OS PAT & FAT-LAB SYLLABUS

1. Banker's Algo

#include <stdio.h>

#include <stdbool.h>

#define P 4 // Number of processes

#define R 3 // Number of resources

// Function to check if the system is in a safe state

bool isSafe(int processes[], int avail[], int max[][R], int allot[][R]) {

int work[R], finish[P], safeSeq[P];

int count = 0;

// Initialize work = avail and finish = false for all processes

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

work[i] = avail[i];

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

finish[i] = 0;

while (count < P) {

bool found = false;

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

if (finish[p] == 0) { // If not yet finished

int j;

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

if (max[p][j] - allot[p][j] > work[j])

break;

if (j == R) { // If resources can be allocated

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

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

safeSeq[count++] = p;

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

found = true;

}

}

}

if (found == false) {

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

return false; // Deadlock

}

}

// If system is safe, print the safe sequence

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

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

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

printf("\n");

return true;

}

void scenario1() {

printf("\nScenario 1: No Deadlock Scenario\n");

// Available resources (No Deadlock)

int avail[R] = {3, 3, 2};

// Maximum resource demand of each process

int max[P][R] = {

{7, 5, 3}, // Max demand of process 0

{3, 2, 2}, // Max demand of process 1

{9, 0, 2}, // Max demand of process 2

{2, 2, 2} // Max demand of process 3

};

// Resources currently allocated to each process

int allot[P][R] = {

{0, 1, 0}, // Resources allocated to process 0

{2, 0, 0}, // Resources allocated to process 1

{3, 0, 2}, // Resources allocated to process 2

{2, 1, 1} // Resources allocated to process 3

};

int processes[] = {0, 1, 2, 3};

isSafe(processes, avail, max, allot);

}

void scenario2() {

printf("\nScenario 2: Deadlock Scenario\n");

// Available resources (Deadlock situation)

int avail[R] = {1, 1, 1}; // Not enough resources to allocate safely

// Maximum resource demand of each process

int max[P][R] = {

{7, 5, 3}, // Max demand of process 0

{3, 2, 2}, // Max demand of process 1

{9, 0, 2}, // Max demand of process 2

{2, 2, 2} // Max demand of process 3

};

// Resources currently allocated to each process

int allot[P][R] = {

{0, 1, 0}, // Resources allocated to process 0

{2, 0, 0}, // Resources allocated to process 1

{3, 0, 2}, // Resources allocated to process 2

{2, 1, 1} // Resources allocated to process 3

};

int processes[] = {0, 1, 2, 3};

isSafe(processes, avail, max, allot);

}

int main() {

// Running both scenarios

scenario1(); // No Deadlock Scenario

scenario2(); // Deadlock Scenario

return 0;

}

2. Fcfs scheduling

#include <stdio.h>

#define P 4 // Number of processes

// Function to calculate waiting time and turnaround time

void calcWTandTAT(int at[], int bt[], int wt[], int tat[]) {

int start\_time[P];

start\_time[0] = at[0]; // First process starts at its arrival time

wt[0] = 0; // First process has no waiting time if it arrives at 0

// Calculate start time, waiting time, and turnaround time for each process

for (int i = 1; i < P; i++) {

start\_time[i] = start\_time[i - 1] + bt[i - 1];

wt[i] = start\_time[i] - at[i];

if (wt[i] < 0) {

start\_time[i] += -wt[i]; // Adjust start time if process arrives later

wt[i] = 0;

}

}

// Calculate turnaround time for each process

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

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

}

}

// Function to calculate and print table, averages, and finish times

void calcAvg(int at[], int bt[]) {

int wt[P], tat[P];

calcWTandTAT(at, bt, wt, tat);

int totalWT = 0, totalTAT = 0;

// Print the table of process details

printf("\nP\tAT\tBT\tWT\tTAT\n");

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

totalWT += wt[i];

totalTAT += tat[i];

printf("%d\t%d\t%d\t%d\t%d\n", i, at[i], bt[i], wt[i], tat[i]);

}

// Calculate and print averages

printf("\nAvg WT = %.2f", (float)totalWT / P);

printf("\nAvg TAT = %.2f", (float)totalTAT / P);

}

// Function to print process IDs and their finish times

void printFinishTimes(int at[], int bt[]) {

int wt[P], tat[P];

calcWTandTAT(at, bt, wt, tat);

// Print process IDs and finish times

printf("\nProcess Finish Times: ");

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

int finish\_time = at[i] + tat[i];

printf("P%d(%d) ", i, finish\_time);

}

printf("\n");

}

int main() {

// Arrival and Burst Times for each process

int at[] = {0, 2, 4, 6}; // Arrival times of processes

int bt[] = {10, 5, 8, 6}; // Burst times of processes

// Calculate and print table, averages, and finish times

calcAvg(at, bt);

printFinishTimes(at, bt);

return 0;

}

3. Producer consumer

// C program for the above approach

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size

// of buffer

int empty = 10, x = 0;

// Function to produce an item and

// add it to the buffer

void producer()

{

// Decrease mutex value by 1

--mutex;

// Increase the number of full

// slots by 1

++full;

// Decrease the number of empty

// slots by 1

--empty;

// Item produced

x++;

printf("\nProducer produces"

"item %d",

x);

// Increase mutex value by 1

++mutex;

}

// Function to consume an item and

// remove it from buffer

void consumer()

{

// Decrease mutex value by 1

--mutex;

// Decrease the number of full

// slots by 1

--full;

// Increase the number of empty

// slots by 1

++empty;

printf("\nConsumer consumes "

"item %d",

x);

x--;

// Increase mutex value by 1

++mutex;

}

// Driver Code

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'

// can give wrong value due to

// synchronization issues.

// 'critical' specifies that code is

// executed by only one thread at a

// time i.e., only one thread enters

// the critical section at a given time

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

// Switch Cases

switch (n) {

case 1:

// If mutex is 1 and empty

// is non-zero, then it is

// possible to produce

if ((mutex == 1)

&& (empty != 0)) {

producer();

}

// Otherwise, print buffer

// is full

else {

printf("Buffer is full!");

}

break;

case 2:

// If mutex is 1 and full

// is non-zero, then it is

// possible to consume

if ((mutex == 1)

&& (full != 0)) {

consumer();

}

// Otherwise, print Buffer

// is empty

else {

printf("Buffer is empty!");

}

break;

// Exit Condition

case 3:

exit(0);

break;

}

}

}

4. Disk scheduling - SCAN

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

int queue[20], n, head, i, j, seek = 0, max, diff;

int queue1[20], queue2[20], temp, temp1 = 0, temp2 = 0;

float avg;

char direction[10];

printf("Enter the maximum range of disk\n");

scanf("%d", &max);

printf("Enter the initial head position\n");

scanf("%d", &head);

printf("Enter the size of queue request\n");

scanf("%d", &n);

printf("Enter the direction (left/right)\n");

scanf("%s", direction);

printf("Enter the queue of disk positions to be read\n");

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

scanf("%d", &temp);

if (temp >= head) {

queue1[temp1++] = temp; // Requests on the right of the head

} else {

queue2[temp2++] = temp; // Requests on the left of the head

}

}

// Sort queue1 in ascending order (right side requests)

for (i = 0; i < temp1 - 1; i++) {

for (j = i + 1; j < temp1; j++) {

if (queue1[i] > queue1[j]) {

temp = queue1[i];

queue1[i] = queue1[j];

queue1[j] = temp;

}

}

}

// Sort queue2 in descending order (left side requests)

for (i = 0; i < temp2 - 1; i++) {

for (j = i + 1; j < temp2; j++) {

if (queue2[i] < queue2[j]) {

temp = queue2[i];

queue2[i] = queue2[j];

queue2[j] = temp;

}

}

}

// Start the seek sequence from the head

int index = 0;

queue[index++] = head; // Initial position of the head

// Build the seek sequence based on the direction

if (strcmp(direction, "left") == 0) {

// Move left first, then reverse to the right

for (i = 0; i < temp2; i++) {

queue[index++] = queue2[i];

}

for (i = 0; i < temp1; i++) {

queue[index++] = queue1[i];

}

} else if (strcmp(direction, "right") == 0) {

// Move right first, then reverse to the left

for (i = 0; i < temp1; i++) {

queue[index++] = queue1[i];

}

for (i = 0; i < temp2; i++) {

queue[index++] = queue2[i];

}

}

// Calculate total seek time and print the sequence

printf("Seek Sequence:\n");

for (j = 0; j < index - 1; j++) {

diff = abs(queue[j + 1] - queue[j]);

seek += diff;

printf("Disk head moves from %d to %d with seek %d\n", queue[j], queue[j + 1], diff);

}

printf("Total seek time is %d\n", seek);

avg = seek / (float)n;

printf("Average seek time is %.2f\n", avg);

return 0;

}

5. Page replacement - LRU

#include<stdio.h>

#include<limits.h>

int checkHit(int incomingPage, int queue[], int occupied){

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

if(incomingPage == queue[i])

return 1;

}

return 0;

}

void printFrame(int queue[], int occupied)

{

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

printf("%d\t\t\t",queue[i]);

}

int main()

{

int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3};

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

int frames = 3;

int queue[n];

int distance[n];

int occupied = 0;

int pagefault = 0;

printf("Page\t Frame1 \t Frame2 \t Frame3\n");

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

{

printf("%d: \t\t",incomingStream[i]);

// what if currently in frame 7

// next item that appears also 7

// didnt write condition for HIT

if(checkHit(incomingStream[i], queue, occupied)){

printFrame(queue, occupied);

}

// filling when frame(s) is/are empty

else if(occupied < frames){

queue[occupied] = incomingStream[i];

pagefault++;

occupied++;

printFrame(queue, occupied);

}

else{

int max = INT\_MIN;

int index;

// get LRU distance for each item in frame

for (int j = 0; j < frames; j++)

{

distance[j] = 0;

// traverse in reverse direction to find

// at what distance frame item occurred last

for(int k = i - 1; k >= 0; k--)

{

++distance[j];

if(queue[j] == incomingStream[k])

break;

}

// find frame item with max distance for LRU

// also notes the index of frame item in queue

// which appears furthest(max distance)

if(distance[j] > max){

max = distance[j];

index = j;

}

}

queue[index] = incomingStream[i];

printFrame(queue, occupied);

pagefault++;

}

printf("\n");

}

printf("Total Page Faults: %d",pagefault);

return 0;

}