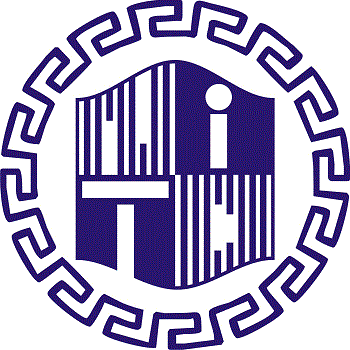
**Practical FILE**

**Session : 2019-20**

**B.Tech. 3rd Year 5th Semester**



**Core Lab – CSB 302**

**(OPERATING SYSTEM LAB)**

**Submitted To:**

***Dr. Karan Verma***

***Dept. of Computer Science & Engineering***

***NIT, Delhi***

**Submitted By:**

***Hitendra Singh***

***171210028***

***Date : 10-12-2019***

# **Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Program Title** | **Date** | **Sign** |
| **1** | * **CPU Core & Memory Specifications** * **Creating file on linux terminal** | **05-08-19** |  |
| **2** | * **Implementation of fork(), sleep(), wait() system calls.** * **Demonstration of Zombie and orphan process.** * **Implementing Merge sort in parallel using fork().** * **Bottom-Up execution of process using fork() system call.** | **19-08-19** |  |
| **3** | * **First Come First Serve Scheduling** * **Shortest Job First Scheduling** * **Shortest Remaining Job First scheduling** * **Round Robin Scheduling** | **30-08-19** |  |
| **4** | * **Priority based scheduling** * **Shortest Remaining Time First including I/O time.** | **09-09-19** |  |
| **5** | * **Synchronization using shared variables.** * **Producer-Consumer Problem** | **16-09-19** |  |
| **6** | * **Implementation of Banker’s Algorithm** | **14-10-19** |  |
| **7** | * **First-In-First-Out Page Replacement** * **Least Recently Used Page Replacement** * **Optimal Page Replacement** | **18-11-19** |  |

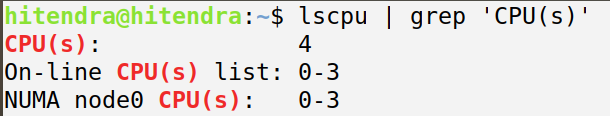
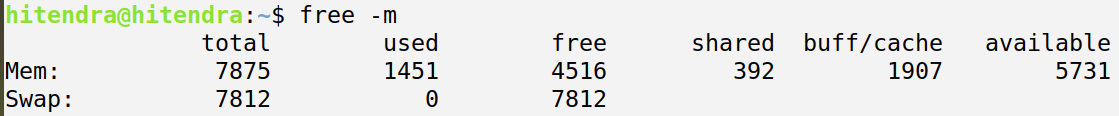
**PRACTICAL - 1**

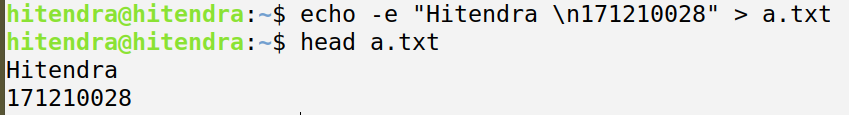
**Problem Statement :**

Answer the following queries by executing appropriate commands on terminal:

1. How many CPU cores does your machine have?
2. How much memory does your machine have and what fraction of it is free?
3. Create a text file and write your name and roll no. in separate lines using a single command.

**Output :**





**PRACTICAL - 2**

**Problem Statement :**

Write programs in C to demonstrate fork(), sleep(), and wait().

**Source Code :**

**// demonstrating fork()**

**#include <stdio.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**int main()**

**{**

**fork();**

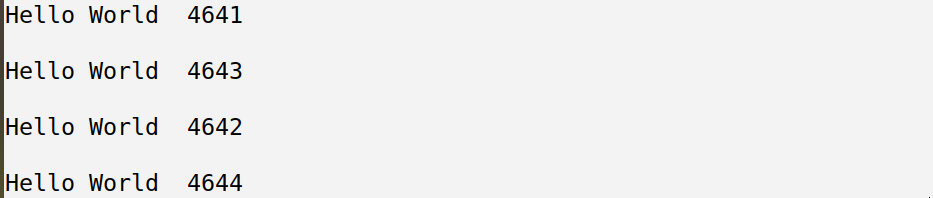
**fork();**

**printf("\nHello World %d \n",getpid());**

**return 0;**

**}**

**Output :**



**Source Code :**

**// demonstrating fork()**

**#include <stdio.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**int main()**

**{**

**int xy,c=5;**

**c = c\*5;**

**xy = fork();**

**if(xy)**

**{**

**c=c+5;**

**printf("\n %d, %d \n",c,xy);**

**}**

**else**

**{**

**c = c+50;**

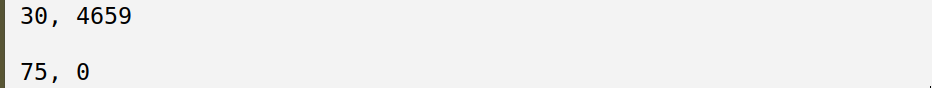
**printf("\n %d, %d \n",c,xy);**

**}**

**return 0;**

**}**

**Output :**



**Source Code :**

**// demonstrating sleep()**

**#include<stdio.h>**

**#include<sys/types.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**int main()**

**{**

**int pid = fork();**

**if(pid == -1) {**

**printf("Cannot create a process\n");**

**exit(1);**

**}**

**else if(pid == 0) {**

**sleep(5);**

**printf("Child process\n");**

**}**

**else {**

**printf("Parent Process\n");**

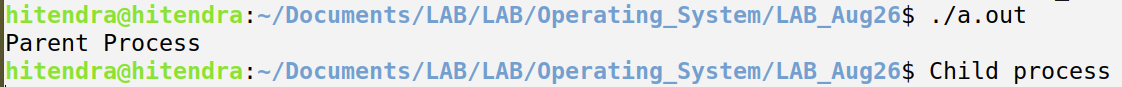
**exit(1);**

**}**

**return 0;**

**}**

**Output :**



**Source Code :**

**// demonstrating wait()**

**#include<stdio.h>**

**#include<sys/wait.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**int i = 10;**

**int main()**

**{**

**int pid = fork();**

**if(pid == 0)**

**{**

**printf("Initial value = %d\n", i);**

**i += 10;**

**printf("New Value = %d\n", i);**

**printf("Child process terminated\n");**

**}**

**else**

**{**

**wait(0);**

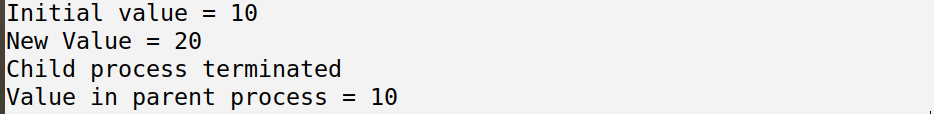
**printf("Value in parent process = %d\n", i);**

**}**

**return 0;**

**}**

**Output :**



**Problem Statement :**

Write a C program to demonstrate the zombie and orphan process.

**Source Code :**

**// Zombie Process demonstration**

**#include<stdio.h>**

**#include<sys/types.h>**

**#include<unistd.h>**

**#include<stdlib.h>**

**int main()**

**{**

**int pid = fork();**

**if(pid == 0)**

**{**

**printf("\nI am child process, and I am going to die.\nI am a zombie now ! \n");**

**exit(1);**

**}**

**else**

**{**

**sleep(10);**

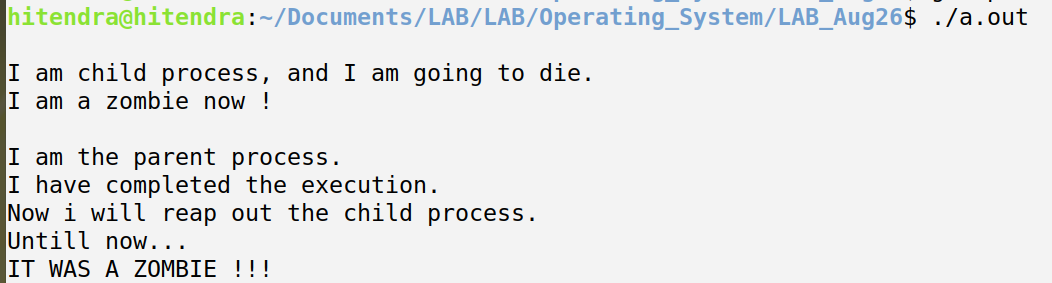
**printf("\nI am the parent process.\nI have completed the execution.\nNow i will reap out the child process.\nUntill now...\nIT WAS A ZOMBIE !!! \n");**

**}**

**return 0;**

**}**

**Output :**



**Source Code :**

**// Orphan Process demonstration**

**#include<stdio.h>**

**#include<sys/types.h>**

**#include<unistd.h>**

**#include<stdlib.h>**

**int main()**

**{**

**int pid = fork();**

**if(pid != 0)**

**{**

**printf("\nI am parent process, and I am going to die.\n\n");**

**}**

**else**

**{**

**sleep(10);**

**printf("\nI am the child process.\nMy parent died 10 seconds ago :( , but i was busy executing.\nSo, I am an orphan since a few seconds.\nWaiting for Adoption :|\n");**

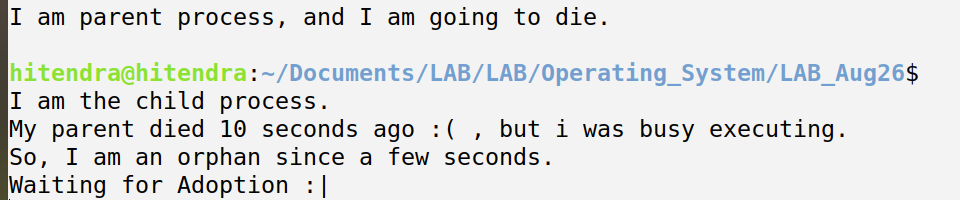
**exit(0);**

**}**

**return 0;**

**}**

**Output :**



**Problem Statement :**

Write a C program in which main program accepts the integers to be sorted. Main program uses the fork system call to create a new process called a child process. Parent process sorts the integers using merge sort and waits for child process using wait system call to sort the integers using quick sort. Also demonstrate zombie and orphan states.

**Source Code :**

**#include<stdio.h>**

**#include<sys/types.h>**

**#include<sys/wait.h>**

**#include<unistd.h>**

**int partition(int arr[],int l,int r)**

**{**

**int i,j,temp,pivot;**

**pivot=arr[l];**

**i=l;**

**j=r+1;**

**do**

**{**

**do**

**i++;**

**while(arr[i]<pivot && i<=r);**

**do**

**j--;**

**while(arr[j]>pivot);**

**if(i<j)**

**{**

**temp=arr[i];**

**arr[i]=arr[j];**

**arr[j]=temp;**

**}**

**}**

**while(i<j);**

**arr[l]=arr[j];**

**arr[j]=pivot;**

**return j;**

**}**

**void quickSort(int arr[],int l,int r)**

**{**

**int pivot;**

**if(l<r)**

**{**

**pivot = partition(arr,l,r);**

**quickSort(arr,l,pivot-1);**

**quickSort(arr,pivot+1,r);**

**}**

**}**

**void merge(int arr[],int l1,int r1,int l2,int r2)**

**{**

**int temp[51];**

**int i,j,k;**

**i=l1;**

**j=l2;**

**k=0;**

**while(i<=r1 && j<=r2)**

**{**

**if(arr[i]<arr[j])**

**temp[k++]=arr[i++];**

**else**

**temp[k++]=arr[j++];**

**}**

**while(i<=r1)**

**temp[k++]=arr[i++];**

**while(j<=r2)**

**temp[k++]=arr[j++];**

**for(i=l1,j=0;i<=r2;i++,j++)**

**arr[i]=temp[j];**

**}**

**void mergeSort(int arr[],int l,int r)**

**{**

**int m;**

**if(l<r)**

**{**

**m=(l+r)/2;**

**mergeSort(arr,l,m);**

**mergeSort(arr,m+1,r);**

**merge(arr,l,m,m+1,r);**

**}**

**}**

**int main()**

**{**

**int n,pid;**

**int arr[100]; //Max size 100**

**printf("\nEnter the number of elements :");**

**scanf("%d",&n);**

**int i;**

**printf("\nEnter the %d elements :",n);**

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

**{**

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

**}**

**pid=fork();**

**if(pid==0)**

**{**

**printf("\nI am a child process (PID=%d). I will perform quick sort. \n",pid);**

**quickSort(arr,0,n-1);**

**printf("\nSorted Array is : ");**

**int i;**

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

**printf(" %d ",arr[i]);**

**}**

**else**

**{**

**wait(0);**

**printf("\nI am a Parent process (PID=%d). I was waiting for my child process. Now i will execute merge sort. \n", pid);**

**mergeSort(arr,0,n-1);**

**printf("\nSorted Array is : ");**

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

**printf(" %d ",arr[i]);**

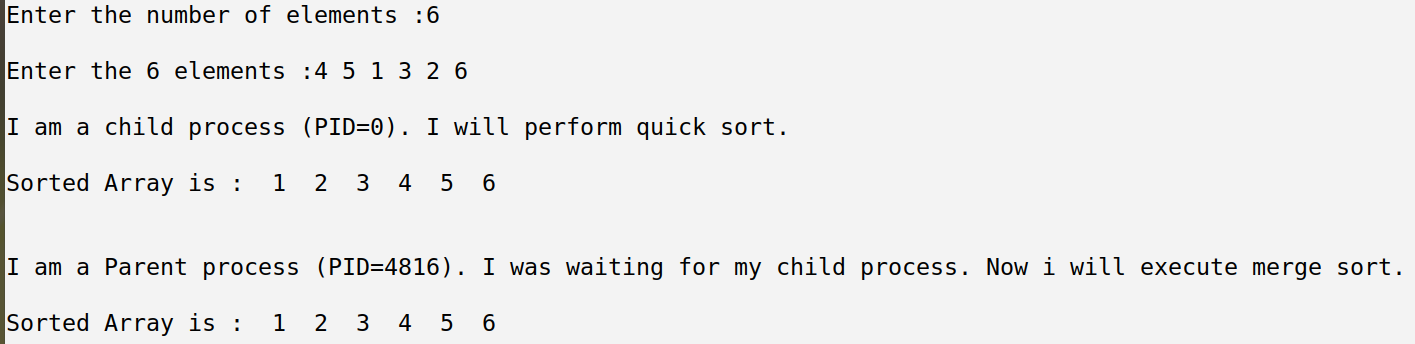
**}**

**printf("\n\n");**

**return 0;**

**}**

**Output :**



**Problem Statement :**

Write a C program to execute processes from bottom up which are created using fork () system call?

**Source Code :**

**// Zombie Process demonstration**

**#include<stdio.h>**

**#include<sys/wait.h>**

**#include<sys/types.h>**

**#include<stdlib.h>**

**#include<unistd.h>**

**int main()**

**{**

**fork();**

**fork();**

**int pid = fork();**

**if(pid==0)**

**{**

**printf("\n Child #%d\n",pid);**

**}**

**else**

**{**

**wait(0);**

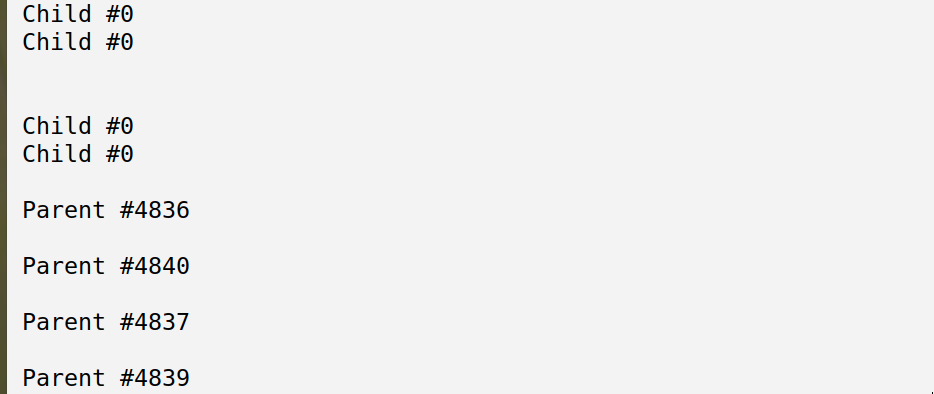
**printf("\n Parent #%d\n",pid);**

**}**

**return 0;**

**}**

**Output :**



**PRACTICAL - 3**

**Problem Statement :**

Write a program to implement the First Come First Serve CPU scheduling algorithms.

**Source Code :**

**#include <bits/stdc++.h>**

**using namespace std;**

**struct process**

**{**

**int pid,burst\_time,arrival\_time;**

**bool operator<(const process& p) const**

**{**

**return this->arrival\_time < p.arrival\_time;**

**}**

**};**

**int main()**

**{**

**int i,pno;**

**cout<<"\n Enter number of processes : ";**

**cin>>pno;**

**set<struct process> p;**

**struct process temp;**

**cout<<"\n Enter the details for "<<pno<<" processes <id arrival\_time burst\_time> : \n";**

**for(i=0; i<pno; i++)**

**{**

**cout<<"\n Process "<<i+1<<" : ";**

**cin>>temp.pid>>temp.arrival\_time>>temp.burst\_time;**

**p.insert(temp);**

**}**

**set<struct process>::iterator it;**

**float wt,tat;**

**int st=0;**

**wt=tat=0.0;**

**int waiting\_time, turn\_around\_time, completion\_time;**

**cout <<"\n\n"<<setw(5)<<"PID"<<setw(5)<<"AT"<<setw(5)<<"BT"<<setw(5)<<"CT"<<setw(5)<<"TAT"<<setw(5)<<"WT"<< endl;**

**for (it = p.begin(); it != p.end(); it++)**

**{**

**waiting\_time = max((st - (\*it).arrival\_time),0);**

**turn\_around\_time = (\*it).burst\_time + waiting\_time;**

**completion\_time = turn\_around\_time + (\*it).arrival\_time;**

**st += (\*it).burst\_time;**

**wt+=waiting\_time;**

**tat+=turn\_around\_time;**

**cout <<"\n"<<setw(5)<<(\*it).pid<<setw(5)<<(\*it).arrival\_time<<setw(5)<<(\*it).burst\_time<<setw(5)<<completion\_time<<setw(5)<<turn\_around\_time<<setw(5)<<waiting\_time<< endl;**

**}**

**float avg\_wt,avg\_tat;**

**avg\_wt = wt/pno;**

**avg\_tat = tat/pno;**

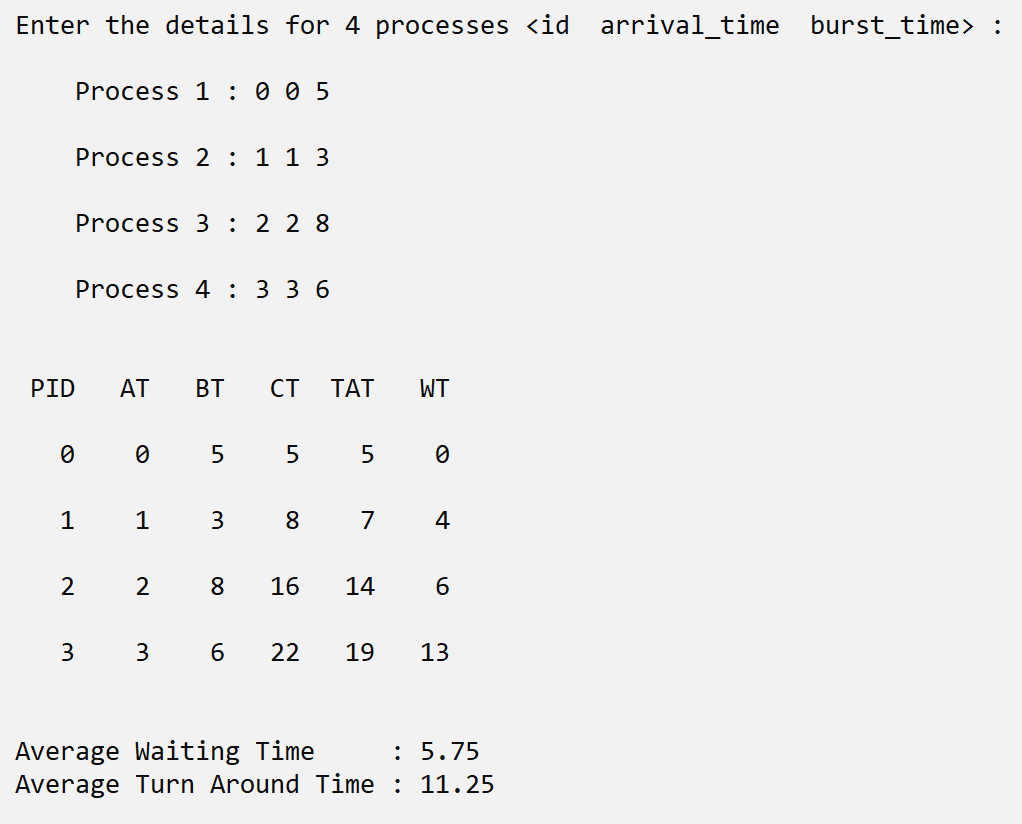
**cout<<"\n\n Average Waiting Time : "<<avg\_wt;**

**cout<<"\n Average Turn Around Time : "<<avg\_tat;**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**

**Problem Statement :**

Write a program to implement the Shortest Job First CPU scheduling algorithms.

**Source Code :**

**#include <bits/stdc++.h>**

**using namespace std;**

**struct process**

**{**

**int pid,burst\_time,arrival\_time;**

**bool done;**

**bool operator<(const process& p) const**

**{**

**return this->burst\_time < p.burst\_time;**

**}**

**};**

**int main()**

**{**

**int pno,nop;**

**cout<<"\n Enter number of processes : ";**

**cin>>pno;**

**nop=pno;**

**vector<struct process> job;**

**set<struct process> p;**

**struct process temp;**

**cout<<"\n Enter the details for "<<pno<<" processes <id arrival\_time burst\_time> : \n";**

**for(int i=0; i<pno; i++)**

**{**

**cout<<"\n Process "<<i+1<<" : ";**

**cin>>temp.pid>>temp.arrival\_time>>temp.burst\_time;**

**job.push\_back(temp);**

**}**

**set<struct process>::iterator it;**

**vector<struct process>::iterator i;**

**float wt,tat;**

**wt=tat=0.0;**

**int waiting\_time, turn\_around\_time, completion\_time;**

**cout <<"\n\n"<<setw(5)<<"PID"<<setw(5)<<"AT"<<setw(5)<<"BT"<<setw(5)<<"CT"<<setw(5)<<"TAT"<<setw(5)<<"WT"<< endl;**

**int j=0,t;**

**while(pno>0)**

**{**

**for (i = job.begin(); i != job.end(); i++)**

**{**

**if( (\*i).arrival\_time <= j && !(\*i).done)**

**{**

**p.insert((\*i));**

**(\*i).done = true;**

**}**

**}**

**it=p.begin();**

**temp=(\*it);**

**p.erase(it);**

**j+=temp.burst\_time;**

**pno--;**

**completion\_time = j;**

**turn\_around\_time = completion\_time - temp.arrival\_time;**

**waiting\_time = turn\_around\_time - temp.burst\_time;**

**wt+=waiting\_time;**

**tat+=turn\_around\_time;**

**cout <<"\n"<<setw(5)<<temp.pid<<setw(5)<<temp.arrival\_time<<setw(5)<<temp.burst\_time<<setw(5)<<completion\_time<<setw(5)<<turn\_around\_time<<setw(5)<<waiting\_time<< endl;**

**}**

**float avg\_wt,avg\_tat;**

**avg\_wt = wt / nop;**

**avg\_tat = tat / nop;**

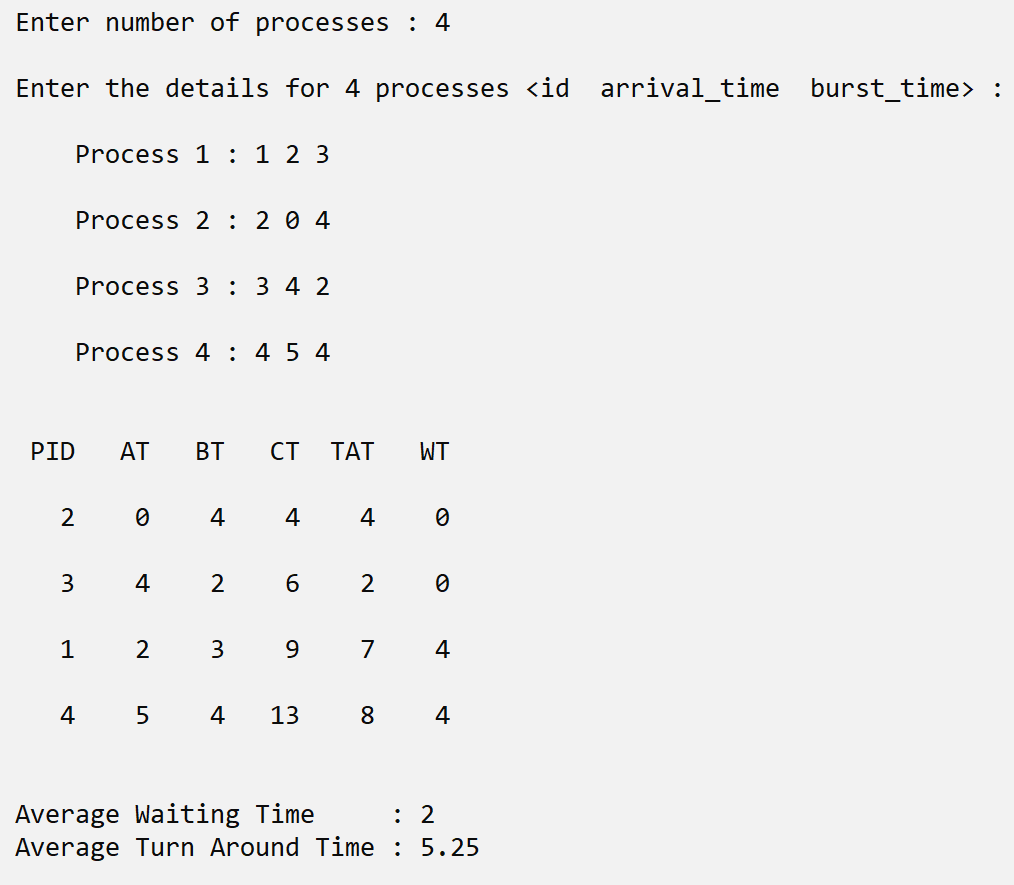
**cout<<"\n\n Average Waiting Time : "<<avg\_wt;**

**cout<<"\n Average Turn Around Time : "<<avg\_tat;**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**

**Problem Statement :**

Write a program to implement the Shortest Remaining Job First CPU scheduling algorithms.

**Source Code :**

**#include <bits/stdc++.h>**

**using namespace std;**

**struct process**

**{**

**int pid,burst\_time,arrival\_time,rt;**

**bool done;**

**bool operator<(const process& p) const**

**{**

**return this->rt < p.rt;**

**}**

**};**

**int main()**

**{**

**int pno,nop;**

**cout<<"\n Enter number of processes : ";**

**cin>>pno;**

**nop=pno;**

**vector<struct process> job;**

**set<struct process> p;**

**struct process temp;**

**cout<<"\n Enter the details for "<<pno<<" processes <id arrival\_time burst\_time> : \n";**

**for(int i=0; i<pno; i++)**

**{**

**cout<<"\n Process "<<i+1<<" : ";**

**cin>>temp.pid>>temp.arrival\_time>>temp.burst\_time;**

**temp.rt = temp.burst\_time;**

**job.push\_back(temp);**

**}**

**set<struct process>::iterator it;**

**vector<struct process>::iterator i;**

**float wt,tat;**

**wt=tat=0.0;**

**int waiting\_time, turn\_around\_time, completion\_time;**

**cout<<"\n\n"<<setw(5)<<"PID"<<setw(5)<<"AT"<<setw(5)<<"BT"<<setw(5)<<"CT"<<setw(5)<<"TAT"<<setw(5)<<"WT"<< endl;**

**int j=0,t;**

**while(pno>0)**

**{**

**for (i = job.begin(); i != job.end(); i++)**

**{**

**if( (\*i).arrival\_time <= j && !(\*i).done)**

**{**

**p.insert((\*i));**

**(\*i).done = true;**

**}**

**}**

**if(!p.empty())**

**{**

**it=p.begin();**

**temp=(\*it);**

**p.erase(it);**

**temp.rt--;**

**if(temp.rt <= 0)**

**{**

**pno--;**

**completion\_time = j + 1;**

**turn\_around\_time = completion\_time - temp.arrival\_time;**

**waiting\_time = turn\_around\_time - temp.burst\_time;**

**wt+=waiting\_time;**

**tat+=turn\_around\_time;**

**cout<<"\n"<<setw(5)<<temp.pid<<setw(5)<<temp.arrival\_time<<setw(5)<<temp.burst\_time<<setw(5)<<completion\_time<<setw(5)<<turn\_around\_time<<setw(5)<<waiting\_time<< endl;**

**}**

**else**

**p.insert(temp);**

**}**

**j++;**

**}**

**float avg\_wt,avg\_tat;**

**avg\_wt = wt / nop; avg\_tat = tat / nop;**

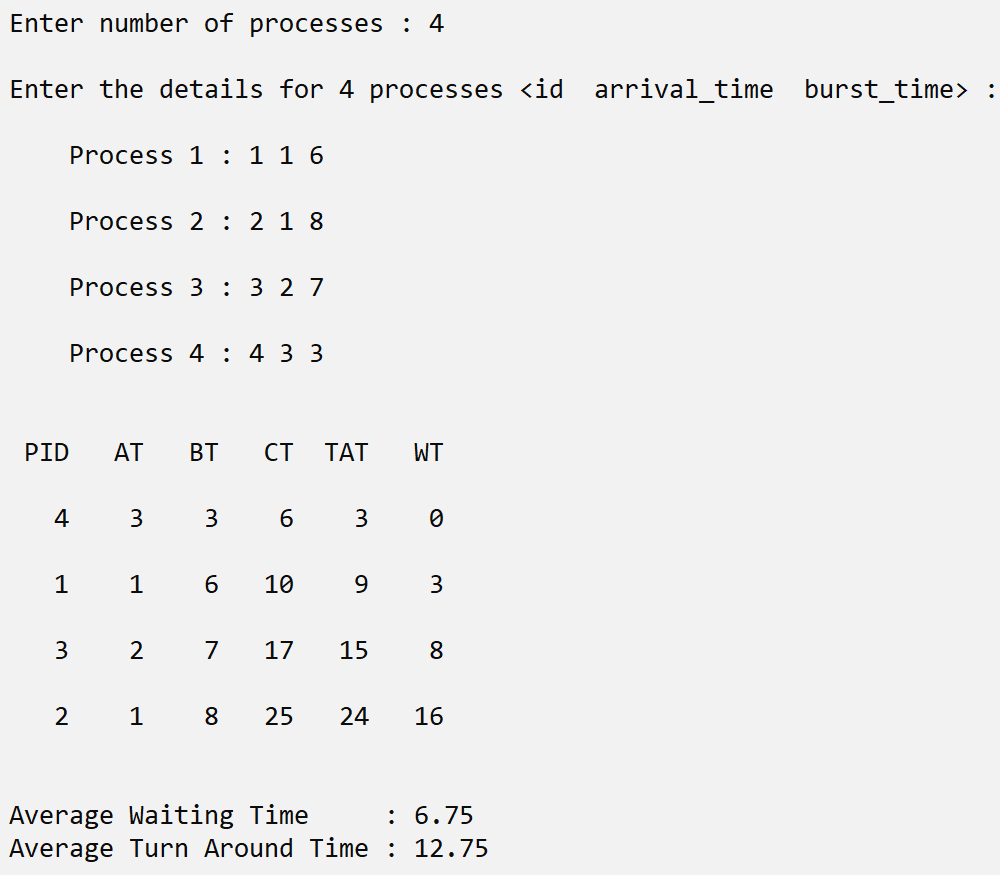
**cout<<"\n\n Average Waiting Time : "<<avg\_wt;**

**cout<<"\n Average Turn Around Time : "<<avg\_tat;**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**

**Problem Statement :**

Write a program to implement the Round Robin CPU scheduling algorithms with time slice = 2.

**Source Code :**

**#include <bits/stdc++.h>**

**using namespace std;**

**struct process**

**{**

**int pid,burst\_time,arrival\_time,rt;**

**bool done;**

**};**

**int main()**

**{**

**int i,pno,nop,q = 2;**

**cout<<"\n Enter number of processes : ";**

**cin>>pno;**

**nop = pno;**

**vector<struct process> p;**

**struct process temp;**

**cout<<"\n Enter the details for "<<pno<<" processes <id arrival\_time burst\_time> : \n";**

**for(i=0; i<pno; i++)**

**{**

**cout<<"\n Process "<<i+1<<" : ";**

**cin>>temp.pid>>temp.arrival\_time>>temp.burst\_time;**

**temp.rt= temp.burst\_time;**

**p.push\_back(temp);**

**}**

**int it=0;**

**float wt,tat;**

**int st=0;**

**wt=tat=0.0;**

**int waiting\_time, turn\_around\_time, completion\_time,cs=0; cout<<"\n\n"<<setw(5)<<"PID"<<setw(5)<<"AT"<<setw(5)<<"BT"<<setw(5)<<"CT"<<setw(5)<<"TAT"<<setw(5)<<"WT"<< endl;**

**int j=0;**

**bool flag = false;**

**while(pno>0)**

**{**

**if(p[it].rt <= q && p[it].rt > 0)**

**{**

**j+=p[it].rt;**

**p[it].rt = 0;**

**flag = true;**

**}**

**else if(p[it].rt>0)**

**{**

**j += q;**

**p[it].rt-=q;**

**}**

**if(p[it].rt==0 && flag)**

**{**

**pno--;**

**completion\_time = j;**

**turn\_around\_time = completion\_time - p[it].arrival\_time;**

**waiting\_time = turn\_around\_time - p[it].burst\_time;**

**wt+=waiting\_time;**

**tat+=turn\_around\_time;**

**cout<<"\n"<<setw(5)<<p[it].pid<<setw(5)<<p[it].arrival\_time<<setw(5)<<p[it].burst\_time<<setw(5)<<completion\_time<<setw(5)<<turn\_around\_time<<setw(5)<<waiting\_time<< endl;**

**flag = false;**

**}**

**if (it == nop-1)**

**{**

**it =0;**

**}**

**else if( p[it+1].arrival\_time <= j)**

**{**

**it++;**

**}**

**else**

**{**

**it = 0;**

**}**

**}**

**float avg\_wt,avg\_tat;**

**avg\_wt = wt/nop; avg\_tat = tat/nop;**

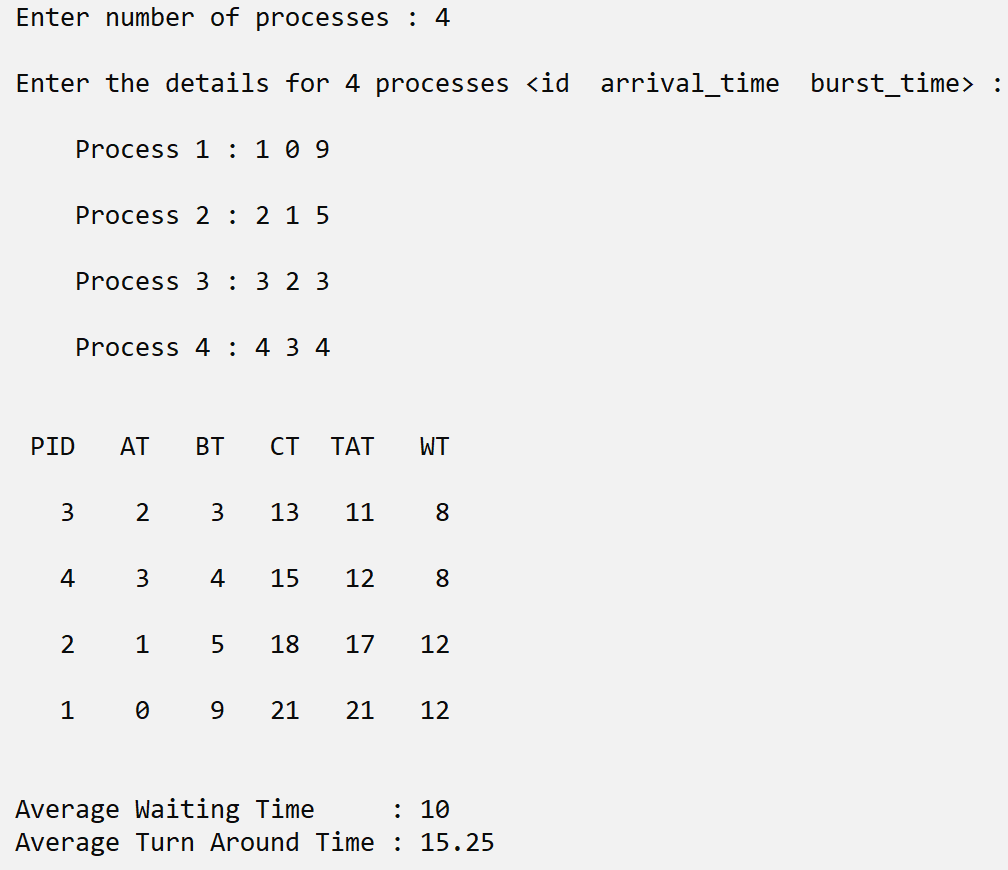
**cout<<"\n\n Average Waiting Time : "<<avg\_wt;**

**cout<<"\n Average Turn Around Time : "<<avg\_tat;**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**

**PRACTICAL - 4**

**Problem Statement :**

Write a program to implement the Priority Based CPU scheduling algorithms.

|  |  |  |  |
| --- | --- | --- | --- |
| Priority | P.No | Arrival Time | Burst Time |
| 4 | 1 | 4 | 6 |
| 7 | 2 | 6 | 3 |
| 6 | 3 | 3 | 4 |
| 6 | 4 | 2 | 2 |
| 1 | 5 | 1 | 3 |
| 3 | 6 | 2 | 2 |

**Source Code :**

**#include <bits/stdc++.h>**

**using namespace std;**

**struct process**

**{**

**int pid,burst\_time,arrival\_time,rt,priority;**

**bool done;**

**bool operator<(const process& p) const**

**{**

**if(this->priority == p.priority)**

**{**

**return this->arrival\_time < p.arrival\_time;**

**}**

**return this->priority < p.priority;**

**}**

**};**

**int main()**

**{**

**int pno,nop;**

**cout<<"\n Enter number of processes : ";**

**cin>>pno;**

**nop=pno;**

**vector<struct process> job;**

**set<struct process> p;**

**struct process temp;**

**cout<<"\n Enter the details for "<<pno<<" processes <id priority arrival\_time burst\_time> : \n";**

**for(int i=0; i<pno; i++)**

**{**

**cout<<"\n Process "<<i+1<<" : ";**

**cin>>temp.pid>>temp.priority>>temp.arrival\_time>>temp.burst\_time;**

**temp.rt = temp.burst\_time;**

**job.push\_back(temp);**

**}**

**set<struct process>::iterator it;**

**vector<struct process>::iterator i;**

**float wt,tat;**

**wt=tat=0.0;**

**int waiting\_time, turn\_around\_time, completion\_time;**

**cout<<"\n\n"<<setw(5)<<"PID"<<setw(5)<<"PR"<<setw(5)<<"AT"<<setw(5)<<"BT"<<setw(5)<<"CT"<<setw(5)<<"TAT"<<setw(5)<<"WT"<< endl;**

**int j=0,t;**

**while(pno>0)**

**{**

**for (i = job.begin(); i != job.end(); i++)**

**{**

**if( (\*i).arrival\_time <= j && !(\*i).done)**

**{**

**p.insert((\*i));**

**(\*i).done = true;**

**}**

**}**

**if(!p.empty())**

**{**

**it=p.begin();**

**temp=(\*it);**

**p.erase(it);**

**temp.rt--;**

**if(temp.rt <= 0)**

**{**

**pno--;**

**completion\_time = j;**

**turn\_around\_time = completion\_time - temp.arrival\_time;**

**waiting\_time = turn\_around\_time - temp.burst\_time;**

**wt+=waiting\_time;**

**tat+=turn\_around\_time;**

**cout<<"\n"<<setw(5)<<temp.pid<<setw(5)<<temp.priority<<setw(5)<<temp.arrival\_time<<setw(5)<<temp.burst\_time<<setw(5)<<completion\_time<<setw(5)<<turn\_around\_time<<setw(5)<<waiting\_time<< endl;**

**}**

**else**

**p.insert(temp);**

**}**

**j++;**

**}**

**float avg\_wt,avg\_tat;**

**avg\_wt = wt / nop; avg\_tat = tat / nop;**

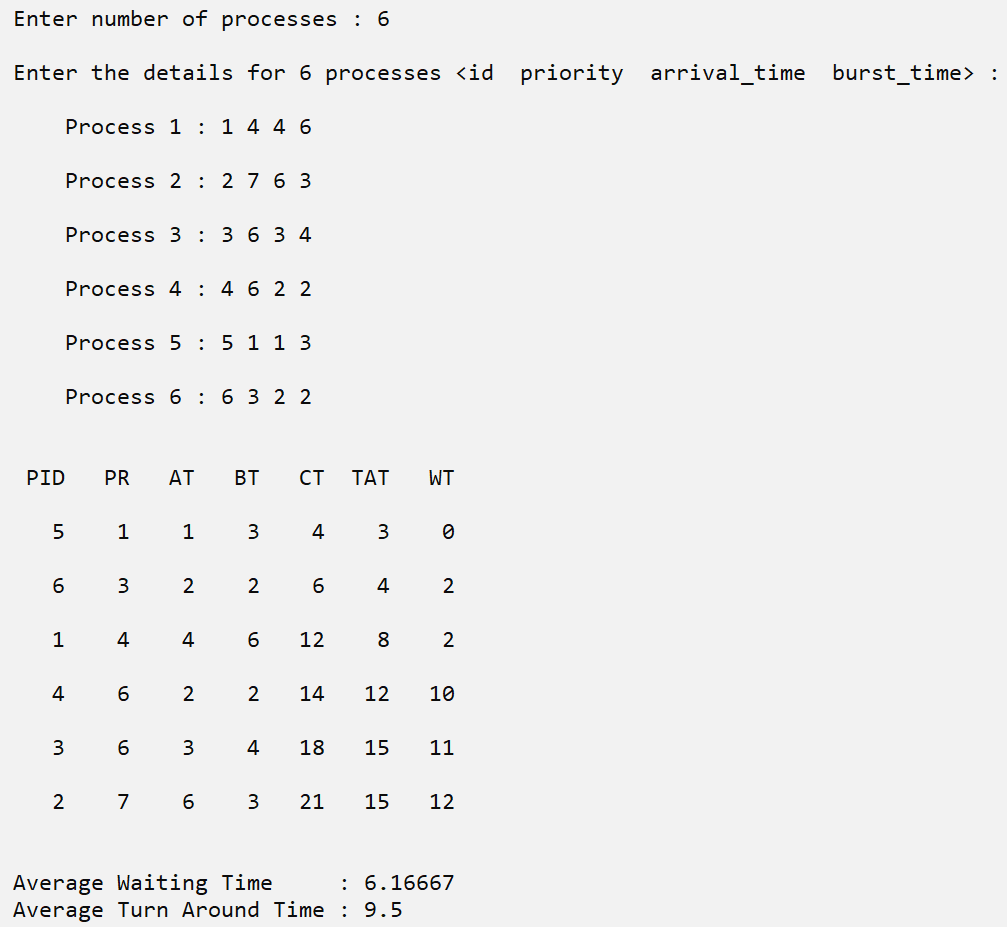
**cout<<"\n\n Average Waiting Time : "<<avg\_wt;**

**cout<<"\n Average Turn Around Time : "<<avg\_tat;**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**

**Problem Statement :**

Write a program to implement the Shortest Remaining Job First CPU scheduling algorithms considering the I/O time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P.No | Arrival Time | CPU time | I/O Time | CPU Time |
| 1 | 0 | 5 | 5 | 2 |
| 2 | 3 | 2 | 22 | 2 |
| 3 | 7 | 8 | 0 | 0 |
| 4 | 25 | 9 | 2 | 1 |

**Source Code :**

**#include <bits/stdc++.h>**

**using namespace std;**

**struct process**

**{**

**int pid,burst\_time\_1,burst\_time\_2,io\_time,arrival\_time,arrival,rt;**

**bool done,io;**

**bool operator<(const process& p) const**

**{**

**if(this->rt == p.rt)**

**return this->arrival < p.arrival;**

**return this->rt < p.rt;**

**}**

**};**

**int main()**

**{**

**int pno,nop;**

**cout<<"\n Enter number of processes : ";**

**cin>>pno;**

**nop=pno;**

**vector<struct process> job;**

**set<struct process> p;**

**struct process temp;**

**cout<<"\n Enter the details for "<<pno<<" processes <id arrival\_time burst\_time\_1 io\_time burst\_time\_2> : \n";**

**for(int i=0; i<pno; i++)**

**{**

**cout<<"\n Process "<<i+1<<" : ";**

**cin>>temp.pid>>temp.arrival\_time>>temp.burst\_time\_1>>temp.io\_time>>temp.burst\_time\_2;**

**temp.rt = temp.burst\_time\_1 + temp.burst\_time\_2;**

**temp.arrival = temp.arrival\_time;**

**temp.done = temp.io = false;**

**job.push\_back(temp);**

**}**

**set<struct process>::iterator it;**

**vector<struct process>::iterator i;**

**float wt,tat;**

**wt=tat=0.0;**

**int waiting\_time, turn\_around\_time, completion\_time;**

**cout <<"\n\n"<<setw(5)<<"PID"<<setw(5)<<"AT"<<setw(5)<<"BT1"<<setw(5)<<"IO"<<setw(5)<<"BT2"<<setw(5)<<"CT"<<setw(5)<<"TAT"<<setw(5)<<"WT"<< endl;**

**int j=0,t;**

**while(pno>0)**

**{**

**for (i = job.begin(); i != job.end(); i++)**

**{**

**if( (\*i).arrival\_time == j && !(\*i).done)**

**{**

**p.insert((\*i));**

**(\*i).done = true;**

**}**

**}**

**if(!p.empty())**

**{**

**it=p.begin();**

**temp=(\*it);**

**p.erase(it);**

**temp.rt--;**

**if(temp.rt <= 0)**

**{**

**pno--;**

**completion\_time = j + 1;**

**turn\_around\_time = completion\_time - temp.arrival;**

**waiting\_time = turn\_around\_time - (temp.burst\_time\_1+temp.burst\_time\_2);**

**wt+=waiting\_time;**

**tat+=turn\_around\_time;**

**cout <<"\n"<<setw(5)<<temp.pid<<setw(5)<<temp.arrival\_time<<setw(5)<<temp.burst\_time\_1<<setw(5)<<temp.io\_time<<setw(5)<<temp.burst\_time\_2<<setw(5)<<completion\_time<<setw(5)<<turn\_around\_time<<setw(5)<<waiting\_time<< endl;**

**}**

**else if(temp.rt <= temp.burst\_time\_2 && !temp.io)**

**{**

**for (i = job.begin(); i != job.end(); i++)**

**{**

**if( (\*i).pid == temp.pid)**

**{**

**(\*i).done = false;**

**(\*i).io = true;**

**(\*i).arrival\_time = j + 1 + (\*i).io\_time;**

**(\*i).rt = temp.rt;**

**}**

**}**

**}**

**else**

**p.insert(temp);**

**j++;**

**}**

**else**

**j++;**

**}**

**float avg\_wt,avg\_tat;**

**avg\_wt = wt / nop;**

**avg\_tat = tat / nop;**

**cout<<"\n\n Average Waiting Time : "<<avg\_wt;**

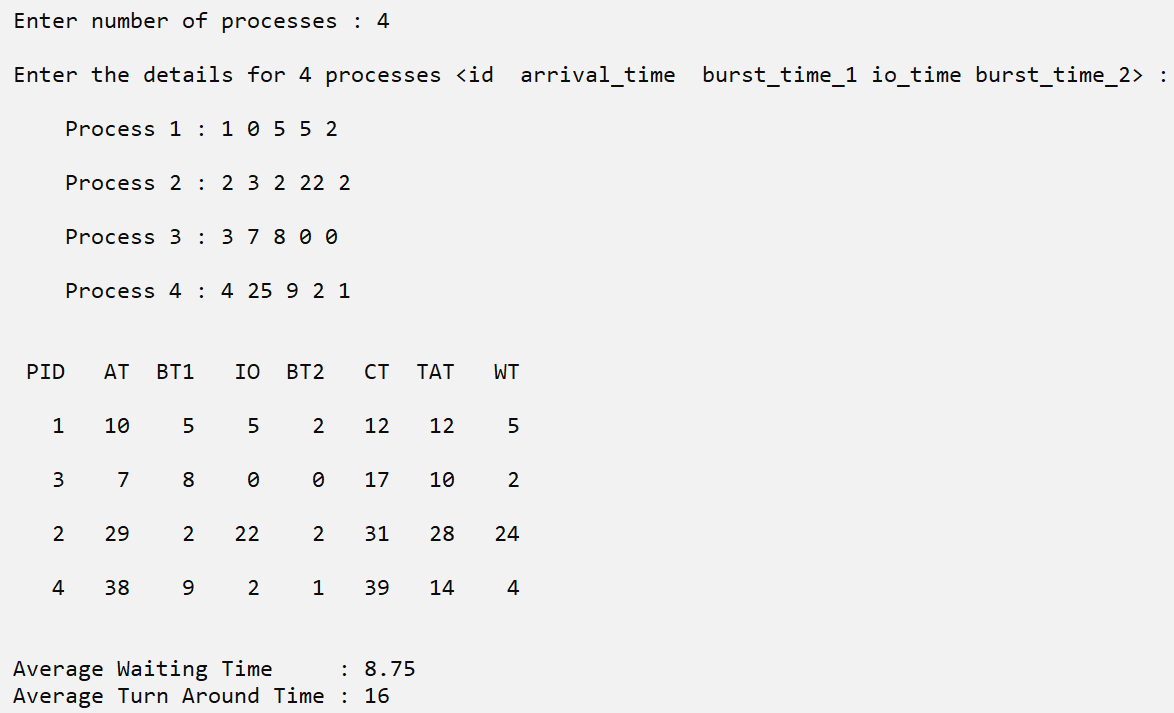
**cout<<"\n Average Turn Around Time : "<<avg\_tat;**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**



**PRACTICAL - 5**

**Problem Statement :**

Implement Readers-Writers Problem where one reader and one writer are trying to access the shared variable containing some data and eliminate the problem of synchronization.

**Source Code :**

**#include <stdio.h>**

**#include <sys/ipc.h>**

**#include <sys/shm.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**#include<sys/wait.h>**

**int main()**

**{**

**int shmid, \*a, \*b, i, lockid, \*la, \*lb;**

**shmid = shmget(IPC\_PRIVATE, 2\*sizeof(int), 0777|IPC\_CREAT);**

**lockid = shmget(IPC\_PRIVATE, sizeof(int), 0777|IPC\_CREAT);**

**if (fork() == 0)**

**{**

**b = (int \*) shmat(shmid, 0, 0);**

**lb = (int \*) shmat(lockid, 0, 0);**

**\*lb = 0;**

**for( i=0; i< 10; i++)**

**{**

**while(\*lb != 1);**

**\*lb = 1;**

**printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);**

**\*lb = 0;**

**}**

**shmdt(b);**

**}**

**else**

**{**

**a = (int \*) shmat(shmid, 0, 0);**

**la = (int \*) shmat(lockid, 0, 0);**

**\*la = 1;**

**a[0] = 0;**

**a[1] = 1;**

**for( i=0; i< 10; i++)**

**{**

**while(\*la != 0);**

**\*la = 0;**

**a[0] = a[0] + a[1];**

**a[1] = a[0] + a[1];**

**printf("Parent writes: %d,%d\n",a[0],a[1]);**

**\*la = 1;**

**}**

**wait(0);**

**shmdt(a);**

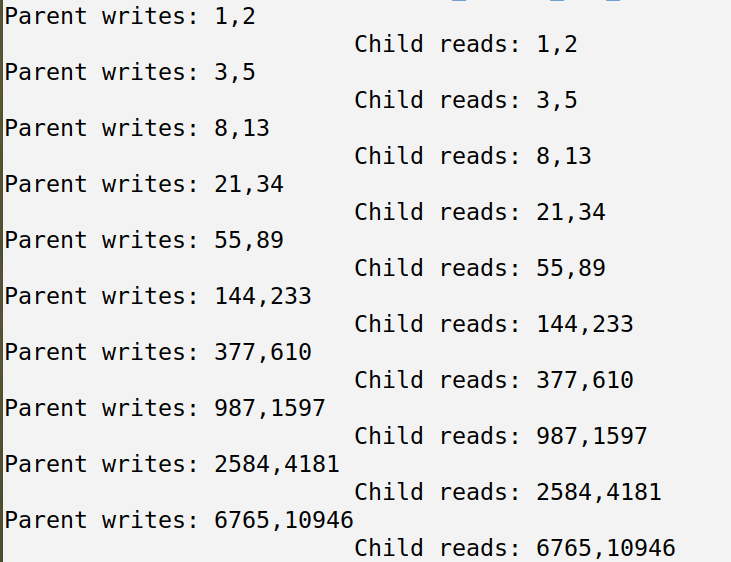
**shmctl(shmid, IPC\_RMID, 0);**

**}**

**return 0;**

**}**

**Output :**



**Problem Statement :**

We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section. Implement the problem using given pseudocode.

**Source Code :**

**#include <stdio.h>**

**#include <sys/ipc.h>**

**#include <sys/shm.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int s;**

**int main()**

**{**

**int shmid,shm, \*p, \*c, i;**

**printf("\n Enter the buffer size : ");**

**scanf("%d",&s);**

**shmid = shmget(IPC\_PRIVATE, (s+2)\*sizeof(int), 0777|IPC\_CREAT);**

**if (fork() == 0)**

**{**

**c = (int \*) shmat(shmid, 0, 0);**

**while(1)**

**{**

**while (c[s] == c[s+1]);**

**printf("\n Item %d is consumed : ",c[c[s+1]]);**

**c[c[s+1]] = 0;**

**for(i=0; i<s; i++)**

**printf(" %d",c[i]);**

**c[1+s] = (c[1+s] + 1) % s;**

**sleep(1);**

**};**

**shmdt(c);**

**}**

**else**

**{**

**p = (int \*) shmat(shmid, 0, 0);**

**p[s]=p[s+1]=0;**

**int np=0;**

**for(i=0; i<s; i++)**

**p[i]=0;**

**while(1)**

**{**

**while (((p[s] + 1) % s) == p[1+s]);**

**np++;**

**printf("\n Item %d is produced : ",np);**

**p[p[s]] = np;**

**for(i=0; i<s; i++)**

**printf(" %d",p[i]);**

**p[s] = (p[s] + 1) % s;**

**sleep(1);**

**}**

**shmdt(p);**

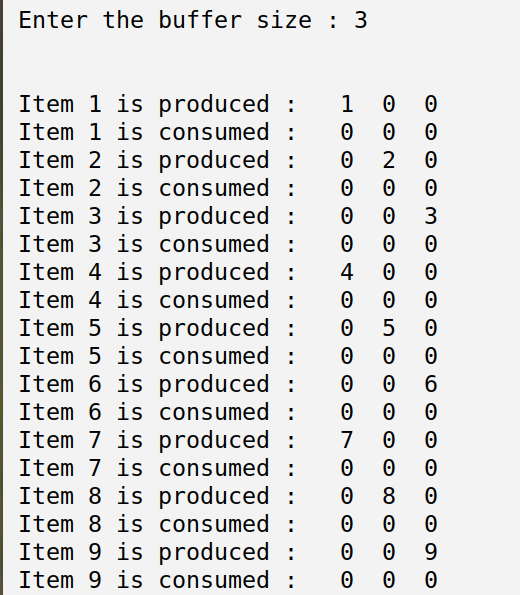
**shmctl(shmid, IPC\_RMID, 0);**

**}**

**return 0;**

**}**

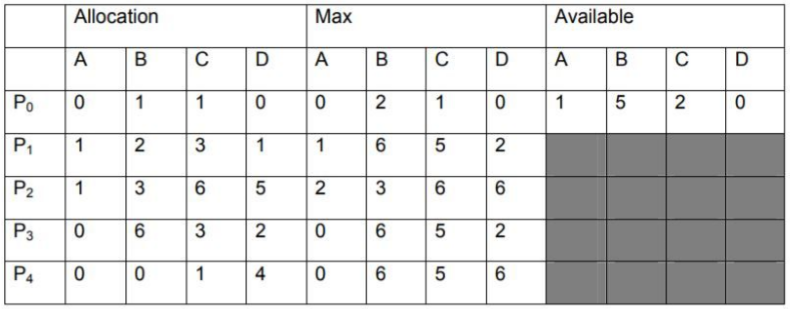
**Output :**



**PRACTICAL - 6**

**Problem Statement :**

Write a program to implement Banker’s Algorithm and find the safe sequence for following allocation.



**Source Code :**

**#include<bits/stdc++.h>**

**#define process 5**

**#define resources 4**

**using namespace std;**

**bool is\_available(int available[],int need[][resources],int p)**

**{**

**bool flag = true;**

**for(int i=0; i<resources; i++)**

**flag = flag && (available[i]>=need[p][i]);**

**return flag;**

**}**

**void PrintSafeSequence(vector<int> safe,bool completed[],int available[],int allocated[][resources],int need[][resources])**

**{**

**for(int i=0; i<process; i++)**

**{**

**if(!completed[i] && is\_available(available,need,i))**

**{**

**completed[i]=true;**

**for(int j=0; j<resources; j++)**

**{**

**available[j]+=allocated[i][j];**

**}**

**safe.push\_back(i);**

**PrintSafeSequence(safe,completed,available,allocated,need);**

**safe.pop\_back();**

**for(int j=0; j<resources; j++)**

**{**

**available[j]-=allocated[i][j];**

**}**

**completed[i]=false;**

**}**

**}**

**if(safe.size()==process)**

**{**

**cout<<"\n --> ";**

**for(int i=0; i<process; i++)**

**{**

**cout<<" "<<safe[i];**

**}**

**}**

**}**

**int main()**

**{**

**int i,j;**

**int allocated[process][resources]= {{0,1,1,0},{1,2,3,1},{1,3,6,5},{0,6,3,2},{0,0,1,4}};**

**int Max[process][resources]= {{0,2,1,0},{1,6,5,2},{2,3,6,6},{0,6,5,2},{0,6,5,6}};**

**int available[resources]= {1,5,2,0};**

**int need[process][resources]= {0};**

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

**{**

**for(j=0; j<resources; j++)**

**{**

**need[i][j] = Max[i][j]-allocated[i][j];**

**}**

**}**

**bool completed[process];**

**vector<int> safe;**

**memset(completed,false,sizeof(completed));**

**cout<<"\n The safe sequences are : \n";**

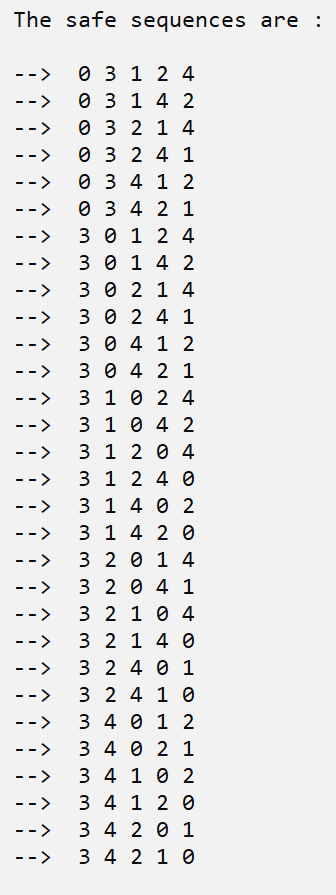
**PrintSafeSequence(safe,completed,available,allocated,need);**

**cout<<"\n\n";**

**return 0;**

**}**

**Output :**



**PRACTICAL - 7**

**Problem Statement :**

Consider the following page reference string :1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

How many page faults would occur assuming three frames for FIFO,LRU and Optimal page replacement algorithms

**Source Code :**

**// First In First Out**

**#include<bits/stdc++.h>**

**using namespace std;**

**int main()**

**{**

**int fno=3,pno=20;**

**int pages[20]= {1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6};**

**int frames[3]= {0};**

**int i,j,k=-1;**

**for(i=0; i<pno; i++)**

**{**

**for(j=0; j<fno; j++)**

**{**

**if(frames[j]==pages[i])**

**break;**

**}**

**if(j==fno)**

**{**

**k++;**

**frames[k%3]=pages[i];**

**cout<<"\n MIS ";**

**}**

**else**

**cout<<"\n HIT ";**

**cout<<" >> Page: "<<pages[i]<<" =>";**

**for(j=0; j<fno; j++)**

**{**

**cout<<" "<<frames[j];**

**}**

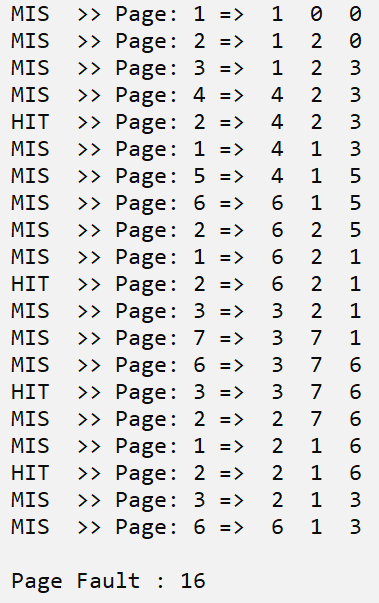
**}**

**cout<<"\n\n Page Fault : "<<k + 1<<"\n\n";**

**return 0;**

**}**

**Output :**



**Source Code :**

**// Least Recently Used**

**#include<bits/stdc++.h>**

**using namespace std;**

**int main()**

**{**

**int fno=3,pno=20;**

**int pages[20]= {1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6};**

**int frames[3]= {0};**

**list<int> f;**

**f.resize(fno,0);**

**int i,j,k = 0,lru;**

**for(i=0; i<pno; i++)**

**{**

**if(find(f.begin(),f.end(),pages[i])!=f.end())**

**{**

**cout<<"\n HIT ";**

**f.remove(pages[i]);**

**f.push\_back(pages[i]);**

**}**

**else**

**{**

**cout<<"\n MIS ";**

**k++;**

**lru = f.front();**

**f.pop\_front();**

**f.push\_back(pages[i]);**

**for(j=0; j<fno; j++)**

**{**

**if(lru == frames[j])**

**{**

**frames[j] = pages[i];**

**break;**

**}**

**}**

**}**

**cout<<" >> Page: "<<pages[i]<<" =>";**

**for(j=0; j<fno; j++)**

**cout<<" "<<frames[j];**

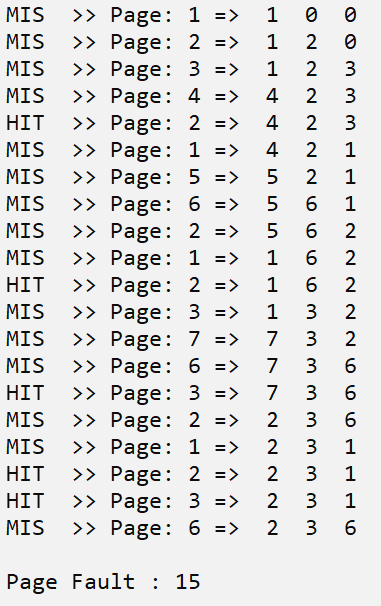
**}**

**cout<<"\n\n Page Fault : "<<k<<"\n\n";**

**return 0;**

**}**

**Output :**



**Source Code :**

**// Optimal Page Replacement**

**#include<bits/stdc++.h>**

**using namespace std;**

**int main()**

**{**

**int fno=3,pno=20,i,j,k = 0,opt,f;**

**int pages[20]= {1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6},frames[3]= {0};**

**for(i=0; i<pno; i++)**

**{**

**if(find(frames,frames+3,pages[i])!=frames+3)**

**cout<<"\n HIT ";**

**else**

**{**

**cout<<"\n MIS ";**

**k++;**

**f=INT\_MIN;**

**for(j=0; j<fno; j++)**

**{**

**if(find(pages+i,pages+20,frames[j]) != pages+20)**

**f = max(f,find(pages+i,pages+20,frames[j])-pages);**

**else**

**{**

**opt = frames[j];**

**break;**

**}**

**opt = pages[f];**

**}**

**for(j=0; j<fno; j++)**

**{**

**if(opt == frames[j])**

**{**

**frames[j] = pages[i];**

**break;**

**}**

**}**

**}**

**cout<<" >> Page: "<<pages[i]<<" =>";**

**for(j=0; j<fno; j++)**

**cout<<" "<<frames[j];**

**}**

**cout<<"\n\n Page Fault : "<<k<<"\n\n";**

**return 0;**

**}**

**Output :**

