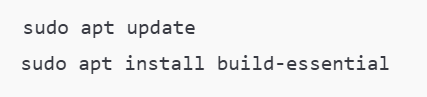
****

**Q1: In a cafeteria, 4 chefs share a cooking station with 3 ovens. Each chef needs to use an oven to prepare their dish, and only one chef can use an oven at a time. If all ovens are occupied, chefs must wait for one to become available. Considering this as a process synchronization problem, write a program to solve it using semaphore concepts.**

**Original Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_CHEFS 4

#define NUM\_OVENS 3

// Semaphore to control access to the ovens (only 3 ovens available)

sem\_t oven\_semaphore;

// Function for chef's work (using an oven)

void\* chef\_work(void\* chef\_id) {

int chef\_number = \*((int\*)chef\_id);

// Wait for an available oven (semaphore decrement)

printf("Chef %d is waiting for an oven...\n", chef\_number);

sem\_wait(&oven\_semaphore);

// Use the oven

printf("Chef %d is using an oven. Available ovens: %d\n", chef\_number, NUM\_OVENS - sem\_getvalue(&oven\_semaphore, NULL));

// Simulate cooking (using sleep)

sleep(2); // Simulate cooking time

// Done cooking, release the oven (semaphore increment)

sem\_post(&oven\_semaphore);

printf("Chef %d is done cooking. Available ovens: %d\n", chef\_number, NUM\_OVENS - sem\_getvalue(&oven\_semaphore, NULL));

return NULL;

}

int main() {

pthread\_t chefs[NUM\_CHEFS];

int chef\_numbers[NUM\_CHEFS];

// Initialize semaphore for oven access (3 ovens available)

sem\_init(&oven\_semaphore, 0, NUM\_OVENS);

// Create chefs (threads)

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

chef\_numbers[i] = i + 1; // Chef numbers start from 1 to 4

pthread\_create(&chefs[i], NULL, chef\_work, &chef\_numbers[i]);

}

// Wait for all chefs to finish their cooking

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

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

}

// Destroy the semaphore

sem\_destroy(&oven\_semaphore);

return 0;

}

**Updated Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_CHEFS 4

#define NUM\_OVENS 3

// Semaphore to control access to the ovens (only 3 ovens available)

sem\_t oven\_semaphore;

// Function for chef's work (using an oven)

void\* chef\_work(void\* chef\_id) {

int chef\_number = \*((int\*)chef\_id);

int available\_ovens;

// Wait for an available oven (semaphore decrement)

printf("Chef %d is waiting for an oven...\n", chef\_number);

sem\_wait(&oven\_semaphore);

// Get the current number of available ovens

sem\_getvalue(&oven\_semaphore, &available\_ovens);

printf("Chef %d is using an oven. Available ovens: %d\n", chef\_number, available\_ovens);

// Simulate cooking (using sleep)

sleep(2); // Simulate cooking time

// Done cooking, release the oven (semaphore increment)

sem\_post(&oven\_semaphore);

// Get the updated number of available ovens

sem\_getvalue(&oven\_semaphore, &available\_ovens);

printf("Chef %d is done cooking. Available ovens: %d\n", chef\_number, available\_ovens);

return NULL;

}

int main() {

pthread\_t chefs[NUM\_CHEFS];

int chef\_numbers[NUM\_CHEFS];

// Initialize semaphore for oven access (3 ovens available)

sem\_init(&oven\_semaphore, 0, NUM\_OVENS);

// Create chefs (threads)

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

chef\_numbers[i] = i + 1; // Chef numbers start from 1 to 4

pthread\_create(&chefs[i], NULL, chef\_work, &chef\_numbers[i]);

}

// Wait for all chefs to finish their cooking

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

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

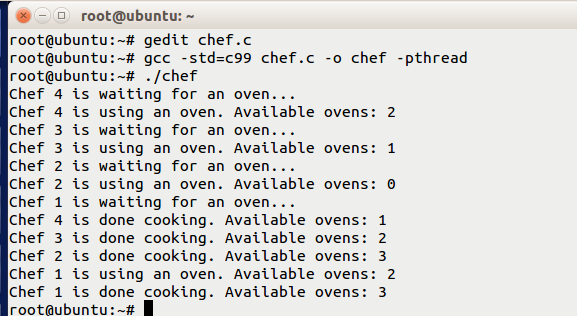
}

// Destroy the semaphore

sem\_destroy(&oven\_semaphore);

return 0;

}



**Q2: A railway ticket booking system has multiple counters and limited seats. The system supports 5 counters and 50 seats. When customers arrive at a counter, they can book a seat if available. If all seats are booked, the customer has to wait for a seat to be freed. Implement this scenario using semaphores to synchronize the access to the seat booking process.**

**Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_COUNTERS 5

#define NUM\_SEATS 50

sem\_t available\_seats; // Semaphore to control the available seats

// Function for customer booking a seat

void\* customer\_booking(void\* customer\_id) {

int customer\_number = \*((int\*)customer\_id);

// Wait for an available seat

sem\_wait(&available\_seats);

// Book the seat

printf("Customer %d is booking a seat. Remaining seats: %d\n", customer\_number, NUM\_SEATS - sem\_getvalue(&available\_seats, NULL));

// Simulate booking time

sleep(1);

// Release the seat after booking

sem\_post(&available\_seats);

printf("Customer %d has booked a seat. Remaining seats: %d\n", customer\_number, NUM\_SEATS - sem\_getvalue(&available\_seats, NULL));

return NULL;

}

int main() {

pthread\_t customers[NUM\_SEATS];

int customer\_numbers[NUM\_SEATS];

// Initialize semaphore for seat availability

sem\_init(&available\_seats, 0, NUM\_SEATS);

// Create customer threads

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

customer\_numbers[i] = i + 1;

pthread\_create(&customers[i], NULL, customer\_booking, &customer\_numbers[i]);

}

// Wait for all customer threads to complete

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

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

}

// Destroy the semaphore

sem\_destroy(&available\_seats);

return 0;

}

**Updated Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_COUNTERS 5

#define NUM\_SEATS 50

sem\_t available\_seats; // Semaphore to control the available seats

// Function for customer booking a seat

void\* customer\_booking(void\* customer\_id) {

int customer\_number = \*((int\*)customer\_id);

// Wait for an available seat

sem\_wait(&available\_seats);

// Book the seat

printf("Customer %d is booking a seat. Remaining seats: %d\n", customer\_number, NUM\_SEATS - sem\_getvalue(&available\_seats, NULL));

// Simulate booking time

sleep(1);

// Release the seat after booking

sem\_post(&available\_seats);

printf("Customer %d has booked a seat. Remaining seats: %d\n", customer\_number, NUM\_SEATS - sem\_getvalue(&available\_seats, NULL));

return NULL;

}

int main() {

pthread\_t customers[NUM\_SEATS];

int customer\_numbers[NUM\_SEATS];

// Initialize semaphore for seat availability

sem\_init(&available\_seats, 0, NUM\_SEATS);

// Create customer threads

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

customer\_numbers[i] = i + 1;

pthread\_create(&customers[i], NULL, customer\_booking, &customer\_numbers[i]);

}

// Wait for all customer threads to complete

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

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

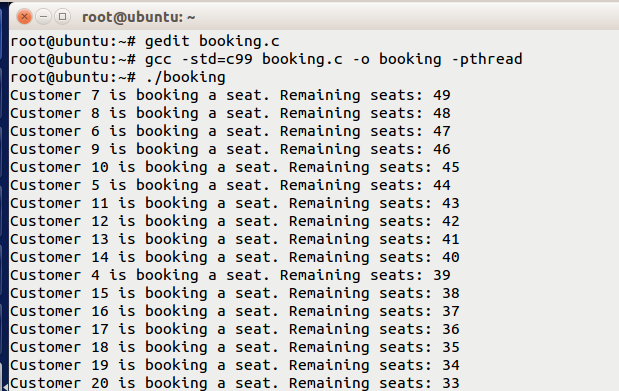
}

// Destroy the semaphore

sem\_destroy(&available\_seats);

return 0;

}

****

**Q3: A customer service desk has 3 counters, each capable of handling only one customer at a time. If all counters are occupied, customers must wait. Implement this scenario using semaphores to synchronize access to the service counters.**

**Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_COUNTERS 3

#define NUM\_CUSTOMERS 10

sem\_t available\_counters; // Semaphore for controlling access to counters

// Function for customers to get served at a counter

void\* customer\_service(void\* customer\_id) {

int customer\_number = \*((int\*)customer\_id);

// Wait for an available counter

sem\_wait(&available\_counters);

// Customer being served

printf("Customer %d is being served at a counter.\n", customer\_number);

// Simulate service time

sleep(2);

// Release the counter after service

sem\_post(&available\_counters);

printf("Customer %d has been served. Counter is available.\n", customer\_number);

return NULL;

}

int main() {

pthread\_t customers[NUM\_CUSTOMERS];

int customer\_numbers[NUM\_CUSTOMERS];

// Initialize semaphore for available counters (3 counters)

sem\_init(&available\_counters, 0, NUM\_COUNTERS);

// Create customer threads

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

customer\_numbers[i] = i + 1;

pthread\_create(&customers[i], NULL, customer\_service, &customer\_numbers[i]);

}

// Wait for all customer threads to complete

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

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

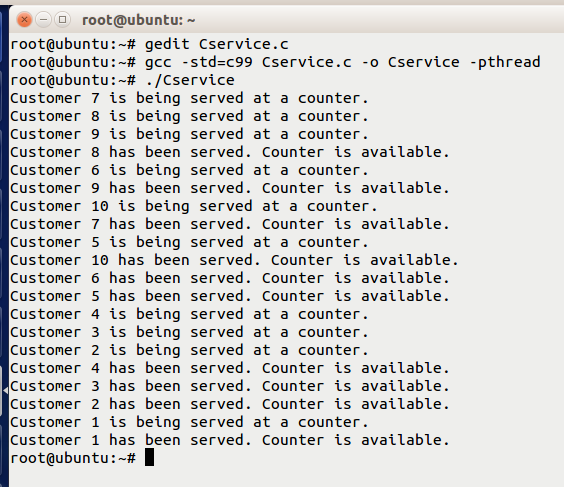
}

// Destroy the semaphore

sem\_destroy(&available\_counters);

return 0;

}



**Q4: In a scientific visualization application, a programmer is tasked with creating a multithreaded program to compute the area and perimeter of a rectangle. The program needs to improve efficiency by performing these calculations concurrently in separate threads. The rectangle's dimensions are given by length and breadth, and the formulas for calculation are as follows:**

**Area = length × breadth**

**Perimeter = 2 × (length + breadth)**

**Write a multithreaded program where one thread accepts length and breadth values from the user and passes them to two other threads. One thread calculates the area, and the other calculates the perimeter. All three threads should run in parallel.**

**Code:**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

typedef struct {

float length;

float breadth;

} Rectangle;

void\* inputDimensions(void\* arg) {

Rectangle\* rect = (Rectangle\*)arg;

printf("Enter the length of the rectangle: ");

scanf("%f", &rect->length);

printf("Enter the breadth of the rectangle: ");

scanf("%f", &rect->breadth);

return NULL;

}

void\* calculateArea(void\* arg) {

Rectangle\* rect = (Rectangle\*)arg;

float area = rect->length \* rect->breadth;

printf("Area of the rectangle: %.2f\n", area);

return NULL;

}

void\* calculatePerimeter(void\* arg) {

Rectangle\* rect = (Rectangle\*)arg;

float perimeter = 2 \* (rect->length + rect->breadth);

printf("Perimeter of the rectangle: %.2f\n", perimeter);

return NULL;

}

int main() {

pthread\_t inputThread, areaThread, perimeterThread;

Rectangle rect;

pthread\_create(&inputThread, NULL, inputDimensions, &rect);

// Wait for the input thread to finish

pthread\_join(inputThread, NULL);

pthread\_create(&areaThread, NULL, calculateArea, &rect);

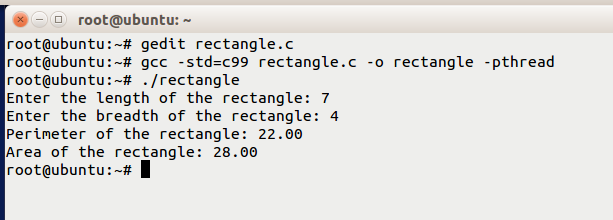
pthread\_create(&perimeterThread, NULL, calculatePerimeter, &rect);

pthread\_join(areaThread, NULL);

pthread\_join(perimeterThread, NULL);

return 0;

}



**Q5: A resource management system has four processes (P0, P1, P2, P3) competing for resources X, Y, and Z. Each process has allocated and maximum resource requirements, and the system has a certain number of available instances for each resource. The objective is to determine if the system is in a safe state and implement a deadlock safety program using the Banker's algorithm.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Allocation** | | | **Max** | | |
|  | **X** | **Y** | **Z** | **X** | **Y** | **Z** |
| **P0** | **1** | **0** | **1** | **2** | **1** | **2** |
| **P1** | **0** | **2** | **1** | **1** | **3** | **1** |
| **P2** | **1** | **3** | **2** | **1** | **4** | **3** |
| **P3** | **0** | **0** | **2** | **1** | **2** | **4** |

**Processes = 4 (P0, P1, P2, P3)**

**X = 3 Instances**

**Y = 5 Instances**

**Z = 6 Instances**

**Consider the above resource and process allocation.**

**a) Write a code that determines if the system is in a safe state.**

**b) Implement a deadlock safety program for the system.**

**Code:**

#include <stdio.h>

#define PROCESSES 4

#define RESOURCES 3

int isSafeState(int available[], int max[][RESOURCES], int allocation[][RESOURCES], int need[][RESOURCES]) {

int finish[PROCESSES] = {0}; // Tracks finished processes (0 = false, 1 = true)

int work[RESOURCES]; // Work array to track available resources during simulation

int safeSequence[PROCESSES]; // To store the safe sequence

int count = 0; // Number of finished processes

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

work[i] = available[i];

}

while (count < PROCESSES) {

int found = 0; // Whether a process is found to proceed (0 = false, 1 = true)

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

if (!finish[p]) {

int canProceed = 1; // Assume the process can proceed

for (int r = 0; r < RESOURCES; r++) {

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

canProceed = 0; // Cannot proceed

break;

}

}

if (canProceed) {

for (int r = 0; r < RESOURCES; r++) {

work[r] += allocation[p][r];

}

safeSequence[count++] = p; // Add process to safe sequence

finish[p] = 1; // Mark process as finished

found = 1;

}

}

}

if (!found) {

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

return 0;

}

}

printf("The system is in a safe state.\nSafe Sequence: ");

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

printf("P%d ", safeSequence[i]);

}

printf("\n");

return 1;

}

int main() {

int allocation[PROCESSES][RESOURCES] = {

{1, 0, 1},

{0, 2, 1},

{1, 3, 2},

{0, 0, 2}

};

int max[PROCESSES][RESOURCES] = {

{2, 1, 2},

{1, 3, 1},

{1, 4, 3},

{1, 2, 4}

};

int totalInstances[RESOURCES] = {3, 5, 6}; // X, Y, Z total instances

int available[RESOURCES];

for (int r = 0; r < RESOURCES; r++) {

int allocatedSum = 0;

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

allocatedSum += allocation[p][r];

}

available[r] = totalInstances[r] - allocatedSum;

}

int need[PROCESSES][RESOURCES];

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

for (int r = 0; r < RESOURCES; r++) {

need[p][r] = max[p][r] - allocation[p][r];

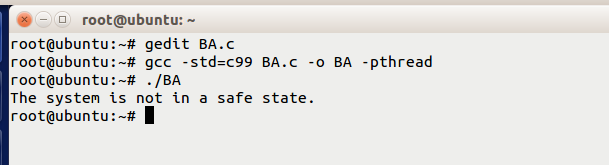
}

}

isSafeState(available, max, allocation, need);

return 0;

}



**Q6:Consider the following page reference string and write a program to calculate the maximum page hits and faults using the FIFO (First-In, First-Out) page replacement technique. Assume a maximum page size of 3.**

**Reference string: 3, 5, 2, 6, 1, 3, 5, 4, 7, 8, 2, 5, 3, 6, 1, 4, 3, 2, 5**

**Code:**

#include <stdio.h>

#define MAX\_FRAMES 3 // Maximum number of page frames

#define MAX\_REFERENCES 19 // Number of references in the string

// Function to check if a page is present in the frame

int isPageInFrame(int frames[], int numFrames, int page) {

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

if (frames[i] == page) {

return 1; // Page found in the frame

}

}

return 0; // Page not found in the frame

}

int main() {

int referenceString[MAX\_REFERENCES] = {3, 5, 2, 6, 1, 3, 5, 4, 7, 8, 2, 5, 3, 6, 1, 4, 3, 2, 5};

int frames[MAX\_FRAMES] = {-1, -1, -1}; // Initialize frames with -1 (empty)

int numFrames = MAX\_FRAMES;

int numReferences = MAX\_REFERENCES;

int pageFaults = 0;

int pageHits = 0;

int currentFrameIndex = 0; // Pointer for FIFO replacement

printf("Reference String: ");

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

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

}

printf("\n\n");

printf("Frame Status:\n");

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

int currentPage = referenceString[i];

// Check if the page is already in the frame

if (isPageInFrame(frames, numFrames, currentPage)) {

pageHits++; // Page hit

} else {

// Page fault occurs; replace the oldest page in FIFO order

frames[currentFrameIndex] = currentPage;

currentFrameIndex = (currentFrameIndex + 1) % numFrames; // Move to the next frame

pageFaults++; // Increment page faults

}

// Print the current state of the frames

printf("After reference %d: ", currentPage);

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

if (frames[j] == -1) {

printf(" - ");

} else {

printf("%d ", frames[j]);

}

}

printf("\n");

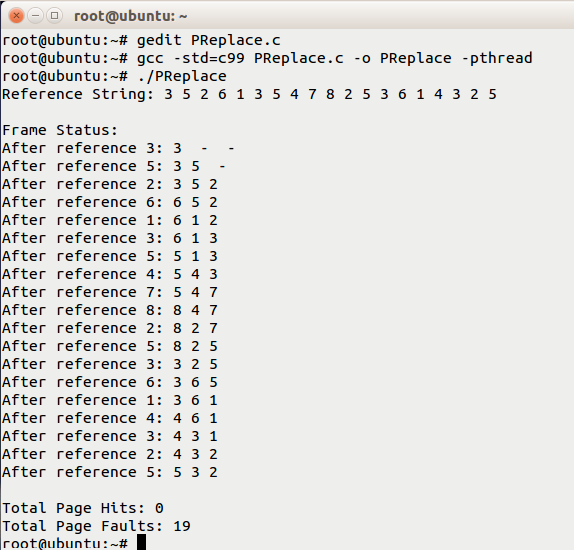
}

printf("\nTotal Page Hits: %d\n", pageHits);

printf("Total Page Faults: %d\n", pageFaults);

return 0;

}



**Q7: Consider a dining hall with 5 tables and 10 chairs, where each table can accommodate up to 2 students. There are 3 serving trays shared among all tables. If all trays are occupied, some students must wait for their turn. Considering the above problem as process synchronization, write a program to solve the issue using semaphore concepts to avoid conflicts.**

**Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_TABLES 5

#define NUM\_CHAIRS 10

#define NUM\_TRAYS 3

#define NUM\_STUDENTS 12

sem\_t traySemaphore; // Semaphore for trays (3 available)

sem\_t tableSemaphore; // Semaphore for tables (5 available)

pthread\_mutex\_t printMutex; // Mutex for synchronized printing

void\* student(void\* arg) {

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

pthread\_mutex\_lock(&printMutex);

printf("Student %d has arrived at the dining hall.\n", studentID);

pthread\_mutex\_unlock(&printMutex);

// Wait for a table

sem\_wait(&tableSemaphore);

pthread\_mutex\_lock(&printMutex);

printf("Student %d occupies a table.\n", studentID);

pthread\_mutex\_unlock(&printMutex);

// Wait for a tray

sem\_wait(&traySemaphore);

pthread\_mutex\_lock(&printMutex);

printf("Student %d picks up a tray.\n", studentID);

pthread\_mutex\_unlock(&printMutex);

// Simulate the time spent eating

sleep(1 + (rand() % 3)); // Random eating time between 1-3 seconds

// Release the tray

sem\_post(&traySemaphore);

pthread\_mutex\_lock(&printMutex);

printf("Student %d returns their tray.\n", studentID);

pthread\_mutex\_unlock(&printMutex);

// Leave the table

sem\_post(&tableSemaphore);

pthread\_mutex\_lock(&printMutex);

printf("Student %d leaves the dining hall.\n", studentID);

pthread\_mutex\_unlock(&printMutex);

return NULL;

}

int main() {

pthread\_t students[NUM\_STUDENTS];

int studentIDs[NUM\_STUDENTS];

// Initialize semaphores

sem\_init(&traySemaphore, 0, NUM\_TRAYS);

sem\_init(&tableSemaphore, 0, NUM\_TABLES);

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

// Create student threads

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

studentIDs[i] = i + 1;

pthread\_create(&students[i], NULL, student, &studentIDs[i]);

}

// Wait for all students to finish

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

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

}

// Destroy semaphores and mutex

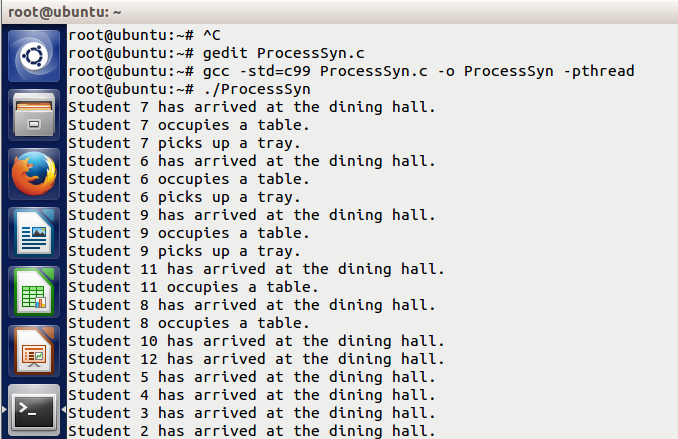
sem\_destroy(&traySemaphore);

sem\_destroy(&tableSemaphore);

pthread\_mutex\_destroy(&printMutex);

return 0;

}



**Q8: In a financial analytics application, a programmer is tasked with developing a multithreaded program to calculate the monthly interest and total payable amount for a loan. The calculations should be performed concurrently in separate threads. The loan details include the principal amount, annual interest rate, and duration in months. The formulas are as follows:**

**Monthly Interest = (Principal \* Annual Interest Rate) / 12**

**Total Payable = Principal + (Monthly Interest \* Duration)**

**Write a multithreaded program where one thread takes the loan details (Principal, Rate, Duration) from the user and passes them to two other threads. One thread calculates the monthly interest, and the other calculates the total payable amount. All three threads should run in parallel.**

**Code:**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

// Struct to hold loan details

typedef struct {

double principal;

double annualInterestRate;

int duration; // in months

double monthlyInterest;

double totalPayable;

} LoanDetails;

// Function to take input for loan details

void\* take\_input(void\* arg) {

LoanDetails\* loan = (LoanDetails\*)arg;

printf("Enter the principal amount: ");

scanf("%lf", &loan->principal);

printf("Enter the annual interest rate (in %%): ");

scanf("%lf", &loan->annualInterestRate);

printf("Enter the loan duration (in months): ");

scanf("%d", &loan->duration);

return NULL;

}

// Function to calculate monthly interest

void\* calculate\_monthly\_interest(void\* arg) {

LoanDetails\* loan = (LoanDetails\*)arg;

// Calculate monthly interest

loan->monthlyInterest = (loan->principal \* loan->annualInterestRate) / (12 \* 100);

return NULL;

}

// Function to calculate total payable amount

void\* calculate\_total\_payable(void\* arg) {

LoanDetails\* loan = (LoanDetails\*)arg;

// Calculate total payable amount

loan->totalPayable = loan->principal + (loan->monthlyInterest \* loan->duration);

return NULL;

}

int main() {

LoanDetails loan = {0};

pthread\_t inputThread, interestThread, payableThread;

// Create thread for taking input

pthread\_create(&inputThread, NULL, take\_input, &loan);

// Wait for input thread to finish

pthread\_join(inputThread, NULL);

// Create threads for calculations

pthread\_create(&interestThread, NULL, calculate\_monthly\_interest, &loan);

pthread\_create(&payableThread, NULL, calculate\_total\_payable, &loan);

// Wait for calculation threads to finish

pthread\_join(interestThread, NULL);

pthread\_join(payableThread, NULL);

// Print results

printf("\nLoan Details:\n");

printf("Principal: %.2lf\n", loan.principal);

printf("Annual Interest Rate: %.2lf%%\n", loan.annualInterestRate);

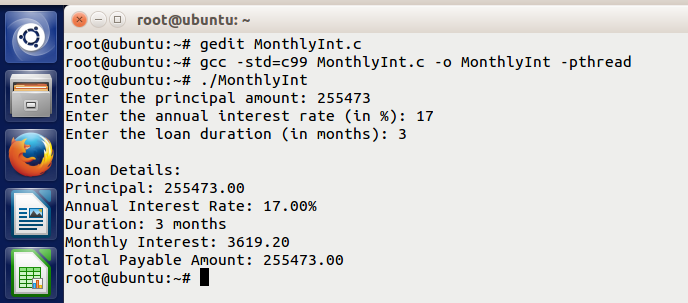
printf("Duration: %d months\n", loan.duration);

printf("Monthly Interest: %.2lf\n", loan.monthlyInterest);

printf("Total Payable Amount: %.2lf\n", loan.totalPayable);

return 0;

}



**Q9: A storage system manages four processes (P0, P1, P2, P3) that require resources R1, R2, and R3. Each process has its allocated and maximum resource requirements, and the system has limited instances of each resource. The objective is to determine if the system is in a safe state and implement a deadlock safety program using the Banker's algorithm.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Allocation** | | | **Max** | | |
|  | **R1** | **R2** | **R3** | **R1** | **R2** | **R3** |
| **P0** | **0** | **1** | **2** | **1** | **2** | **2** |
| **P1** | **1** | **0** | **1** | **2** | **2** | **1** |
| **P2** | **1** | **2** | **3** | **2** | **3** | **3** |
| **P3** | **0** | **1** | **0** | **1** | **1** | **1** |

**Processes = 4 (P0, P1, P2, P3)**

**R1 = 4 Instances**

**R2 = 5 Instances**

**R3 = 6 Instances**

**a) Write a code to check if the system is in a safe state.**

**b) Write a deadlock safety program for the system.**

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define NUM\_PROCESSES 4

#define NUM\_RESOURCES 3

// Function to calculate the Need matrix (Max - Allocation)

void calculate\_need(int max[NUM\_PROCESSES][NUM\_RESOURCES], int allocation[NUM\_PROCESSES][NUM\_RESOURCES], int need[NUM\_PROCESSES][NUM\_RESOURCES]) {

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

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

}

}

}

// Function to check if the system is in a safe state using Banker's algorithm

bool is\_safe(int available[NUM\_RESOURCES], int allocation[NUM\_PROCESSES][NUM\_RESOURCES], int max[NUM\_PROCESSES][NUM\_RESOURCES]) {

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

calculate\_need(max, allocation, need);

bool finish[NUM\_PROCESSES] = {0}; // Keep track of finished processes

int work[NUM\_RESOURCES];

// Initialize work as the available resources

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

work[i] = available[i];

}

// Loop to check if the system can reach a safe state

for (int count = 0; count < NUM\_PROCESSES; count++) {

bool progress\_made = false;

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

if (!finish[i]) {

// Check if the process can finish (Need <= Work)

bool can\_finish = true;

for (int j = 0; j < NUM\_RESOURCES; j++) {

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

can\_finish = false;

break;

}

}

// If process can finish, update work and finish

if (can\_finish) {

for (int j = 0; j < NUM\_RESOURCES; j++) {

work[j] += allocation[i][j]; // Add allocated resources to work

}

finish[i] = true;

progress\_made = true;

break;

}

}

}

// If no progress was made, then the system is not in a safe state

if (!progress\_made) {

return false;

}

}

return true; // If all processes can finish, system is in a safe state

}

int main() {

// Allocation Matrix

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {

{0, 1, 2}, // P0

{1, 0, 1}, // P1

{1, 2, 3}, // P2

{0, 1, 0} // P3

};

// Maximum Matrix

int max[NUM\_PROCESSES][NUM\_RESOURCES] = {

{1, 2, 2}, // P0

{2, 2, 1}, // P1

{2, 3, 3}, // P2

{1, 1, 1} // P3

};

// Available Resources

int available[NUM\_RESOURCES] = {4, 5, 6}; // R1 = 4, R2 = 5, R3 = 6

// Check if the system is in a safe state

if (is\_safe(available, allocation, max)) {

printf("The system is in a safe state.\n");

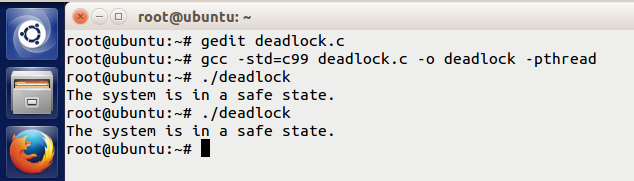
} else {

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

}

return 0;

}



**Q10:**

**Consider the following page reference string and write a program to calculate the maximum page hits and faults using the FIFO (First-In, First-Out) page replacement technique. Assume a maximum page size of 3.**

**Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0**

**Code:**

#include <stdio.h>

#define FRAME\_SIZE 3

// Function to simulate the FIFO page replacement algorithm

void fifo\_page\_replacement(int reference\_string[], int length) {

int frame[FRAME\_SIZE] = {-1, -1, -1}; // Initialize page frames to -1 (empty)

int page\_faults = 0, page\_hits = 0;

// Iterate over the reference string

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

int page = reference\_string[i];

int found = 0;

// Check if the page is already in the frame (page hit)

for (int j = 0; j < FRAME\_SIZE; j++) {

if (frame[j] == page) {

found = 1;

page\_hits++;

break;

}

}

// If the page is not found in the frame (page fault)

if (!found) {

page\_faults++;

// Find the first empty slot (if any)

int replaced = 0;

for (int j = 0; j < FRAME\_SIZE; j++) {

if (frame[j] == -1) {

frame[j] = page;

replaced = 1;

break;

}

}

// If no empty slot, replace the oldest page (FIFO)

if (!replaced) {

for (int j = 0; j < FRAME\_SIZE - 1; j++) {

frame[j] = frame[j + 1]; // Shift pages left

}

frame[FRAME\_SIZE - 1] = page; // Add the new page at the end

}

}

// Print the current state of the frame after each reference

printf("Reference: %d | Frame: [", page);

for (int j = 0; j < FRAME\_SIZE; j++) {

if (frame[j] != -1) {

printf("%d", frame[j]);

if (j < FRAME\_SIZE - 1 && frame[j + 1] != -1) {

printf(", ");

}

}

}

printf("]\n");

}

// Print the total number of page hits and page faults

printf("\nTotal page hits: %d\n", page\_hits);

printf("Total page faults: %d\n", page\_faults);

}

int main() {

// Page reference string

int reference\_string[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0};

int length = sizeof(reference\_string) / sizeof(reference\_string[0]);

// Simulate FIFO page replacement

fifo\_page\_replacement(reference\_string, length);

return 0;

}

