**OPERATING SYSTEMS LAB**

**Module-1**

**Practice-1**

**Aim:** Implementation of FCFS Scheduling Algorithm in C Language

**Description:**

The First Come First Serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. Simplest CPU scheduling algorithm that schedules according to arrival times of processes. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked to the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-pre-emptive scheduling algorithm.

**Source Code:**

//FCFS

#include<stdio.h>

int main()

{

int n,bt[20],wt[20],tat[20],p[20],i,j,t=0;

float avwt=0,avtat=0;

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

scanf("%d",&n);

printf("Enter Process Burst Time\n");

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

{

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

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

p[i]=i+1;

}

wt[0]=0; //waiting time for first process is 0

//calculating waiting time

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

{

wt[i]=0;

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

wt[i]+=bt[j];

}

printf("Order of process Execution is\n");

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

{

printf("P%d is executed from %d to %d\n",p[i],t,t+bt[i]);

t+=bt[i];

}

printf("Process\t\tBurst Time\tWaiting Time\tTurnaround Time");

//calculating turnaround time

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

{

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

avwt+=wt[i];

avtat+=tat[i];

printf("\nP%d\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);

}

avwt/=i;

avtat/=i;

printf("\nAverage Waiting Time:%.2f",avwt);

printf("\nAverage Turnaround Time:%.2f",avtat);

return 0;

}

**Output:**

Enter total number of processes:5

Enter Process Burst Time

P1:8

P2:6

P3:1

P4:9

P5:3

Order of process Execution is

P1 is executed from 0 to 8

P2 is executed from 8 to 14

P3 is executed from 14 to 15

P4 is executed from 15 to 24

P5 is executed from 24 to 27

Process Burst Time Waiting Time Turnaround Time

P1 8 0 8

P2 6 8 14

P3 1 14 15

P4 9 15 24

P5 3 24 27

Average Waiting Time:12.20

Average Turnaround Time:17.60

**Practie-2**

**Aim:** To implement SJF Scheduling Algorithm in C language

**Description:**

The **Shortest Job First (SJF)** scheduling algorithm, also known as **Shortest Job Next (SJN)**, selects the waiting process with the smallest execution time to execute next. In non-pre-emptive SJF, once a process starts executing, it continues until completion without interruption. The process with the **smallest burst time** is selected from the ready queue for execution. It aims to minimize the average waiting time among all scheduling algorithms. However, it may cause **starvation** if shorter processes keep arriving. This can be mitigated using the concept of **ageing**. Non-pre-emptive SJF is suitable when accurate estimates of running time are available.

**Source Code:**

//SJF

#include<stdio.h>

void main()

{

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

float avg\_wt,avg\_tat;

printf("Enter number of processes: ");

scanf("%d",&n);

printf("Enter Burst Time:\n");

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

{

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

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

p[i]=i+1; //contains process number

}

//sorting burst time in ascending order using selection sort

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; //waiting time for first process will be zero

//calculate waiting time

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; //average waiting time

total=0;

printf("Order of process Execution is\n");

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

{

printf("P%d is executed from %d to %d\n",p[i],t,t+bt[i]);

t+=bt[i];

}

printf("Process\t Burst Time \tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i]; //calculate turnaround time

total+=tat[i];

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

}

avg\_tat=(float)total/n; //average turnaround time

printf("\nAverage Waiting Time=%.2f",avg\_wt);

printf("\nAverage Turnaround Time=%.2f\n",avg\_tat);

}

**Output:**

Enter number of processes: 5

Enter Burst Time:

P1:8

P2:6

P3:1

P4:9

P5:3

Order of process Execution is

P3 is executed from 0 to 1

P5 is executed from 1 to 4

P2 is executed from 4 to 10

P1 is executed from 10 to 18

P4 is executed from 18 to 27

Process Burst Time Waiting Time Turnaround Time

P3 1 0 1

P5 3 1 4

P2 6 4 10

P1 8 10 18

P4 9 18 27

Average Waiting Time=6.60

Average Turnaround Time=12.00

**Practice-3**

**Aim:** To implement non-pre-emptive Priority Scheduling Algorithm in C language

**Description:**

**Priority CPU Scheduling Algorithm** is used to schedule the processes as per the priorities assigned to respective processes. In Non-preemptive Priority CPU Scheduling Algorithm, processes are scheduled as per the priorities assigned to respective task and next process is not schedule until and unless current execution of process is not completely finished.

**Source Code:**

//Priority Non-preemptive

#include <stdio.h>

void swap(int \*a,int \*b)

{

int temp=\*a;

\*a=\*b;

\*b=temp;

}

int main()

{

int n,i,j,t=0,wait\_time[20],turn\_Around[20],total\_wait\_time = 0,total\_Turn\_Around = 0;

printf("Enter Number of Processes: ");

scanf("%d",&n);

int burst[n],priority[n],index[n];

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

{

printf("Enter Burst Time and Priority Value for Process %d: ",i+1);

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

index[i]=i+1;

}

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

{

int temp=priority[i],m=i;

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

{

if(priority[j] < temp)

{

temp=priority[j];

m=j;

}

}

swap(&priority[i], &priority[m]);

swap(&burst[i], &burst[m]);

swap(&index[i],&index[m]);

}

printf("Order of process Execution is\n");

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

{

printf("P%d is executed from %d to %d\n",index[i],t,t+burst[i]);

t+=burst[i];

}

printf("Process Id\tBurst Time\tWait Time\\t Turn Around Time\n");

wait\_time[0]=0;

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

{

wait\_time[i]=0;

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

wait\_time[i] += burst[j];

total\_wait\_time += wait\_time[i];

}

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

turn\_Around[i]= burst[i]+wait\_time[i];

total\_Turn\_Around+=turn\_Around[i];

printf("P%d\t\t%d\t\t%d\t\t%d\n",index[i],burst[i],wait\_time[i],turn\_Around[i]);

}

float avg\_wait\_time =(float)total\_wait\_time / n;

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

float avg\_Turn\_Around =(float)total\_Turn\_Around / n;

printf("Average TurnAround Time is %.2f",avg\_Turn\_Around);

return 0;

}

**Output:**

Enter Number of Processes: 5

Enter Burst Time and Priority Value for Process 1: 8 4

Enter Burst Time and Priority Value for Process 2: 6 1

Enter Burst Time and Priority Value for Process 3: 1 2

Enter Burst Time and Priority Value for Process 4: 9 2

Enter Burst Time and Priority Value for Process 5: 3 3

Order of process Execution is

P2 is executed from 0 to 6

P3 is executed from 6 to 7

P4 is executed from 7 to 16

P5 is executed from 16 to 19

P1 is executed from 19 to 27

Process Id Burst Time Wait Time Turn Around Time

P2 6 0 6

P3 1 6 7

P4 9 7 16

P5 3 16 19

P1 8 19 27

Average waiting time is 9.60

Average TurnAround Time is 15.00

**Practice-4**

**Aim:** To implement Round Robin Scheduling Algorithm in C language

**Description:**

One of the CPU scheduling strategies is round-robin. It is **preemptive** in nature that it switches between processes according to the time allotted for each process The round-robin scheduling algorithm equally distributes each resource and processes each division in a circular sequence without giving any consideration to priority. The terms "time quantum" and "time slice" are used to distribute all resources equally. The time slice that is used to switch between the processes is known as **Quantum**. Round Robin scheduling is cyclic in nature and is also known as Time Slicing Scheduling. If a process is finished, it is removed from the ready queue in this round-robin method; otherwise, it will return to the ready queue for the remaining execution. The running queue’s processes that have arrived and are awaiting execution are in the ready queue. The processes that originated from the ready queue are carried out using the running queue. Let’s look at the round-robin scheduling method right now.

**Source Code:**

// Round Robin scheduling program in c

#include<stdio.h>

void main()

{

int i, n, sum=0,count=0,y,quant, wt=0, tat=0, at[10], bt[10], temp[10];

printf("Enter number of process in the system: ");

scanf("%d", &n);

y = n;

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

{

printf("Enter the Arrival and Burst time of the Process %d : ", i+1);

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No %d \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==n-1)

i=0;

else if(at[i+1]<=sum)

i++;

else

i=0;

}

printf("\nAverage Waiting time is %.2f\n", wt\*1.0/n);

printf("Average TurnAround Time is %.2f",tat\*1.0/n);

}

**Output:**

Enter number of process in the system: 4

Enter the Arrival and Burst time of the Process 1 : 0 5

Enter the Arrival and Burst time of the Process 2 : 1 4

Enter the Arrival and Burst time of the Process 3 : 2 2

Enter the Arrival and Burst time of the Process 4 : 4 1

Enter the Time Quantum for the process: 2

Process No Burst Time TAT Waiting Time

Process No 3 2 4 2

Process No 4 1 3 2

Process No 2 4 10 6

Process No 1 5 12 7

Average Waiting time is 4.25

Average TurnAround Time is 7.25

**Practice-5**

**Aim:** To implement Producer-Consumer Problem in C language

**Description:**

The producer-consumer problem in C is one of the most famous problems associated with operating systems. In the producer-consumer problem in C, there is a producer which produces products (data) and there is a consumer who consumes the products produced by the producer. Now, in the producer-consumer problem in C, there is a buffer. A buffer is a temporary storage area in the memory that stores data. In the producer-consumer problem in C, we have been provided with a fixed-sized buffer. The buffer in the Producer-Consumer Problem in C contains the produced items by the producer. The consumer consumes the products from the same buffer. In simpler terms, we can say that the buffer is a shared space between the producer and consumer.

* **Producer**: The producer's role is to produce or generate the data and put it in the buffer and start again with the same.
* **Consumer**: The consumer's role is to consume the produced data from the same shared buffer. When the consumer consumes the data, the data is removed from the buffer.

Since the problem is that the producer keeps on producing the data even if the buffer is full and the consumer tries to consume the data even if the buffer is empty. We can either put the producer to sleep or discard the data produced by the producer if the buffer is full. So, whenever the consumer tries to consume the data from the buffer, it will notify the producer that the data is being consumed. This will make the producer produce the data again.

Similarly, we can put the consumer to sleep when there is no data in the buffer (empty buffer case). So, whenever the producer produces the data and put it into the buffer, it will notify the consumer that the data is being produced. This will make the consumer consume the data again.

We should also notice that if there is an inadequate solution is proposed then there may arise a situation when both the producer and consumer are on wait (to be awakened)

The main cause behind the producer-consumer problem is that there is no way that will tell the consumer if the buffer is empty or not. Similarly, the producer cannot know that the buffer is already full.

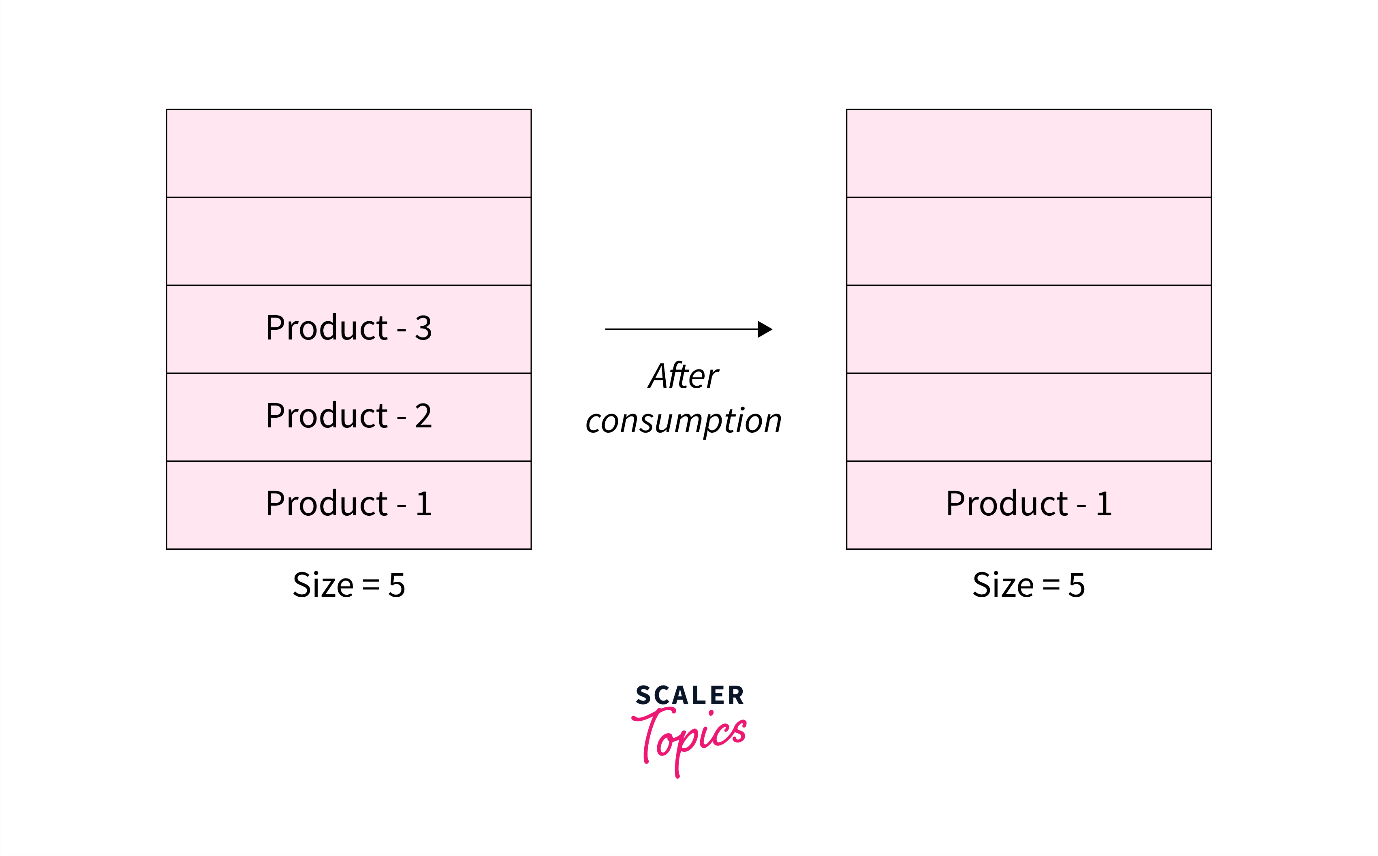
So, as we have discussed above, we can put the consumer to sleep when there is no data in the buffer (empty buffer case). We can similarly put the producer to sleep if the buffer is full. By the term sleep, we mean that the producer should not produce any product when the consumer is consuming the product and vice versa. We can use the sleep() function to put the consumer or the producer to sleep.

**Examples of Producer-Consumer Problems in C**

**Example 1**: Suppose we have a buffer of size = 5.

Initially, the buffer is empty and neither producer is producing the data nor the consumer is consuming the data.

Suppose that the producer starts producing the data one at a time. The producer produces 3 data during the production and when the consumer comes he/she consumes two of the three produced data. Now in this scenario, we can see that neither of the two has caused the problem as when the consumer was consuming the data, the buffer was not empty. Similarly, when the producer was producing the data the buffer was not full.



**Source Code:**

// C Program for Producer-Consumer Problem

//The idea is to use the concept of parallel programming and Critical Section to implement the Producer-Consumer problem in C language using OpenMP.

#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;

}

}

}

**Output:**

1. Press 1 for Producer

2. Press 2 for Consumer

3. Press 3 for Exit

Enter your choice:1

Producer produces item 1

Enter your choice:1

Producer produces item 2

Enter your choice:1

Producer produces item 3

Enter your choice:2

Consumer consumes item 3

Enter your choice:2

Consumer consumes item 2

Enter your choice:2

Consumer consumes item 1

Enter your choice:2

Buffer is empty!

Enter your choice:1

Producer produces item 1

Enter your choice:1

Producer produces item 2

Enter your choice:1

Producer produces item 3

Enter your choice:1

Producer produces item 4

Enter your choice:

Producer produces item 5

Enter your choice:1

Producer produces item 6

Enter your choice:1

Producer produces item 7

Enter your choice:1

Producer produces item 8

Enter your choice:

Producer produces item 9

Enter your choice:1

Producer produces item 10

Enter your choice:1

Buffer is full!

Enter your choice:2

Consumer consumes item 10

Enter your choice:2

Consumer consumes item 9

Enter your choice:2

Consumer consumes item 8

Enter your choice:2

Consumer consumes item 7

Enter your choice:2

Consumer consumes item 6

Enter your choice:2

Consumer consumes item 5

Enter your choice:2

Consumer consumes item 4

Enter your choice:2

Consumer consumes item 3

Enter your choice:2

Consumer consumes item 2

Enter your choice:2

Consumer consumes item 1

Enter your choice:2

Buffer is empty!

Enter your choice:2

Buffer is empty!

Enter your choice:3

**Practice-6**

**Aim:** To implement Bankers Algorithm in C language

**Description:**

The Banker's Algorithm, a key deadlock avoidance and detection tool, strategically allocates resources by simulating their distribution to predetermined maximums. This process includes an 's-state' safety check to decide if resource allocation can proceed without leading to a deadlock condition. The Banker’s Algorithm is a smart way for computer systems to manage how programs use resources, like memory or CPU time. It helps prevent situations where programs get stuck and can’t finish their tasks, which is called deadlock. By keeping track of what resources each program needs and what’s available, the algorithm makes sure that programs only get what they need in a safe order. This helps computers run smoothly and efficiently, especially when lots of programs are running at the same time.

**Source Code:**

//Bankers Algorithm in C

#include <stdio.h>

void main() {

int k=0,a=0,b=0,instance[5],availability[5],allocated[10][5],need[10][5],MAX[10][5],process,P[10],op[10],no\_of\_resources, cnt=0,i, j;

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

scanf("%d", &no\_of\_resources);

printf("Enter the max instances of each resources\n");

for (i=0;i<no\_of\_resources;i++) {

//availability[i]=0;

printf("%c= ",(i+65));

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

}

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

scanf("%d", &process);

printf("Enter the allocation matrix \n");

for (i=0;i<no\_of\_resources;i++)

printf(" \t%c",(i+65));

printf("\n");

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

P[i]=i;

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

for (j=0;j<no\_of\_resources;j++) {

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

//availability[j]+=allocated[i][j];

}

}

printf("Enter the MAX matrix \n");

for (i=0;i<no\_of\_resources;i++) {

printf(" \t%c",(i+65));

//availability[i]=instance[i]-availability[i];

}

printf("\n");

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

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

for (j=0;j<no\_of\_resources;j++)

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

}

printf("\n");

printf("Availability Matrix \n");

for (i=0;i<no\_of\_resources;i++)

printf("%c\t",(i+65));

printf("\n");

for (i=0;i<no\_of\_resources;i++)

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

printf("\n");

A: a=-1;

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

cnt=0;

b=P[i];

for (j=0;j<no\_of\_resources;j++) {

need[b][j] = MAX[b][j]-allocated[b][j];

if(need[b][j]<=availability[j])

cnt++;

}

if(cnt==no\_of\_resources) {

op[k++]=P[i];

for (j=0;j<no\_of\_resources;j++)

availability[j]+=allocated[b][j];

} else

P[++a]=P[i];

}

if(a!=-1) {

process=a+1;

goto A;

}

printf("Order of Processes without Dead Lock\n");

printf("<");

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

printf(" P[%d] ",op[i]);

printf(">");

}

**Output:**

Enter the number of resources : 3

Enter the max instances of each resources

A= 10

B= 5

C= 7

Enter the number of processes : 5

Enter the allocation matrix

A B C

P[0] 0 1 0

P[1] 2 0 0

P[2] 3 0 2

P[3] 2 1 1

P[4] 0 0 2

Enter the MAX matrix

A B C

P[0] 7 5 3

P[1] 3 2 2

P[2] 9 0 2

P[3] 2 2 2

P[4] 4 3 3

Availability Matrix

A B C

3 3 2

Order of Processes without Dead Lock

< P[1] P[3] P[4] P[0] P[2] >

**Module-2**

**Practice-1**

**Aim:**  To implement FIFO page replacement algorithm in C

**Description:**

The simplest algorithm for replacing pages is this one. The operating system maintains a queue for all of the memory pages in this method, with the oldest page at the front of the queue. The first page in the queue is chosen for removal when a page has to be replaced.

**Source Code:**

//FIFO Page Replacement

#include <stdio.h>

int main()

{

int pageFaults = 0,frames,m, n, s, pages,i;

printf("Enter number of frames: ");

scanf("%d", &frames);

printf("Enter number of pages: ");

scanf("%d", &pages);

int incomingStream[pages];

printf("Enter reference string: ");

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

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

printf(" Incoming \t Frame 1 \t Frame 2 \t Frame 3 \t Frame 4 ");

int temp[ frames ];

for(m = 0; m < frames; m++)

temp[m] = -1;

for(m = 0; m < pages; m++)

{

s = 0;

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

{

if(incomingStream[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

temp[m] = incomingStream[m];

else if(s == 0)

temp[(pageFaults - 1) % frames] = incomingStream[m];

printf("\n");

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

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

{

if(temp[n] != -1)

printf(" %d\t\t\t\t", temp[n]);

else

printf(" - \t\t\t\t");

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

**Output:**

**Practice-2**

**Aim:** To implement Second Chance page replacement algorithm in C

**Description:**

The **Second Chance Page Replacement Algorithm**, also known as the **Clock Algorithm**, is a page replacement policy used in operating systems to manage virtual memory. It is an enhancement of the **First-In-First-Out (FIFO)** algorithm, designed to reduce the number of unnecessary page replacements by giving pages a "second chance" before they are replaced.

In the Second Chance algorithm, each page in memory is associated with a reference bit. When a page is accessed, its reference bit is set to 1. When a page needs to be replaced, the algorithm scans through the pages in a circular manner (like a clock) and checks the reference bit of each page. If the reference bit is 0, the page is replaced. If the reference bit is 1, the bit is cleared (set to 0), and the page is given a second chance to stay in memory. The algorithm continues scanning until it finds a page with a reference bit of 0.

**Source Code:**

//Second Chance Page Replacemnet

#include <stdio.h>

int hasspace(int m, int f[m][3])

{

int i;

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

{

if(f[i][0] == -1)

return i;

}

return -1;

}

int available(int m, int f[m][3], int x)

{

int i;

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

if(f[i][0] == x)

return i;

return -1;

}

void printF(int m, int f[m][3])

{

int i;

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

{

if(f[i][0] != -1)

{

if(i != 0)

printf("\t");

printf("\t\t%d\t %d\t\t %d\n",f[i][0],f[i][1],f[i][2]);

}

}

}

int doweskip(int k, int m, int skip[m])

{

int i;

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

if(k == skip[i])

return 1;

return 0;

}

int FIFO(int m, int f[m][3], int skip[m])

{

int i, index, maxtc, flag = 0;

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

{

if(doweskip(f[i][0],m,skip) == 0)

{

if(flag == 0)

{

maxtc = f[i][1];

index = i;

flag = 1;

}

else if(f[i][1] > maxtc)

{

maxtc = f[i][1];

index = i;

}

}

}

return index;

}

void updateskip(int k, int m, int skip[m])

{

int i;

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

{

if(skip[i] == -1)

{

skip[i] = k;

return;

}

}

}

int getvictim(int m, int f[m][3])

{

int i, index, count = 0, maxtc, flag = 0, skip[m];

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

skip[i] = -1;

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

if(f[i][2] == 1)

count++;

if(count == 0 || count == m)

{

if(count == m)

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

f[i][2] = 0;

index = FIFO(m,f,skip);

}

else

{

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

{

index = FIFO(m,f,skip);

if(f[index][2] == 1)

{

f[index][2] = 0;

updateskip(f[index][0],m,skip);

}

else

break;

}

}

return index;

}

void timecounter(int k, int m, int f[m][3], int flag)

{

int i;

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

{

if(flag == 1 && f[i][0] != -1)

f[i][1]++;

else if(flag == 0 && f[i][0] != -1)

{

if(i == k)

f[i][1] = 0;

else

f[i][1]++;

}

}

}

void line()

{

int i;

printf("\n\t");

for(i=0; i<55; i++)

printf("\_");

printf("\n\n");

}

int iterate(int n, int s[n], int m, int f[m][3])

{

printf("\n\n\tPage\t Frame\ttime-counter\treference-bit");

line();

int i, k = -1, pf = 0, index = -1;

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

{

//printf("\t");

if( available(m,f,s[i]) == -1 )

{

pf++;

if(hasspace(m,f) != -1)

{

index = hasspace(m,f);

f[index][0] = s[i];

timecounter(index,m,f,0);

f[index][2] = 0;

}

else

{

index = getvictim(m,f);

f[index][0] = s[i];

timecounter(index,m,f,0);

printf(" !--");

}

}

else

{

index = available(m,f,s[i]);

timecounter(index,m,f,1);

f[index][2] = 1;

}

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

printF(m,f);

line();

}

return pf;

}

void setdefault(int m, int f[m][3])

{

int i;

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

{

f[i][0] = -1;

f[i][1] = -1;

f[i][2] = -1;

}

}

void main()

{

int n, m = 3, i;

printf("\n\n\t\tSecond Chance Algorithm\n");

printf("\n Enter the length of the reference string: ");

scanf("%d",&n);

int s[n];

printf("\n Enter the reference string: ");

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

{

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

if(s[i] < 0)

{

printf("\n Reference string cannot have negative entries.\n");

return;

}

}

printf("\n Enter the no. of frames: ");

scanf("%d",&m);

printf("\n");

int f[m][3];

setdefault(m,f);

int pf = iterate(n,s,m,f);

printf("\n\tThe Total number of page faults: %d\n\n",pf);

}

**Output:**

**Practice-3**

**Aim:** To implement LRU page replacement algorithm in C

**Description:**

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

**Source Code:**

//LRU Page Replacement

#include <stdio.h>

// Function to find the index of the least recently used page in frames

int findLRU(int time[], int n) {

int i, minimum = time[0], pos = 0;

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

if (time[i] < minimum) {

minimum = time[i];

pos = i;

}

}

return pos;

}

int main() {

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], counter = 0, time[10], i, j, pos, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter reference string: ");

for (i = 0; i < no\_of\_pages; ++i)

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

for (i = 0; i < no\_of\_frames; ++i) {

frames[i] = -1;

time[i] = 0; // Initialize the time array to 0

}

for (i = 0; i < no\_of\_pages; ++i) {

int page = pages[i];

int page\_found = 0;

// Check if the page is already in frames

for (j = 0; j < no\_of\_frames; ++j) {

if (frames[j] == page) {

time[j] = counter++; // Update the time of use

page\_found = 1;

break;

}

}

// If the page is not in frames, find the LRU page to replace

if (!page\_found) {

pos = findLRU(time, no\_of\_frames);

frames[pos] = page; // Replace the LRU page

time[pos] = counter++; // Update the time of use

faults++;

}

printf("Current frames: ");

for (j = 0; j < no\_of\_frames; ++j) {

printf("%d\t", frames[j]);

}

printf("\n");

}

printf("\nTotal Page Faults = %d\n", faults);

return 0;

}

**Output:**

**Reference:**

<https://www.geeksforgeeks.org/page-replacement-algorithms-in-operating-systems/>

**Practice-4**

**Aim:** To implement FCFS (First Come First Serve) Disk Scheduling algorithm in C

**Description:**

FCFS is the simplest of all Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

**Source Code:**

**//Disk Scheduling FCFS**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

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

scanf("%d",&initial);

// logic for FCFS disk scheduling

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

printf("Total head moment is %d",TotalHeadMoment);

return 0;

}

**Output:**

Enter the number of Requests

8

Enter the Requests sequence

98 183 41 122 14 124 65 67

Enter initial head position

53

Total head moment is 632

**Practice-5**

**Aim:** To implement SSTF (Shortest Seek Time First) Disk Scheduling algorithm in C

**Description:**

In SSTF (Shortest Seek Time First), requests having the shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of the system.

**Source Code:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial,count=0;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

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

scanf("%d",&initial);

// logic for sstf disk scheduling

while(count!=n)

{

int min=1000,d,index;

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

{

d=abs(RQ[i]-initial);

if(min>d)

{

min=d;

index=i;

}

}

TotalHeadMoment=TotalHeadMoment+min;

initial=RQ[index];

// 1000 is for max

// you can use any number

RQ[index]=1000;

count++;

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**Output:**

Enter the number of Requests

8

Enter the Requests sequence

98 183 41 122 14 124 65 67

Enter initial head position

53

Total head movement is 323

**Practice-6**

**Aim:** To implement SCAN Disk Scheduling algorithm in C

**Description:**

In the SCAN algorithm the disk arm moves in a particular direction and services the requests coming in its path and after reaching the end of the disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and is hence also known as an elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

**Source Code:**

**//Disk Scheduling SCAN**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

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

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for Scan disk scheduling

/\*logic for sort the request array \*/

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

{

for(j=0;j<n-i-1;j++)

{

if(RQ[j]>RQ[j+1])

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;

}

}

// if movement is towards high value

if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

initial = size-1;

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

initial =0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**Output:**

Enter the number of Requests

8

Enter the Requests sequence

98 183 41 122 14 124 65 67

Enter initial head position

53

Enter total disk size

199

Enter the head movement direction for high 1 and for low 0

1

Total head movement is 331

**Practice-7**

**Aim:** To implement LOOK Disk Scheduling algorithm in C

**Description:**

LOOK Algorithm is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

**Source Code:**

////Disk Scheduling LOOK

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

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

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for look disk scheduling

/\*logic for sort the request array \*/

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

{

for(j=0;j<n-i-1;j++)

{

if(RQ[j]>RQ[j+1])

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

} } }

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;

}

}

// if movement is towards high value

if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**Output:**

Enter the number of Requests

8

Enter the Requests sequence

98 183 41 122 14 124 65 67

Enter initial head position

53

Enter total disk size

199

Enter the head movement direction for high 1 and for low 0

1

Total head movement is 299

**Reference: (Practice-4,5,6,7)**

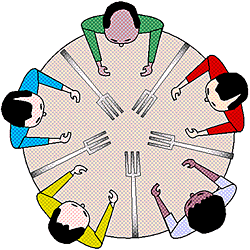
<https://www.easycodingzone.com/2021/07/c-program-of-disk-scheduling-algorithms.html>

**Dining Philosopher's Problem**

**Description:**

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.



**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <time.h>

#define NUM\_PHILOSOPHERS 5

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

void\* philosopher(void\* arg) {

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

free(arg);

while (1) {

// Think

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

int odd\_number = rand() % 10 + 1;

if (odd\_number % 2 == 0) {

printf("Philosopher %d has an even philosophical odd number: %d\n", id, odd\_number);

} else {

printf("Philosopher %d has an odd philosophical odd number: %d\n", id);

printf("The core says: No odd philosophical odd numbers allowed!\n");

}

sleep(rand() % 3 + 1);

// Become hungry

int hunger\_percentage = rand() % 100 + 1;

if (hunger\_percentage >= 50) {

printf("Philosopher %d is more than 50%% hungry and will not graze.\n", id);

} else {

printf("Philosopher %d is less than 50%% hungry and will graze.\n", id);

}

// Eat

pthread\_mutex\_lock(&forks[id]);

pthread\_mutex\_lock(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

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

sleep(rand() % 3 + 1);

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

pthread\_mutex\_unlock(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

pthread\_mutex\_unlock(&forks[id]);

}

return NULL;

}

int main() {

srand(time(NULL));

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int i;

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

int\* id = malloc(sizeof(int));

\*id = i;

pthread\_create(&philosophers[i], NULL, philosopher, id);

}

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

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

}

return 0;

}