1. MULTILEVEL CPU SCHEDULING:

#include <stdio.h>

#include<stdlib.h>

#include <stdbool.h>

#define MAX\_QUEUE\_SIZE 100

int totalTime=0;

int userProcess=0,systemProcess=0;

// Structure to represent a process

typedef struct {

int processID;

int arrivalTime;

int burstTime;

int remainingTime;

int priority; // 0 for system process, 1 for user process

} Process;

// Function to execute a process

void executeProcess(Process process) {

int i;

printf("Executing Process %d\n", process.processID);

// Simulating the execution time of the process

for (i = 1; i <= process.burstTime; i++) {

printf("Process %d: %d/%d\n", process.processID, i, process.burstTime);

}

printf("Process %d executed\n", process.processID);

}

// Function to perform FCFS scheduling for a queue of processes

void scheduleFCFS(Process system[],Process user[]) {

int i,j;

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

{

for(j=i+1;j<systemProcess;j++)

{

if(system[i].arrivalTime>system[j].arrivalTime)

{

Process temp=system[i];

system[i]=system[j];

system[j]=temp;

}

}

}

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

{

for(j=i+1;j<userProcess;j++)

{

if(user[i].arrivalTime>user[j].arrivalTime)

{

Process temp=user[i];

user[i]=user[j];

user[j]=temp;

}

}

}

int completed=0;

int currentProcess=-1;

bool isUserProcess=false;

int size=userProcess+systemProcess;

while(1)

{

int count=0;

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

{

if(system[i].remainingTime<=0)

{

count++;

}

}

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

{

if(user[j].remainingTime<=0)

{

count++;

}

}

if(count==size)

{

printf("\n end of processess");

exit(0);

}

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

{

if(totalTime>=system[i].arrivalTime && system[i].remainingTime>0)

{

currentProcess=i;

isUserProcess=false;

break;

}

}

if(currentProcess==-1)

{

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

{

if(totalTime>=user[j].arrivalTime && user[j].remainingTime>0)

{

currentProcess=j;

isUserProcess=true;

break;

}

}

}

if(currentProcess==-1)

{

totalTime++;

printf("\n %d idle time...",totalTime);

if(totalTime==1000)

{

exit(0);

}

continue;

}

if(isUserProcess==true)

{

user[currentProcess].remainingTime--;

printf("\n User process %d will excecute at %d ",user[currentProcess].processID,(totalTime));

totalTime++;

isUserProcess=false;

currentProcess=-1;

if(user[currentProcess].remainingTime==0)

{

completed++;

}

}else{

int temp=totalTime;

while(system[currentProcess].remainingTime--){

totalTime++;

}

if(system[currentProcess].remainingTime==0)

{

completed++;

}

printf("\n System process %d will excecute from %d to %d ",system[currentProcess].processID,temp,(totalTime));

isUserProcess=false;

currentProcess=-1;

}

}

}

int main() {

int numProcesses,i;

Process processes[MAX\_QUEUE\_SIZE];

// Reading the number of processes

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

scanf("%d", &numProcesses);

// Reading process details

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

printf("Process %d:\n", i + 1);

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

printf("System(0)/User(1): ");

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

processes[i].processID = i + 1;

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

if(processes[i].priority==1)

{

userProcess++;

}else{

systemProcess++;

}

}

Process systemQueue[MAX\_QUEUE\_SIZE];

int systemQueueSize = 0;

Process userQueue[MAX\_QUEUE\_SIZE];

int userQueueSize = 0;

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

if (processes[i].priority == 0) {

systemQueue[systemQueueSize++] = processes[i];

} else {

userQueue[userQueueSize++] = processes[i];

}

}

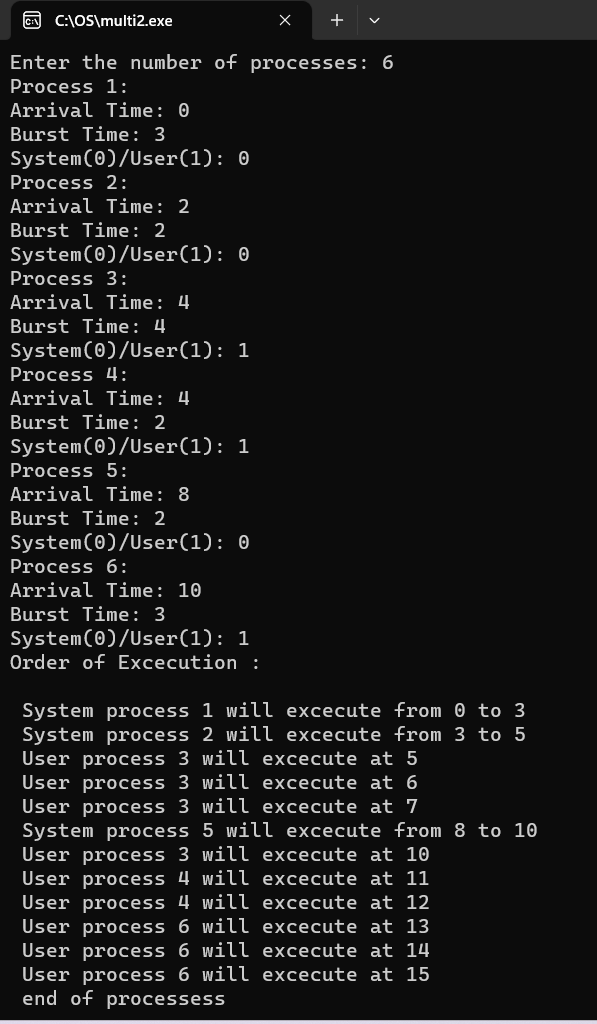
printf("Order of Excecution :\n");

scheduleFCFS(systemQueue,userQueue);

return 0;

}

OUTPUT:



1. RATE MONOMOTIC SCHEDULING:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#define MAX\_PROCESS 10

int num\_of\_process = 3, count, remain, time\_quantum;

int execution\_time[MAX\_PROCESS], period[MAX\_PROCESS], remain\_time[MAX\_PROCESS], deadline[MAX\_PROCESS], remain\_deadline[MAX\_PROCESS];

int burst\_time[MAX\_PROCESS], wait\_time[MAX\_PROCESS], completion\_time[MAX\_PROCESS], arrival\_time[MAX\_PROCESS];

// collecting details of processes

void get\_process\_info(int selected\_algo)

{

printf("Enter total number of processes (maximum %d): ", MAX\_PROCESS);

scanf("%d", &num\_of\_process);

if (num\_of\_process < 1)

{

printf("Do you really want to schedule %d processes? -\_-", num\_of\_process);

exit(0);

}

if (selected\_algo == 2)

{

printf("\nEnter Time Quantum: ");

scanf("%d", &time\_quantum);

if (time\_quantum < 1)

{

printf("Invalid Input: Time quantum should be greater than 0\n");

exit(0);

}

}

for (int i = 0; i < num\_of\_process; i++)

{

printf("\nProcess %d:\n", i + 1);

if (selected\_algo == 1)

{

printf("==> Burst time: ");

scanf("%d", &burst\_time[i]);

}

else if (selected\_algo == 2)

{

printf("=> Arrival Time: ");

scanf("%d", &arrival\_time[i]);

printf("=> Burst Time: ");

scanf("%d", &burst\_time[i]);

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

}

else if (selected\_algo > 2)

{

printf("==> Execution time: ");

scanf("%d", &execution\_time[i]);

remain\_time[i] = execution\_time[i];

if (selected\_algo == 4)

{

printf("==> Deadline: ");

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

}

else

{

printf("==> Period: ");

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

}

}

}

}

// get maximum of three numbers

int max(int a, int b, int c)

{

int max;

if (a >= b && a >= c)

max = a;

else if (b >= a && b >= c)

max = b;

else if (c >= a && c >= b)

max = c;

return max;

}

// calculating the observation time for scheduling timeline

int get\_observation\_time(int selected\_algo)

{

if (selected\_algo < 3)

{

int sum = 0;

for (int i = 0; i < num\_of\_process; i++)

{

sum += burst\_time[i];

}

return sum;

}

else if (selected\_algo == 3)

{

return max(period[0], period[1], period[2]);

}

else if (selected\_algo == 4)

{

return max(deadline[0], deadline[1], deadline[2]);

}

}

// print scheduling sequence

void print\_schedule(int process\_list[], int cycles)

{

printf("\nScheduling:\n\n");

printf("Time: ");

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

{

if (i < 10)

printf("| 0%d ", i);

else

printf("| %d ", i);

}

printf("|\n");

for (int i = 0; i < num\_of\_process; i++)

{

printf("P[%d]: ", i + 1);

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

{

if (process\_list[j] == i + 1)

printf("|####");

else

printf("| ");

}

printf("|\n");

}

}

void rate\_monotonic(int time)

{

int process\_list[100] = {0}, min = 999, next\_process = 0;

float utilization = 0;

for (int i = 0; i < num\_of\_process; i++)

{

utilization += (1.0 \* execution\_time[i]) / period[i];

}

int n = num\_of\_process;

if (utilization > n \* (pow(2, 1.0 / n) - 1))

{

printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");

exit(0);

}

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

{

min = 1000;

for (int j = 0; j < num\_of\_process; j++)

{

if (remain\_time[j] > 0)

{

if (min > period[j])

{

min = period[j];

next\_process = j;

}

}

}

if (remain\_time[next\_process] > 0)

{

process\_list[i] = next\_process + 1; // +1 for catering 0 array index.

remain\_time[next\_process] -= 1;

}

for (int k = 0; k < num\_of\_process; k++)

{

if ((i + 1) % period[k] == 0)

{

remain\_time[k] = execution\_time[k];

next\_process = k;

}

}

}

print\_schedule(process\_list, time);

}

int main(int argc, char \*argv[])

{

int option = 0;

printf("3. Rate Monotonic Scheduling\n");

printf("Select > ");

scanf("%d", &option);

printf("-----------------------------\n");

get\_process\_info(option); // collecting processes detail

int observation\_time = get\_observation\_time(option);

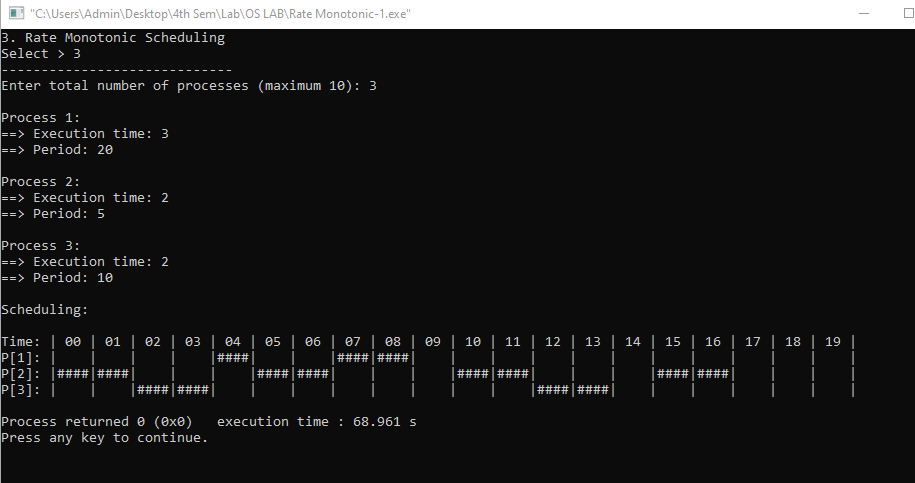
if (option == 3)

rate\_monotonic(observation\_time);

return 0;

}

OUTPUT:



3. EARLIEST DEADLINE FIRST

#include <stdio.h>

#define arrival 0

#define execution 1

#define deadline 2

#define period 3

#define abs\_arrival 4

#define execution\_copy 5

#define abs\_deadline 6

typedef struct

{

int T[7],instance,alive;

}task;

#define IDLE\_TASK\_ID 1023

#define ALL 1

#define CURRENT 0

void get\_tasks(task \*t1,int n);

int hyperperiod\_calc(task \*t1,int n);

float cpu\_util(task \*t1,int n);

int gcd(int a, int b);

int lcm(int \*a, int n);

int sp\_interrupt(task \*t1,int tmr,int n);

int min(task \*t1,int n,int p);

void update\_abs\_arrival(task \*t1,int n,int k,int all);

void update\_abs\_deadline(task \*t1,int n,int all);

void copy\_execution\_time(task \*t1,int n,int all);

int timer = 0;

int main()

{

task \*t;

int n, hyper\_period, active\_task\_id;

float cpu\_utilization;

printf("Enter number of tasks\n");

scanf("%d", &n);

t = malloc(n \* sizeof(task));

get\_tasks(t, n);

cpu\_utilization = cpu\_util(t, n);

printf("CPU Utilization %f\n", cpu\_utilization);

if (cpu\_utilization < 1)

printf("Tasks can be scheduled\n");

else

printf("Schedule is not feasible\n");

hyper\_period = hyperperiod\_calc(t, n);

copy\_execution\_time(t, n, ALL);

update\_abs\_arrival(t, n, 0, ALL);

update\_abs\_deadline(t, n, ALL);

while (timer <= hyper\_period)

{

if (sp\_interrupt(t, timer, n))

{

active\_task\_id = min(t, n, abs\_deadline);

}

if (active\_task\_id == IDLE\_TASK\_ID)

{

printf("%d Idle\n", timer);

}

if (active\_task\_id != IDLE\_TASK\_ID)

{

if (t[active\_task\_id].T[execution\_copy] != 0)

{

t[active\_task\_id].T[execution\_copy]--;

printf("%d Task %d\n", timer, active\_task\_id + 1);

}

if (t[active\_task\_id].T[execution\_copy] == 0)

{

t[active\_task\_id].instance++;

t[active\_task\_id].alive = 0;

copy\_execution\_time(t, active\_task\_id, CURRENT);

update\_abs\_arrival(t, active\_task\_id, t[active\_task\_id].instance, CURRENT);

update\_abs\_deadline(t, active\_task\_id, CURRENT);

active\_task\_id = min(t, n, abs\_deadline);

}

}

++timer;

}

free(t);

return 0;

}

void get\_tasks(task \*t1, int n)

{

int i = 0;

while (i < n)

{

printf("Enter Task %d parameters\n", i + 1);

printf("Arrival time: ");

scanf("%d", &t1->T[arrival]);

printf("Execution time: ");

scanf("%d", &t1->T[execution]);

printf("Deadline time: ");

scanf("%d", &t1->T[deadline]);

printf("Period: ");

scanf("%d", &t1->T[period]);

t1->T[abs\_arrival] = 0;

t1->T[execution\_copy] = 0;

t1->T[abs\_deadline] = 0;

t1->instance = 0;

t1->alive = 0;

t1++;

i++;

}

}

int hyperperiod\_calc(task \*t1, int n)

{

int i = 0, ht, a[10];

while (i < n)

{

a[i] = t1->T[period];

t1++;

i++;

}

ht = lcm(a, n);

return ht;

}

int gcd(int a, int b)

{

if (b == 0)

return a;

else

return gcd(b, a % b);

}

int lcm(int \*a, int n)

{

int res = 1, i;

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

{

res = res \* a[i] / gcd(res, a[i]);

}

return res;

}

int sp\_interrupt(task \*t1, int tmr, int n)

{

int i = 0, n1 = 0, a = 0;

task \*t1\_copy;

t1\_copy = t1;

while (i < n)

{

if (tmr == t1->T[abs\_arrival])

{

t1->alive = 1;

a++;

}

t1++;

i++;

}

t1 = t1\_copy;

i = 0;

while (i < n)

{

if (t1->alive == 0)

n1++;

t1++;

i++;

}

if (n1 == n || a != 0)

{

return 1;

}

return 0;

}

void update\_abs\_deadline(task \*t1, int n, int all)

{

int i = 0;

if (all)

{

while (i < n)

{

t1->T[abs\_deadline] = t1->T[deadline] + t1->T[abs\_arrival];

t1++;

i++;

}

}

else

{

t1 += n;

t1->T[abs\_deadline] = t1->T[deadline] + t1->T[abs\_arrival];

}

}

void update\_abs\_arrival(task \*t1, int n, int k, int all)

{

int i = 0;

if (all)

{

while (i < n)

{

t1->T[abs\_arrival] = t1->T[arrival] + k \* (t1->T[period]);

t1++;

i++;

}

}

else

{

t1 += n;

t1->T[abs\_arrival] = t1->T[arrival] + k \* (t1->T[period]);

}

}

void copy\_execution\_time(task \*t1, int n, int all)

{

int i = 0;

if (all)

{

while (i < n)

{

t1->T[execution\_copy] = t1->T[execution];

t1++;

i++;

}

}

else

{

t1 += n;

t1->T[execution\_copy] = t1->T[execution];

}

}

int min(task \*t1, int n, int p)

{

int i = 0, min = 0x7FFF, task\_id = IDLE\_TASK\_ID;

while (i < n)

{

if (min > t1->T[p] && t1->alive == 1)

{

min = t1->T[p];

task\_id = i;

}

t1++;

i++;

}

return task\_id;

}

float cpu\_util(task \*t1, int n)

{

int i = 0;

float cu = 0;

while (i < n)

{

cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];

t1++;

i++;

}

return cu;

}

OUTPUT:

