6.

**Description:**

**First Come First Serve (FCFS): It** is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.

**Shortest Job First (SJF) :** Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-pre-emptive algorithm.

**Turn around time:** Turnaround time is the total amount of time spent by the process from coming in the ready state for the first time to its completion.

**ALGORITHM:**

**First Come First Serve(FCFS):**

Step 1-> In function int waitingtime(int proc[], int n, int burst\_time[], int wait\_time[])

Set wait\_time[0] = 0

Loop For i = 1 and i < n and i++

Set wait\_time[i] = burst\_time[i-1] + wait\_time[i-1]

End For

Step 2-> In function int turnaroundtime( int proc[], int n, int burst\_time[], int wait\_time[], int tat[])

Loop For i = 0 and i < n and i++

Set tat[i] = burst\_time[i] + wait\_time[i]

End For

Step 3-> In function int avgtime( int proc[], int n, int burst\_time[])

Declare and initialize wait\_time[n], tat[n], total\_wt = 0, total\_tat = 0;

Call waitingtime(proc, n, burst\_time, wait\_time)

Call turnaroundtime(proc, n, burst\_time, wait\_time, tat)

Loop For i=0 and i<n and i++

Set total\_wt = total\_wt + wait\_time[i]

Set total\_tat = total\_tat + tat[i]

Print process number, burstime wait time and turnaround time

End For

Print "Average waiting time =i.e. total\_wt / n

Print "Average turn around time = i.e. total\_tat / n

Step 4-> In int main()

Declare the input int proc[] = { 1, 2, 3}

Declare and initialize n = sizeof proc / sizeof proc[0]

Declare and initialize burst\_time[] = {10, 5, 8}

Call avgtime(proc, n, burst\_time)

Stop

**Shortest Job First(SJF):**

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

**Description (purpose of use):**

Input the processes along with their burst time (bt).

Find waiting time (wt) for all processes.

As first process that comes need not to wait so

waiting time for process 1 will be 0 i.e. wt[0] = 0.

Find waiting time for all other processes i.e. for

process i ->

wt[i] = bt[i-1] + wt[i-1] .

Find turnaround time = waiting time + burst time

for all processes.

Find average waiting time =

total\_waiting\_time / no\_of\_processes.

Similarly, find average turnaround time =

total\_turn\_around\_time / no\_of\_processes.

**Code Snippet:**

**First Come First Serve(FCFS):**

#include<stdio.h>

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

wt[0] = 0;

for (int i = 1; i < n ; i++ )

wt[i] = bt[i-1] + wt[i-1] ;

}

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

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

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

}

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst time Waiting time Turn around time\n");

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

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

printf(" %d ",(i+1));

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

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

printf(" %d\n",tat[i] );

}

int s=(float)total\_wt / (float)n;

int t=(float)total\_tat / (float)n;

printf("Average waiting time = %d",s);

printf("\n");

printf("Average turn around time = %d ",t);

}

int main()

{

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

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

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

findavgTime(processes, n, burst\_time);

return 0;

}

**Shortest Job First(SJF):**

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("nEnter Burst Time:n");

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

{

printf("p%d:",i+1);

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

p[i]=i+1;

}

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

{ pos=i;

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

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n;

total=0;

printf("nProcesst Burst Time tWaiting TimetTurnaround Time");

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

{

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

total+=tat[i];

printf("np%dtt %dtt %dttt%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n;

printf("nnAverage Waiting Time=%f",avg\_wt);

printf("nAverage Turnaround Time=%fn",avg\_tat);

}

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

Solution:

Average turnaround for these processes:

(8 + (12 - 0.4) + (13 - 1)) / 3 = 10.53

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

Solution:

Average turnaround for these processes:

(8 + (9 - 1) + (13 – 0.4)) / 3 = 9.53

c. Compute what average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes *P*1 and *P*2 are waiting during this idle time, so their waiting time may increase.

Solution:

CPU is left idle:

Average turnaround for these processes:

((2 - 1) + (6 – 0.4 ) + ( 14 - 0)) / 3 = 6.87

21.A number of cats and mice inhabit a house. The cats and mice have worked out a deal where the mice can steal pieces of the cats’ food, so long as the cats never see the mice actually doing so. If the cats see the mice, then the cats must eat the mice (or else lose face with all of their cat friends). There are **NumBowls** cat food dishes, **NumCats** cats, and **NumMice** mice. Your job is to synchronize the cats and mice so that the following requirements are satisfied:

No mouse should ever get eaten. You should assume that if a cat is eating at a food dish, any mouse attempting to eat from that dish or any other food dish will be seen and eaten. When cats aren’t eating, they will not see mice eating. In other words, this requirement states that if a cat is eating from any bowl, then no mouse should be eating from any bowl. Only one mouse or one cat may eat from a given dish at any one time. Neither cats nor mice should starve. A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve.

Solution:

**DESCRIPTION:**

**Process Synchronization**: It means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions.

**Critical Section Problem:**

* A Critical Section is a code segment that accesses shared variables/resources and has to be executed as an atomic action.
* Critical section problem refers to problem of how to ensure that at most one process is executing its critical section at a given time.

A screenshot of a cell phone

Description automatically generated

1. **Entry section**: Before entering the critical section, a process must request permission to enter
2. **Critical section**: After permission is granted, a process may execute the code in the critical section. Other threads respect the request and keep out of their critical sections.
3. **Exit section**: The process acknowledges it has left its critical section.

**Solution to Critical Section Problem:**

A solution to the critical section problem must satisfy the following three conditions :

**1.Mutual Exclusion** – If process Pi is executing in its critical section, then no other processes can be executing in their critical sections  
**2.Progress** – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely  
**3.Bounded Waiting** –  A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

**ALGORITHM:**

Boolean waiting[N];

int j;

Boolean key;

do {

waiting[i] = TRUE;

key = TRUE;

while( waiting[i] && key )

key = test\_and\_set( lock );

waiting[ i ] = FALSE;

j = ( i + 1 ) mod N;

while ( ( j != i ) && ( ! waiting[ j ] ) )

j = ( j + 1 ) % N;

if ( j == i )

lock = FALSE;

else

waiting[ j ] = FALSE;

} while (TRUE);

**Description (Purpose of use):**

Question 1.

List all of the synchronization primitives and shared variables that you have used to synchronize the

cats and mice and identify the purpose of each one.

Question 2.

Briefly explain how the listed variables and synchronization primitives are used to synchronize the cats

and mice.

Question 3.

Briefly explain why it is not possible for two creatures to eat from the same bowl at the same time

under your synchronization technique.

Question 4.

Briefly explain why it is not possible for mice to be eaten by cats under your synchronization technique.

6

Question 5.

Briefly explain why it is not possible for cats or mice to starve under your synchronization technique.

Question 6.

Briefly discuss the fairness and efficiency of your synchronization technique, and describe any unfairness

(bias against cats or mice) that you can identify in your design. Did you have to balance fairness and

efficiency in some aspect of your design? If so, explain how you chose to do so.

To get answer for all these question we have to run the program with conditions being implemented correctly.

**Code Snippet:**

#include<stdio.h>

#include<pthread.h>

int shared=5;

void \*func1()

{

int local;

/\*critical section \*/

local=shared;

local=local+1;

sleep(5); /\* causes a context switch \*/

shared=local;

/\*critical section \*/

printf(“shared in func1: %d\n”,shared);

pthread exit(NULL);

}

void \*func2()

{

int local;

/\*critical section \*/

local=shared;

local=local-1;

shared=local;

/\*critical section \*/

printf(“shared in func2: %d\n”,shared);

pthread exit(NULL);

}

int main()

{

pthread t t1,t2;

pthread create(&t1,NULL,func1,NULL);

pthread create(&t2,NULL,func2,NULL);

pthread join(t1,NULL);

pthread join(t2,NULL);

}

**Description:**

A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve. Your solution must not rely on knowledge of the numbers of cats and mice in the system. In particular, you should not make direct or indirect use of the variables NumCats and NumMice in your solution. (Those parameters should be used only by the catmouse() function to create the correct numbers of cats and mice to run a particular test.) It is OK to make use of the NumBowls parameter in your solution. There are many ways to synchronize the cats and mice that will satisfy the requirements above. From among the possible solutions that satisfy the requirements, we prefer solutions that are (a) efficient and (b) fair. An efficient solution avoids allowing food bowls to go unused when there are creatures waiting to eat. Sometimes it is necessary to make creatures wait, even when there are unused food bowls, in order to satisfy the synchronization requirements, so an efficient solution should avoid unnecessary delays. For example, a solution that uses a single lock to ensure that only one creature eats at a time (regardless of the number of bowls) satisfies the problem requirements, but it is not efficient because it may delay creatures unnecessarily. To see this, consider a scenario in which there are 5 cats, 5 bowls and no mice. In that scenario, the 5 cats should should be able to eat simultaneously. Fairness means avoiding favouring some creatures over others, and in particular avoiding bias against cats or mice. For example, a solution that makes all mice wait until all cats have finished eating as many times as they want satisfies the three problem requirements, but it is biased in favour of the cats. There is a tradeoff between efficiency and fairness. For example, consider a situation in which there are two bowls, a cat is eating at one of the bowls, and a mouse is waiting to eat so that it does not get seen by the cat that is eating. Now, suppose that a second cat shows up, wanting to eat. Should you allow it to eat immediately from the unused bowl? Or should you make it wait until the mouse has had a chance to eat? After all, the mouse was there first. If your solution allows the cat to eat, it has sacrificed some fairness for efficiency by allowing the cat to jump in front of the waiting mouse. On the other hand, if your solution makes the second cat wait then it sacrifices some efficiency since a bowl is left unused though there is an eligible cat waiting