Experiment 1

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

#!/bin/bash

address\_book="address\_book.txt"

# Function to create an empty address book

create\_book() {

> "$address\_book"

echo "Address book created."

}

# Function to view the address book

view\_book() {

if [[ -s $address\_book ]]; then

cat "$address\_book"

else

echo "Address book is empty."

fi

}

# Function to insert a new record

insert\_record() {

echo "Enter Name: "

read name

echo "Enter Phone: "

read phone

echo "$name, $phone" >> "$address\_book"

echo "Record inserted."

}

# Function to delete a record

delete\_record() {

echo "Enter Name to delete: "

read name

grep -v "^$name," "$address\_book" > temp && mv temp "$address\_book"

echo "Record deleted if it existed."

}

# Function to modify a record

modify\_record() {

echo "Enter Name to modify: "

read name

grep -v "^$name," "$address\_book" > temp && mv temp "$address\_book"

echo "Enter New Name: "

read new\_name

echo "Enter New Phone: "

read new\_phone

echo "$new\_name, $new\_phone" >> "$address\_book"

echo "Record modified."

}

# Menu loop

while true; do

echo "Choose an option:"

echo "a) Create address book"

echo "b) View address book"

echo "c) Insert a record"

echo "d) Delete a record"

echo "e) Modify a record"

echo "f) Exit"

read option

case $option in

a) create\_book ;;

b) view\_book ;;

c) insert\_record ;;

d) delete\_record ;;

e) modify\_record ;;

f) exit ;;

\*) echo "Invalid option." ;;

esac

done

**How to run the script:**

1. Save the script to a file, e.g., address\_book.sh.
2. Make the script executable:

chmod +x address\_book.sh

1. Run the script:

./address\_book.sh

Experiment 2

Implement the C program in which main program accepts an integer array. Main program uses the fork system call to create a new process called a child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve system call. The child process uses execve system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

parent.c

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <sys/wait.h>

void bubblesort(int a[], int n) {

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

for (int d=0; d<n-c-1; d++)

if (a[d] > a[d+1]) {

int swap = a[d];

a[d] = a[d+1];

a[d+1] = swap;

}

}

int main() {

int a[20], n;

pid\_t id = fork();

if (id == 0) { // Child process

printf("Enter number of elements: ");

scanf("%d", &n);

printf("Enter elements: ");

for (int i = 0; i < n; i++) scanf("%d", &a[i]);

bubblesort(a, n);

printf("\nSorted array: ");

for (int i = 0; i < n; i++) printf("%d ", a[i]);

char \*arg[20], str[20];

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

sprintf(str, "%d", a[i]);

arg[i] = strdup(str);

}

arg[n] = NULL;

execve("./child.out", arg, NULL);

exit(0);

} else {

wait(NULL);

printf("\nParent process ID: %d\n", getpid());

}

return 0;

}

child.c

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*\*argv) {

int a[20], key, first = 0, mid, last = argc - 1;

for (int i = 0; i < argc; i++) a[i] = atoi(argv[i]);

printf("Enter key to search: ");

scanf("%d", &key);

while (first <= last) {

mid = (first + last) / 2;

if (a[mid] == key) {

printf("Element %d found at position %d\n", key, mid+1);

return 0;

}

(a[mid] < key) ? (first = mid + 1) : (last = mid - 1);

}

printf("Element not found!\n");

return 0;

}

##How to run the code

gcc child.c -o child.out

gcc parent.c -o parent.out

./parent.out

Experiment No 3

AIM : Matrix Multiplication using POSIX pthread

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define SIZE 4 // Size of the matrices

int A[SIZE][SIZE] = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12},

{13, 14, 15, 16}

};

int B[SIZE][SIZE] = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12},

{13, 14, 15, 16}

};

int C[SIZE][SIZE]; // Result matrix

// Function to multiply a specific row

void\* multiply\_row(void\* row) {

int r = \*((int\*)row); // Get the row index

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

C[r][j] = 0; // Initialize the result cell

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

C[r][j] += A[r][k] \* B[k][j]; // Perform multiplication

}

}

return NULL;

}

int main() {

pthread\_t threads[SIZE];

int row\_indices[SIZE]; // Array to hold row indices

// Create threads for each row

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

row\_indices[i] = i; // Assign row index

pthread\_create(&threads[i], NULL, multiply\_row, &row\_indices[i]); // Create thread

}

// Wait for all threads to complete

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

pthread\_join(threads[i], NULL); // Wait for thread completion

}

// Print the result matrix

printf("Result Matrix C:\n");

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

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

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

}

printf("\n");

}

return 0;

}

gcc matrix.c -lpthread

./a.out

Experiment No 4

AIM: Thread synchronization using counting semaphores and mutual exclusion using mutex.

OBJECTIVE: Implement C program to demonstrate producer-consumer problem with counting semaphores and mutex.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5 // Size of the buffer

int buffer[BUFFER\_SIZE]; // Buffer

int in = 0; // Index for the producer

int out = 0; // Index for the consumer

// Semaphores and mutex

sem\_t empty; // Count of empty slots

sem\_t full; // Count of full slots

pthread\_mutex\_t mutex; // Mutex for mutual exclusion

// Producer function

void\* producer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&empty); // Wait for an empty slot

pthread\_mutex\_lock(&mutex); // Enter critical section

buffer[in] = i; // Produce an item

printf("Produced: %d\n", buffer[in]);

in = (in + 1) % BUFFER\_SIZE; // Update index

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Signal that a new item is produced

sleep(1); // Simulate time taken to produce

}

return NULL;

}

// Consumer function

void\* consumer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&full); // Wait for a full slot

pthread\_mutex\_lock(&mutex); // Enter critical section

int item = buffer[out]; // Consume an item

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

out = (out + 1) % BUFFER\_SIZE; // Update index

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&empty); // Signal that a new slot is empty

sleep(1); // Simulate time taken to consume

}

return NULL;

}

int main() {

pthread\_t prod, cons;

// Initialize semaphores and mutex

sem\_init(&empty, 0, BUFFER\_SIZE); // All slots are initially empty

sem\_init(&full, 0, 0); // No slots are initially full

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

// Create producer and consumer threads

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

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

// Wait for threads to finish

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

// Clean up

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

gcc producer\_consumer.c -lpthread

./a.out

Experiment No 5

To implement Dining Philosopher’s problem using ‘C’ in Linux

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5 // Number of philosophers

#define EAT\_TIME 2 // Time to eat

#define THINK\_TIME 1 // Time to think

sem\_t chopsticks[NUM\_PHILOSOPHERS]; // Semaphores representing chopsticks

// Philosopher function

void\* philosopher(void\* num) {

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

while (1) {

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

sleep(THINK\_TIME); // Simulate thinking

// Pick up chopsticks (left and right)

sem\_wait(&chopsticks[id]); // Pick up left chopstick

sem\_wait(&chopsticks[(id + 1) % NUM\_PHILOSOPHERS]); // Pick up right chopstick

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

sleep(EAT\_TIME); // Simulate eating

// Put down chopsticks

sem\_post(&chopsticks[id]); // Put down left chopstick

sem\_post(&chopsticks[(id + 1) % NUM\_PHILOSOPHERS]); // Put down right chopstick

}

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize semaphores

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_init(&chopsticks[i], 0, 1); // Each chopstick is available (1)

philosopher\_ids[i] = i; // Assign an ID to each philosopher

}

// Create philosopher threads

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

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

}

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

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

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

}

// Clean up (not reached in this example)

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_destroy(&chopsticks[i]);

}

return 0;

}

gcc dining\_philosopher.c -lpthread

./a.out

Experiment No 6

Inter process communication in Linux using pipes.

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#define SIZE 1024

int main() {

int pipe1[2], pipe2[2];

char path[SIZE], buffer[SIZE];

pipe(pipe1); // Pipe for parent-to-child

pipe(pipe2); // Pipe for child-to-parent

if (fork() == 0) { // Child process

close(pipe1[1]); // Close write end of pipe1

read(pipe1[0], path, SIZE); // Read pathname from parent

int fd = open(path, O\_RDONLY); // Open file

int bytes = read(fd, buffer, SIZE); // Read file contents

close(fd);

close(pipe2[0]); // Close read end of pipe2

write(pipe2[1], buffer, bytes); // Send file contents to parent

} else { // Parent process

close(pipe1[0]); // Close read end of pipe1

printf("Enter file path: ");

fgets(path, SIZE, stdin); // Get file path from user

write(pipe1[1], path, SIZE); // Send pathname to child

close(pipe2[1]); // Close write end of pipe2

read(pipe2[0], buffer, SIZE); // Read file contents from child

printf("File Contents:\n%s", buffer); // Print file contents

}

return 0;

}

//gcc -o ipc\_pipes ipc\_pipes.c

//./ipc\_pipes

Experiment no 7

AIM: Inter process communication using system application to demonstrate client and server programs in which server process creates a shared memory segment and write message to the shared memory segment.

server.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#include <unistd.h>

#define SHM\_SIZE 1024 // Size of shared memory segment

int main() {

key\_t key = ftok("shmfile", 65); // Generate unique key

int shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT); // Create shared memory

char \*str = (char \*)shmat(shmid, (void \*)0, 0); // Attach to shared memory

printf("Server: Listening for messages...\n");

while (1) {

// Check for new messages every second

sleep(1);

if (strlen(str) > 0) { // If there's a message

printf("Server: Message received: %s\n", str);

memset(str, 0, SHM\_SIZE); // Clear the message after displaying

}

}

shmdt(str); // Detach from shared memory

return 0;

}

client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#define SHM\_SIZE 1024 // Size of shared memory segment

int main() {

key\_t key = ftok("shmfile", 65); // Generate unique key

int shmid = shmget(key, SHM\_SIZE, 0666); // Access shared memory

char \*str = (char \*)shmat(shmid, (void \*)0, 0); // Attach to shared memory

char input[SHM\_SIZE];

while (1) {

printf("Client: Enter message: ");

fgets(input, SHM\_SIZE, stdin); // Read input from user

input[strcspn(input, "\n")] = 0; // Remove newline character

strcpy(str, input); // Write the message to shared memory

}

shmdt(str); // Detach from shared memory

return 0;

}