1)ROUND ROBBIN

#include <stdio.h>

#define MAX\_PROCESSES 100

struct Process {

int id;

int burstTime;

int remainingTime;

int waitingTime;

int turnaroundTime;

};

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

int time = 0;

int finishedProcesses = 0;

while (finishedProcesses < n) {

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

if (processes[i].remainingTime > 0) {

if (processes[i].remainingTime > quantum) {

time += quantum;

processes[i].remainingTime -= quantum;

} else {

time += processes[i].remainingTime;

processes[i].remainingTime = 0;

finishedProcesses++;

processes[i].turnaroundTime = time;

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

}

}

}

}

}

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

int totalWaitingTime = 0, totalTurnaroundTime = 0;

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

float averageWaitingTime = (float) totalWaitingTime / n;

float averageTurnaroundTime = (float) totalTurnaroundTime / n;

printf("Average waiting time = %.2f\n", averageWaitingTime);

printf("Average turnaround time = %.2f\n", averageTurnaroundTime);

}

int main() {

int n, quantum;

struct Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

printf("Enter the quantum time: ");

scanf("%d", &quantum);

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

printf("Enter burst time for process %d: ", i + 1);

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

processes[i].id = i + 1;

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

processes[i].waitingTime = 0;

processes[i].turnaroundTime = 0;

}

roundRobinScheduling(processes, n, quantum);

calculateAverageTimes(processes, n);

return 0;

}

Input:

Enter the number of processes: 3

Enter the quantum time: 4

Enter the burst time for process 1: 10

Enter the burst time for process 2: 5

Enter the burst time for process 3: 8

Output:

Average Waiting time: 12.67

Average Turn around time: 20.33

2)INTER PROCESS COMMUNICATION

PIPES:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main() {

int pipefd[2];

pid\_t pid;

char buffer[100];

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

perror("pipe");

exit(EXIT\_FAILURE);

}

pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

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

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

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

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

close(pipefd[0]);

} else { // Parent process

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

write(pipefd[1], "Hello from parent!", strlen("Hello from parent!") + 1);

close(pipefd[1]);

wait(NULL);

}

return 0;

}

Output:

Child received: Hello from parent!

Message Queues

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key;

int msgid;

struct msg\_buffer message;

key = ftok("progfile", 65);

msgid = msgget(key, 0666 | IPC\_CREAT);

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello from message queue!");

msgsnd(msgid, &message, sizeof(message), 0);

msgrcv(msgid, &message, sizeof(message), 1, 0);

printf("Received message: %s\n", message.msg\_text);

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

Output:

Received message: Hello from message queue!

Shared Memory

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

int main() {

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

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);

char \*str = (char\*) shmat(shmid, (void\*)0, 0);

strcpy(str, "Hello from shared memory!");

printf("Data written to shared memory: %s\n", str);

shmdt(str);

shmctl(shmid, IPC\_RMID, NULL);

return 0;

}

Output:

Data written to shared memory: Hello from shared memory!

Semaphores

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <unistd.h>

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

void semaphore\_wait(int semid) {

struct sembuf sb = {0, -1, 0};

semop(semid, &sb, 1);

}

void semaphore\_signal(int semid) {

struct sembuf sb = {0, 1, 0};

semop(semid, &sb, 1);

}

int main() {

key\_t key = ftok("semfile", 65);

int semid = semget(key, 1, 0666 | IPC\_CREAT);

union semun sem\_union;

sem\_union.val = 1;

semctl(semid, 0, SETVAL, sem\_union);

if (fork() == 0) {

semaphore\_wait(semid);

printf("Child process entered critical section.\n");

sleep(2);

printf("Child process leaving critical section.\n");

semaphore\_signal(semid);

exit(0);

} else {

semaphore\_wait(semid);

printf("Parent process entered critical section.\n");

sleep(2);

printf("Parent process leaving critical section.\n");

semaphore\_signal(semid);

wait(NULL);

semctl(semid, 0, IPC\_RMID);

}

return 0;

}

Output:

Parent process entered critical section.

Child process entered critical section.

Parent process leaving critical section.

Child process leaving critical section.

3)DINING PHILOSPHER’S PROBLEM

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define N 5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define LEFT (phil\_num + N - 1) % N

#define RIGHT (phil\_num + 1) % N

int state[N];

sem\_t mutex;

sem\_t S[N];

void test(int phil\_num) {

if (state[phil\_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {

state[phil\_num] = EATING;

sleep(1);

printf("Philosopher %d takes fork %d and %d\n", phil\_num + 1, LEFT + 1, phil\_num + 1);

printf("Philosopher %d is Eating\n", phil\_num + 1);

sem\_post(&S[phil\_num]);

}

}

void take\_fork(int phil\_num) {

sem\_wait(&mutex);

state[phil\_num] = HUNGRY;

printf("Philosopher %d is Hungry\n", phil\_num + 1);

test(phil\_num);

sem\_post(&mutex);

sem\_wait(&S[phil\_num]);

sleep(1);

}

void put\_fork(int phil\_num) {

sem\_wait(&mutex);

state[phil\_num] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n", phil\_num + 1, LEFT + 1, phil\_num + 1);

printf("Philosopher %d is thinking\n", phil\_num + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num) {

while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main() {

int i;

pthread\_t thread\_id[N];

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

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

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

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

pthread\_create(&thread\_id[i], NULL, philosopher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

}

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

pthread\_join(thread\_id[i], NULL);

}

Output:

Philosopher 1 is Hungry

Philosopher 1 takes fork 5 and 1

Philosopher 1 is Eating

Philosopher 2 is Hungry

Philosopher 5 is Hungry

Philosopher 4 is Hungry

Philosopher 4 takes fork 3 and 4

Philosopher 4 is Eating

Philosopher 1 putting fork 5 and 1 down

...

4)BANKER’S ALGORITHM

#include <stdio.h>

#include <stdbool.h>

#define P 5 // Number of processes

#define R 3 // Number of resources

int available[R] = {3, 3, 2}; // Available instances of resources

int maximum[P][R] = { {7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3} }; // Maximum demand of each process

int allocation[P][R] = { {0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2} }; // Initially allocated resources

int need[P][R]; // Remaining needs of each process

void calculateNeed() {

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

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

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

}

}

}

bool isSafeState() {

int work[R];

bool finish[P] = {0};

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

work[i] = available[i];

}

while (true) {

bool found = false;

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

if (!finish[i]) {

bool canProceed = true;

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

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

canProceed = false;

break;

}

}

if (canProceed) {

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

work[k] += allocation[i][k];

}

finish[i] = true;

found = true;

}

}

}

if (!found) {

break;

}

}

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

if (!finish[i]) {

return false; // Not all processes could finish

}

}

return true;

}

void requestResources(int process, int request[]) {

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

if (request[i] > need[process][i] || request[i] > available[i]) {

printf("Process %d's request cannot be granted.\n", process + 1);

return;

}

}

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

available[i] -= request[i];

allocation[process][i] += request[i];

need[process][i] -= request[i];

}

if (isSafeState()) {

printf("Process %d's request has been granted.\n", process + 1);

} else {

printf("Process %d's request would lead to an unsafe state. Rolling back.\n", process + 1);

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

available[i] += request[i];

allocation[process][i] -= request[i];

need[process][i] += request[i];

}

}

}

int main() {

calculateNeed();

int process, request[R];

printf("Enter the process number (0-%d): ", P - 1);

scanf("%d", &process);

printf("Enter the request for resources: ");

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

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

}

requestResources(process, request);

return 0;

}

INPUT:

Enter the process number (0-4): 1

Enter the request for resources: 1 0 2

OUTPUT:

Process 2's request has been granted.

5) PRODUCER CONSUMER PROBLEM

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void\* producer(void\* arg) {

int item;

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

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

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

sleep(1); // Sleep to simulate production time

}

}

void\* consumer(void\* arg) {

int item;

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

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(2); // Sleep to simulate consumption time

}

}

int main() {

pthread\_t prod, cons;

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

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

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

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

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

OUTPUT:

Producer produced 45

Consumer consumed 45

Producer produced 18

Producer produced 77

Consumer consumed 18

Producer produced 33

Consumer consumed 77

Producer produced 65

Consumer consumed 33

Producer produced 89

Consumer consumed 65

Producer produced 50

Consumer consumed 89

Consumer consumed 50