcode (instruction) is valid by comparing it against a predefined list of valid opcodes.

Parameters: opcode is the opcode (instruction) to be checked.

Logic:

The function defines an array valid\_opcodes[] that contains all the valid opcodes (in this case: "LOAD", "ADD", "SUB", "STORE", "JMP").

The function uses a loop to compare the provided opcode with each valid opcode using strcmp().

If strcmp() finds a match, it returns 1, indicating that the opcode is valid.

If no match is found after checking all valid opcodes, it returns 0, indicating an invalid opcode.

Purpose: This function processes a list of instructions (opcodes) and checks each one for validity.

Parameters:

instructions[][10]: A 2D array containing the instructions. The array is designed to store up to num\_instructions opcodes, each with a maximum length of 10 characters.

num\_instructions: The number of instructions to process.

Logic:

The function loops over each instruction in the instructions array.

For each instruction, it prints a message saying that the instruction is being processed.

It then checks if the opcode is valid by calling the is\_valid\_opcode() function.

If an invalid opcode is found, it raises an interrupt using the raise\_interrupt() function and stops the program.

If all instructions are valid, it prints a success message.

Purpose: The main() function is the entry point of the program. It interacts with the user, accepts input for the number of instructions and the instructions themselves, and processes them.

Logic:

It first asks the user for the number of instructions they want to input (num\_instructions).

It then declares a 2D array instructions[num\_instructions][10] to store the instructions. The 10 indicates that each instruction can be up to 9 characters long (plus a null terminator).

The program then asks the user to input the instructions one by one. The scanf("%s", instructions[i]); reads each instruction as a string and stores it in the instructions array.

After all instructions are input, the program calls process\_job() to check each instruction for validity.

If all instructions are valid, the program prints "All instructions processed successfully." Otherwise, it will terminate upon encountering an invalid opcode.

Summary of How the Code Works:

The program first asks the user how many instructions they will input and then prompts them to enter each instruction one by one.

For each instruction, the program checks if it matches any of the predefined valid opcodes (LOAD, ADD, SUB, STORE, JMP).

If an invalid opcode is found, the program raises an error and terminates. If all instructions are valid, it prints a success message.

If an invalid opcode is encountered, the program calls the raise\_interrupt() function, which prints an error message and terminates the program using exit(1).

\*/

27 Operand Code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define maximum number of valid opcodes and operand range

#define MAX\_OPCODES 5

#define MAX\_OPERAND 100 // Define the range of valid operands (e.g., 0 to 100)

// Function to simulate an interrupt (error)

void raise\_interrupt(const char \*error\_message) {

printf("Error: %s\n", error\_message);

printf("Interrupt raised! Exiting the program...\n");

exit(1); // Terminate the program on error

}

// Function to check if the opcode is valid

int is\_valid\_opcode(const char \*opcode) {

// List of valid opcodes

const char \*valid\_opcodes[MAX\_OPCODES] = {"LOAD", "ADD", "SUB", "STORE", "JMP"};

for (int i = 0; i < MAX\_OPCODES; i++) {

if (strcmp(opcode, valid\_opcodes[i]) == 0) {

return 1; // Valid opcode

}

}

return 0; // Invalid opcode

}

// Function to check if the operand is valid

int is\_valid\_operand(int operand) {

if (operand < 0 || operand > MAX\_OPERAND) {

return 0; // Invalid operand

}

return 1; // Valid operand

}

// Function to process the job (list of instructions and operands)

void process\_job(const char instructions[][10], int operands[], int num\_instructions) {

for (int i = 0; i < num\_instructions; i++) {

printf("Processing instruction: %s %d\n", instructions[i], operands[i]);

// Check if the opcode is valid

if (!is\_valid\_opcode(instructions[i])) {

raise\_interrupt("Invalid opcode encountered!");

}

// Check if the operand is valid

if (!is\_valid\_operand(operands[i])) {

raise\_interrupt("Invalid operand encountered!");

}

}

printf("All instructions processed successfully.\n");

}

int main() {

int num\_instructions;

// Ask the user for the number of instructions

printf("Enter the number of instructions: ");

scanf("%d", &num\_instructions);

// Arrays to store instructions and operands

char instructions[num\_instructions][10];

int operands[num\_instructions];

// Take input instructions and operands from the user

printf("Enter the instructions and operands (e.g., LOAD 25):\n");

for (int i = 0; i < num\_instructions; i++) {

scanf("%s %d", instructions[i], &operands[i]);

}

// Process the job and check for opcode and operand errors

process\_job(instructions, operands, num\_instructions);

return 0;

}

// Enter the number of instructions: 3

// Enter the instructions and operands (e.g., LOAD 25):

// LOAD 20

// ADD 10

// STORE 50

/\*

The code simulates a basic instruction processing system, where each instruction consists of an opcode (such as LOAD, ADD, SUB, etc.)

and an operand (a numeric value that accompanies the opcode).

The goal of the program is to check if each opcode is valid and if the operand is within an acceptable range (0 to 100).

If any instruction contains an invalid opcode or operand, the program raises an "interrupt" (simulated error) and exits.

MAX\_OPCODES: Specifies the maximum number of valid opcodes (5 in this case: LOAD, ADD, SUB, STORE, JMP).

MAX\_OPERAND: Defines the range for valid operands (0 to 100). If an operand is outside this range, it is considered invalid.

This function simulates an interrupt (an error in the system) by printing an error message and terminating the program with exit(1).

This is invoked whenever an invalid opcode or operand is encountered.

Function is\_valid\_opcode():

This function checks if a given opcode is valid.

It compares the provided opcode with the valid opcodes (LOAD, ADD, SUB, STORE, JMP) using strcmp().

If a match is found, it returns 1 (valid opcode). If no match is found, it returns 0 (invalid opcode).

unction is\_valid\_operand():

This function checks if the operand is within the valid range (0 to 100).

If the operand is outside this range, it returns 0 (invalid operand). Otherwise, it returns 1 (valid operand).

Function process\_job():

This function processes each instruction (opcode and operand) in a loop.

It prints each instruction as it is being processed.

It checks whether the opcode is valid using is\_valid\_opcode().

It checks whether the operand is valid using is\_valid\_operand().

If either the opcode or operand is invalid, it raises an interrupt by calling raise\_interrupt().

If all instructions are valid, the function prints "All instructions processed successfully.".

The program first asks the user for the number of instructions.

It then dynamically creates two arrays: instructions[] for the opcodes and operands[] for the operand values, based on the number of instructions entered.

The program prompts the user to input the opcode and operand for each instruction.

It calls the process\_job() function to process the instructions, check for errors, and print the result.

Overall Program Flow:

The user enters the number of instructions.

The user enters the instructions and operands.

The program processes each instruction:

Checks if the opcode is valid (matches one of LOAD, ADD, SUB, STORE, JMP).

Checks if the operand is valid (within the range 0-100).

If all instructions are valid, the program prints a success message.

If any instruction has an invalid opcode or operand, the program raises an interrupt and terminates.

\*/