**CSA0497 - OPERATING SYSTEMS FOR MOBILE**

**APPLICATIONS**

**LAB PROGRAM 2 OUTPUTS**

**~ K. Parameswari**

**192372266**

**EX.1: Round Robin Scheduling**

**Code:**

#include <stdio.h>

void round\_robin(int processes[], int n, int burst\_time[], int time\_quantum) {

int remaining\_time[n];

int waiting\_time[n];

int turnaround\_time[n];

// Initialize remaining time and waiting time

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

remaining\_time[i] = burst\_time[i];

waiting\_time[i] = 0;

}

int time = 0;

int completed = 0;

while (completed < n) {

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

if (remaining\_time[i] > 0) {

if (remaining\_time[i] > time\_quantum) {

time += time\_quantum;

remaining\_time[i] -= time\_quantum;

} else {

time += remaining\_time[i];

waiting\_time[i] = time - burst\_time[i]; // Calculate waiting time

turnaround\_time[i] = time; // Calculate turnaround time

printf("Process %d completed at time %d\n", processes[i], time);

remaining\_time[i] = 0;

completed++;

}

}

}

}

// Calculate average waiting time and turnaround time

float total\_waiting\_time = 0;

float total\_turnaround\_time = 0;

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

total\_waiting\_time += waiting\_time[i];

total\_turnaround\_time += turnaround\_time[i];

}

printf("\nAverage Waiting Time: %.2f\n", total\_waiting\_time / n);

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

}

int main() {

int processes [] = {1, 2, 3}; // Process IDs

int burst\_time[] = {10, 5, 8}; // Burst times

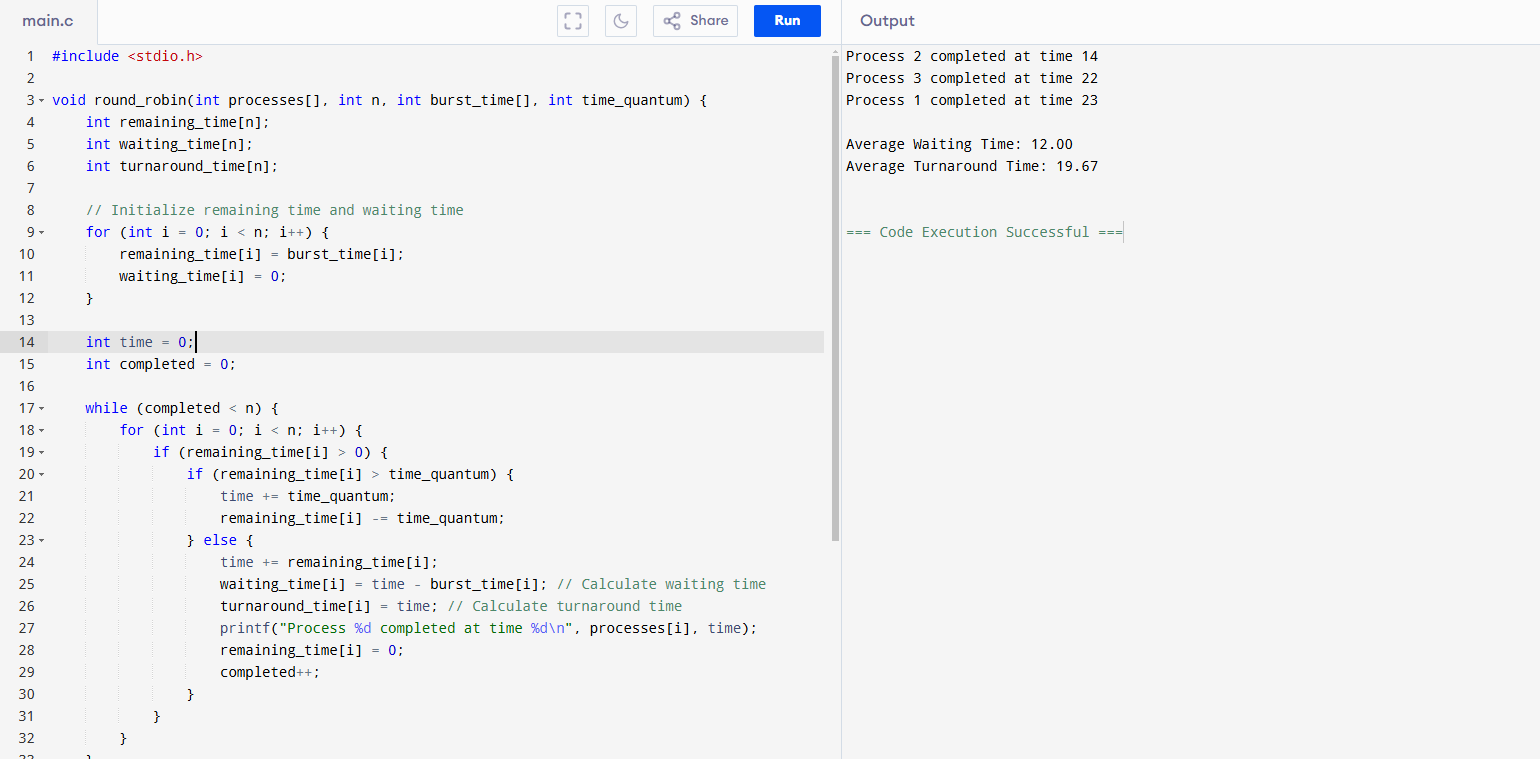
int n = sizeof(processes) / sizeof(processes [0]);

int time\_quantum = 3;

round\_robin(processes, n, burst\_time, time\_quantum);

return 0;

}



**EX.2: INTER- PROCESS COMMUNICATION**

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main() {

int pipefd[2]; // Array to hold the read and write file descriptors

pid\_t pid;

char buffer[100];

// Create a pipe

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

perror("pipe");

exit(EXIT\_FAILURE);

}

// Fork a child process

pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) { // Child process (Receiver)

close(pipefd[1]); // Close the write end of the pipe

read(pipefd[0], buffer, sizeof(buffer)); // Read from the pipe

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

close(pipefd[0]); // Close the read end of the pipe

} else { // Parent process (Sender)

close(pipefd[0]); // Close the read end of the pipe

const char \*message = "Hello from Sender (Parent)!";

write(pipefd[1], message, strlen(message) + 1); // Write to the pipe

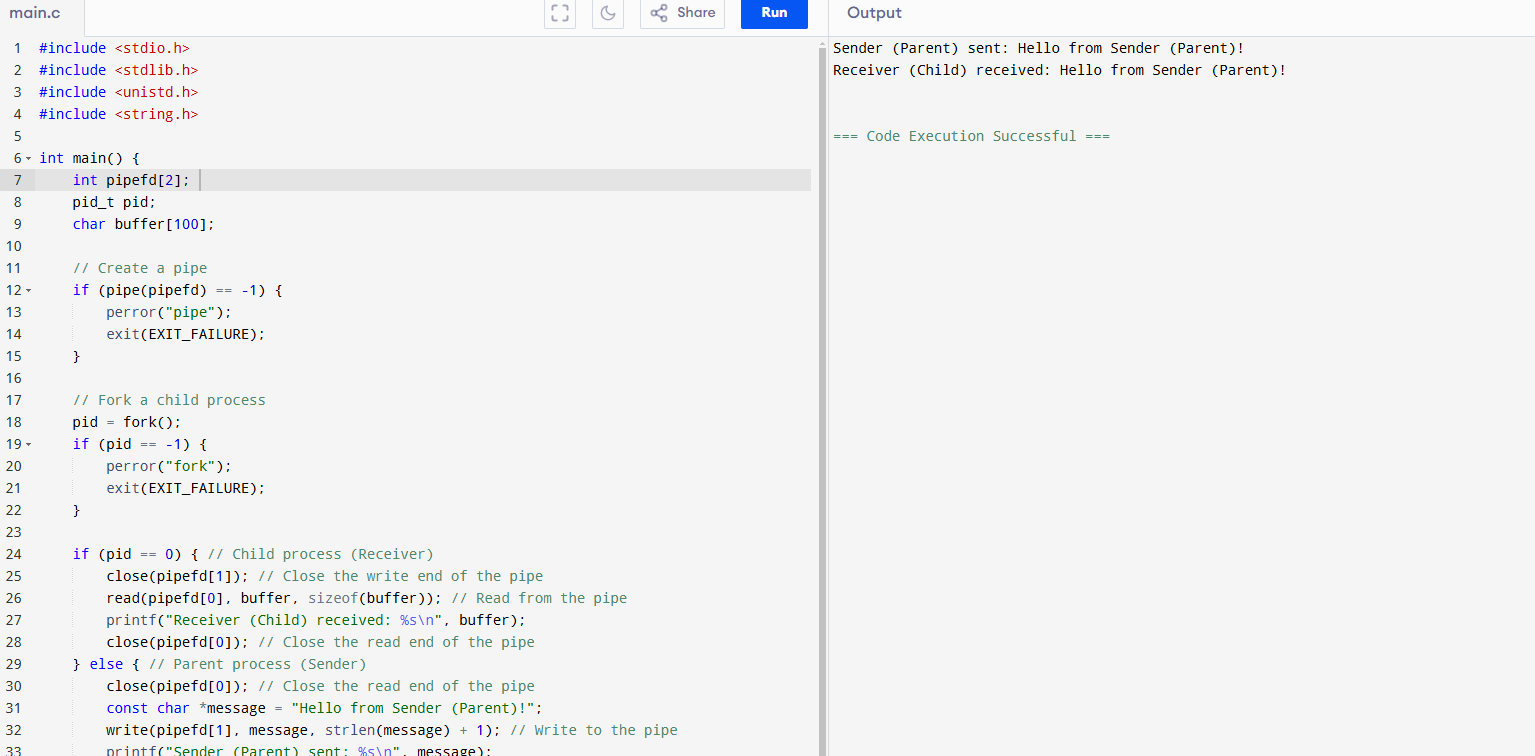
printf("Sender (Parent) sent: %s\n", message);

close(pipefd[1]); // Close the write end of the pipe

}

return 0;

}



**EX.3: DINING-PHILOSOPHERS PROBLEM**

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t forks[NUM\_PHILOSOPHERS];

void\* philosopher(void\* num) {

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

while (1) {

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

sleep(1); // Simulate thinking time

// Pick up forks

pthread\_mutex\_lock(&forks[id]); // Pick up left fork

pthread\_mutex\_lock(&forks[(id + 1) % NUM\_PHILOSOPHERS]); // Pick up right fork

// Eating

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

sleep(1); // Simulate eating time

// Put down forks

pthread\_mutex\_unlock(&forks[(id + 1) % NUM\_PHILOSOPHERS]); // Put down right fork

pthread\_mutex\_unlock(&forks[id]); // Put down left fork

}

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize mutexes for forks

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

pthread\_mutex\_init(&forks[i], NULL);

philosopher\_ids[i] = i;

}

// 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 example)

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

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

}

// Destroy mutexes

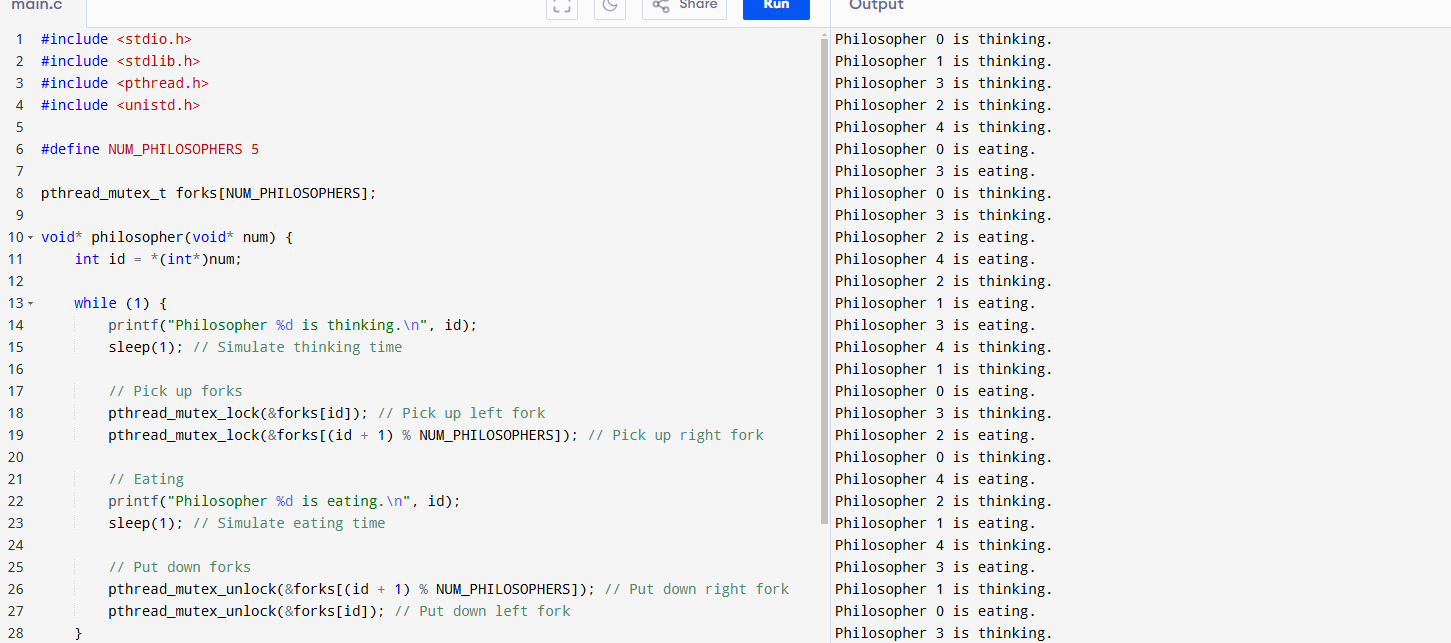
for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_mutex\_destroy(&forks[i]);

}

return 0;

}



**EX.4:BANKER’S ALGORITHM**

**CODE**

#include <stdio.h>

#include <stdbool.h>

#define MAX 10

#define RESOURCES 3

#define PROCESSES 5

int allocation[PROCESSES][RESOURCES] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int max[PROCESSES][RESOURCES] = {

{0, 1, 0},

{2, 0, 2},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int available[RESOURCES] = {3, 3, 2};

int need[PROCESSES][RESOURCES];

void calculateNeed() {

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

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

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

}

}

}

bool isSafe() {

int work[RESOURCES];

bool finish[PROCESSES] = {false};

int safeSequence[PROCESSES];

int count = 0;

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

work[i] = available[i];

}

while (count < PROCESSES) {

bool found = false;

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

if (!finish[p]) {

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] += allocation[p][k];

}

safeSequence[count++] = p;

finish[p] = true;

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 ", safeSequence[i]);

}

printf("\n");

return true;

}

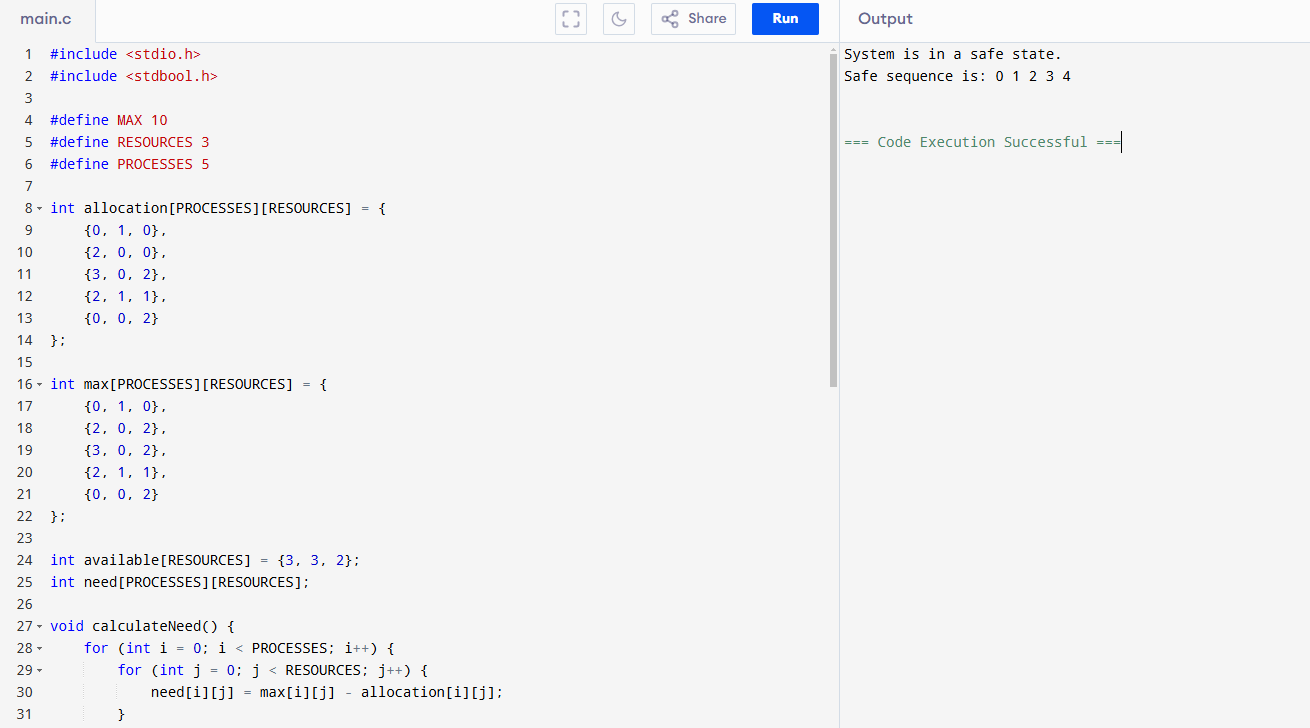
int main() {

calculateNeed();

isSafe();

return 0;

}



**EX.5:PRODUCER CONSUMER PROBLEM**

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0; // Index for the next produced item

int out = 0; // Index for the next consumed item

sem\_t empty; // Semaphore to count empty slots

sem\_t full; // Semaphore to count full slots

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

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; // Move to the next index

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

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

sleep(rand() % 2); // Sleep for a random time

}

return NULL;

}

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; // Move to the next index

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

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

sleep(rand() % 2); // Sleep for a random time

}

return NULL;

}

int main() {

pthread\_t prod, cons;

// Initialize semaphores and mutex

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

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

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

// 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;

}