***A***

***Laboratory File***

***On***

**Operating System**

***Submitted***

***For***

**Bachelor of Technology**

***In***

**Computer Science and Engineering**

***At***



**Bundelkhand Institute of Engineering and Technology**

**Jhansi (UP)**

# 2020-21

**Submitted To: Submitted By:**

Er. Hemant Kushwaha Ankush Kumar Kushwaha

Faculty CSE - 2nd Year (Even Sem)

BIET, Jhansi Roll No.: 2004310015

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp No.** | **Experiment Name** | **Page No.** | **Signature** |
| 1 | Study of hardware and software requirements of different operating systems (UNIX, LINUX,  WINDOWS XP, WINDOWS7/8 | 1-2 |  |
| 2 | Execute various UNIX system calls for i. Process management ii. File management iii. Input/output Systems calls | 3-8 |  |
| 3 | Implement CPU Scheduling Policies: i. SJF ii. Priority iii. FCFS iv. Multi-level Queue | 9-16 |  |
| 4 | Implement file storage allocation technique:   1. Contiguous (using array) 2. Linked –list (using linked-list) iii. Indirect allocation (indexing) | 17-23 |  |
| 5 | Implementation of contiguous allocation techniques: i. Worst-Fit ii. Best- Fit iii. First- Fit | 24-31 |  |
| 6 | Calculation of external and internal fragmentation  i. Free space list of blocks from system ii. List  process file from the system | 32 |  |
| 7 | Implementation of compaction for the continually changing memory layout and calculate total  movement of data | 33-34 |  |
| 8 | Implementation of resource allocation graph  RAG) | 35-36 |  |
| 9 | Implementation of Banker’s algorithm | 37-40 |  |
| 10 | Conversion of resource allocation graph (RAG) to wait for graph (WFG) for each type of method used for storing graph. | 41-42 |  |
| 11 | Implement the solution for Bounded Buffer  (producer-consumer) problem using inter process communication techniques-Semaphores | 43-47 |  |
| 12 | Implement the solutions for Readers-Writer’s problem using inter process communication | 48-50 |  |
|  | technique -Semaphore |  |  |

## EXPERIMENT :- 01

***Objective****:* Study of hardware and software requirements of different operating systems(UNIX, LINUX, WINDOWS XP, WINDOWS7/8.

***Content:***

Hardware requirements: The most common set of requirements defined by any operating system or software application is the physical computer resources, also known as hardware. A hardware requirements list is often accompanied by a hardware compatibility list(HCL), especially in case of operating systems.

**Hardware and Software Minimum Requirements**

**Windows 10**

* Processor: 1 gigahertz (GHz) or faster processor or SoC
* RAM: 1 Gigabyte(GB) for 32 bit or 2GB for 64 bit
* Hard disk space: 16 GB for 32bit OS or 20 GB for 64bit OS
* Graphics card: DirectX 9 or later with WDDM 1.0 driver
* Display: 800x6009 with WDDM driver

**WINDOWS XP**

The minimum hardware requirements for windows XP Home Edition are:

* Pentium 233 megahertz(MHz) processor or faster (300MHz is recommended)
* At least 64 megabytes (MB) of RAM (128MB is recommended)
* At least 1.5 gigabytes (GB) of available space on the hard disk
* CD-ROM or DVD-ROM drive
* Keyboard and a Microsoft Mouse or some other compatible pointing device
* Video adapter and monitor with Super VGA(800x600) or higher resolution
* Sound card
* Speakers or headphones

**UNIX OS**

* RAM:2GB
* Processor: IBM 604e processor with aclock speed of 375 MHz or faster
* Free disk space: /tmp must have 1GB free disk space. If Tivoli Identity Manager installs WebSphere Application Server, {WAS\_HOME} must have 800 MB free disk space and /var must have 300MB free disk space. Allocate 500 MB for /itim45
* **LINUX**
* 32 bit Intel compatible processor running at 2GHz or greater
* 512 MB RAM
* Disk space: 2.5 GB for Pipeline Pilot server plus components

## EXPERIMENT :- 02

***Objective:*** Execute various UNIX system calls for

1. Process management
2. File management
3. Input/Output Systems calls

***Content:***

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers.

***Unix System Calls***

System calls in Unix are used for file system control, process control, interprocess communication etc. Access to the Unix kernel is only available through these system calls. Generally, system calls are similar to function calls, the only difference is that they remove the control from the user process. There are around 80 system calls in the Unix interface currently. Details about some of the important ones are given as follows-

**System Call Description**  access() This checks if a calling process has access to the required file chdir() The chdir command changes the current directory of the system chmod() The mode of a file can be changed using this command chown() This changes the ownership of a particular file kill() This system call sends kill signal to one or more processes link() A new file name is linked to an existing file using link system call open() This opens a file for the reading or writing process pause() The pause call suspends a file until a particular signal occurs stime() This system call sets the correct time times() Gets the parent and child process times alarm() The alarm system call sets the alarm clock of a process folk() Anew process is created using this command

chroot() This changes the root directory of a file exit() This exit system call is used to exita process

File Structure Related calls Creating a Channel creat()

open()

close()

Input/Output read()

write()

Random Access lseek()

Channel Duplication dup()

Aliasing and Removing link()

Files unlink() File status stat()

fstat()

Access Control access()

chmod() chown() umask()

Device Control ioctl() Process Related Calls Process Creation and exec()

Termination fork()

wait() exit()

Process Owner and Group getuid()

gateuid()

Process Control signal()

kill()

alarm()

Change Working Directory chdir()

Interprocess Pipelines pipe()

Communication messages msgget()

msgsnd() msgrcv() msgtl()

Semaphores semget() semop()

Shared Memory shmget()

shmat() shmdt()

/\*errmsgl.c

Print all system error messages using “perror()”

\*/

#include <stdio.h> int main()

{

Int I;

extern int errno, sys\_nerr; for(i=0;i<sys\_nerr;++i);

{

fprintf(stderr,”%3d”,i); errno=I; perror(“”);

}

exit(0);

}

/\*errmsg2.c

Print all system error messages using the global massage table

\*/

#include <stdio.h> int main()

{ int i;

extern int sys\_nerr; extern char \*sys\_errlist[]; fprintf(stderr, “Here are the current %d error messages:\n\n”,sys\_nerr); for(i=0;i<sys\_nerr; ++i) fprintf(stderr,”%3d:%s\n”,I, sys\_errlist[i]);

}

/\* creat.c\*/

#include<stdio.h>

#include<sys/types.h> /\*definestypes used by sys/stat.h\*/ #include<sys/stat.h> /\*defines S\_IREAD & S\_IWRITE\*/ int main()

{

Int fd; fd=creat(“datafile.dat”,S\_Iread|S\_IWRITE); if(fd==\_1) printf(“Error in opening datafile.dat\n”); else

{

printf(“ datafile.dat opened for read/write access\n”); printf(“datafile.dat is currently empty\n”);

}

close(fd); exit(0);

}

The following is a sample of the manifest constants for the mode argument as defined in /usr/include/sys/stat.h:

#define S\_IRWXU 0000700 /\*-rwx----------\*/

#define S\_IREAD 0000400 /\*read permission, owner\*/

#define S\_IRUSR S\_IREAD

#define S\_IWRITE 0000200 /\*write permission, owner\*/

#define S\_IWUSR S\_IWRITE

#define S\_IEXEC 0000100 /\*execute/search permission, owner\*/

#define S\_IXUSR S\_IEXEC

#define S\_IRWXG 0000070 /\*----rwx----\*/

#define S\_IRGRP 0000040 /\*read permission,group\*/

#define S\_IWGRP0000020 /\*write “ “ \*/

#define S\_IXGRP 0000010 /\*execute/search” “ \*/

#define S\_IRWXO 0000007 /\*----------rwx\*/

#define S\_IROTH 0000004 /\*read permission,other\*/

#define S\_IWOTH 0000002 /\*write “ “ \*/ #define S\_IXOTH 0000001 /\*execute/search “ “ \*/ open()

Next is the open() system call. Open() lets youy open a file fpor reading,writing,or reading and writing.

The prototype for the open() system call is:

#include<fcntl.h> int open(file\_name, option\_flags[,mode]) char\*file\_name; int option\_flags, mode;

where file\_name is a pointer to the character string that names the file, option\_flags represent the type of channel, and mode defines the filesaccess permissionsif the file is being created.

The allowable option\_flags as defined in “/usr/include/fcntl.h”are:

#define O\_RDONLY 0 /\*Open the file for reading only\*/

#define O\_WRONGLY 1 /\*Open the file for writing only\*/

#define O\_RDWR 2 /\*Open the file for both reading and writing\*/

#define O\_NDELAY 04 /\*Non-blockingI/O\*/

#define O\_APPEND 010 /\*append(writes guaranteed at the end)\*/

#define O\_CREAT 00400 /\*open with file creat( uses third open arg)\*/

#define O\_TRUNC 01000 /\*open with truncation\*/

#define O\_EXCL 02000 /\*exclusiveopen \*/

## EXPERIMENT:- 03

***Objective:*** Implement CPU Scheduling Policies:

1. SJF
2. Priority
3. FCFS iv. Multi-level Queue

***Content:***

***FCFS CPU SCHEDULING ALGORITHM***

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

***SJF CPU SCHEDULING ALGORITHM***

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

***PRIORITY CPU SCHEDULING ALGORITHM***

For priority scheduling algorithm, read the number of processes/jobs in thesystem, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and the FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

***Program***

**1. FCFS SCHEDULING**

#include<stdio.h> #include<conio.h> void main()

{

int i,j,bt[10],n,wt[10],tt[10],w1=0,t1=0;

float aw,at; clrscr(); printf(“enter no. of processes:\n”); scanf(“%d”,&n); printf(“enter the burst time of processes:”); for(i=0;i<n;i++) scanf(“%d”,&bt[i]); for(i=0;i<n;i++)

{

wt[0]=0; tt[0]=bt[0]; wt[i+1]=bt[i]+wt[i]; tt[i+1]=tt[i]+bt[i+1]; w1=w1+wt[i];

t1=t1+tt[i];

}

aw=w1/n;

printf(“\nbt\t wt\t tt\n”); printf(“%d\t %d\n”,bt[i],wt[i],tt[i]); printf(“aw=%f\n,at=%f\n”,aw,at); getch();

}

**INPUT**

Enter no of processes

3

Enter burst time

12

8

20

**EXPECTED OUTPUT**

bt wt tt

12 0 12

8 12 20 20 20 40 aw=10.666670 at=24.00000

**2. SJF SCHEDULING** #include<stdio.h> #include<conio.h> void main()

{

int i,j,bt[10],t,n,wt[10],tt[10],w1=0,t1=0; float aw,at; clrscr(); printf(“enter no. of processes:\n”); scanf(“%d”,&n); printf(“enter the burst time of processes:”); for(i=0;i<n;i++) scanf(“%d”,&bt[i]); for(i=0;i<n;i++)

{

for(j=i;j<n;j++) if(bt[i]>bt[j])

{ t=bt[i]; bt[i]=bt[j]; bt[j]=t;

}

}

for(i=0;i<n;i++) printf(“%d”,bt[i]); for(i=0;i<n;i++)

{

wt[0]=0; tt[0]=bt[0]; wt[i+1]=bt[i]+wt[i]; tt[i+]=tt[i]+bt[i+1];

w1=w1+wt[i];

t1=t1+tt[i];

}

aw=w1/n;

at=t1/n; printf(“\nbt\t wt\t tt\n”); for(i=0;i<n;i++) printf(“%d\t %d\t %d\n”,bt[i],wt[i],tt[i]); printf(“aw=%f\n,at=%f\n”,aw,at); getch();

}

**INPUT**

Enter no of processes

3

Enter burst time

12

8

20

**OUTPUT**

bt wt tt

12 8 20

8 0 8 20 20 40 aw=9.33 at=22.64

3. **PRIORITY SCHEDULING**

#include<stdio.h> #include<conio.h> void main()

{

int I,j,pno[10],prior[10],bt[10],n,wt[10],tt[10],w1=0,t1=0,s; float aw,at; clrscr();

printf(“enter the number of processes:”); scanf(“%d”,&n); for(i=0;i<n;i++)

{

printf(“The process %d:\n”,i+1); printf(“Enter the burst time of processes:”); scanf(“%d”,&bt[i]); printf(“Enter the priority of processes %d:”,i+1); scanf(“%d”,&prior[i]); pno[i]=i+1;

}

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

{

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

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

{

s=prior[i]; prior[i]=prior[j]; prior[j]=s; s=bt[i]; bt[i]=bt[j]; bt[j]=s; s=pno[i]; pno[i]=pno[j]; pno[j]=s;

}

}

}

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

{

wt[0]=0; tt[0]=bt[0]; wt[i+1]=bt[i]+wt[i]; tt[i+1]=tt[i]+bt[i+1]; w1=w1+wt[i];

t1=t1+tt[i];

aw=w1/n;

at=t1/n;

}

printf(“\n job \t bt \t wt \t tat \t prior\n”); for(i=0;i<n;i++); printf(“%d \t %d \t %d\t %d\t %d\n”,pno[i],bt[i],wt[i],tt[i],prior[i]); printf(“aw=%f\t at=%f\n”,aw,at); getch();}

## EXPERIMENT :- 04

***Objective:***Implement filr storage allocation technique:

1. Contiguous (using array)
2. Linked list(using linked list)
3. Indirect allocation (indexing)

***Content***:The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

Contiguous Allocation

Linked Allocation

Indexed Allocation

The main idea behind these methods is ton provide:

Efficient disk space utilization.

Fast access to the blocks.

***Program:***

***SEQUENTIAL FILE ALLOCATION***

#include<stdio.h> main()

{ int f[50],I,st,j,len,c,k; clrscr(); for(i=0;i<50;i++) f[i]=0;

X:

printf(“\n Enter the starting block & length of file”); scanf(“%d%d”,&st,&len); for(j=st;j<(st+len);j++) if(f[j]==0)

{ f[j]=1:

printf(“\n%d->%d”,j,f[j]);

}

else

{

printf(“Block already allocated”); break;

}

if(j==(st+len)) printf(“\n the file is allocated to disk”); printf(\n if u want to enter more files?(y-1/n-0)”); scanf(“%d”,&c); if(c==1) goto X; else exit();

getch();}

**OUTPUT**

Enter the starting block & length of file of file 4 10

4->1

5->1

6->1

7->1

8->1

9->1

10->1

23

11->1

12->1

13->1

The file is allocated to disk

If you want toenter more files?(Y-1/N-0)

***LINKED FILE ALLOCATION*** #include<stdio.h> main()

{ int f[50],p,i,j,k,a,st,len,n,c; clrscr();

for(i=0;i<50;i++) f[i]=0; printf(“Enter how many blocks that are already allocated”); scanf(“%d”,&p); printf(“\nEnter the blocks no.s that are already allocated”); for(i=0;i<p;i++)

{

scanf(“%d”,&a); f[a]=1;

} X:

printf(“Enter the starting index block & length”); scanf(“%d%d”,&st,&len); k=len; for(j=st;j<(k+st);j++)

{ if(f[j]==0)

{ f[j]=1; printf(“\n%d->%d”,j,f[j]);

}

else

{

printf(“\n%d->file is already allocated”,j); k++;

}

}

printf(“\n If u want to enter one more file?(yes-1/no-0)”); scanf(“%d”,&c); if(c==1) goto X; else exit(); getch();}

***INDEXED ALLOCATION TECHNIQUE***

#include<stdio.h> int f[50],I,k,j,inde[50],n,c,count=0,p; main()

{

clrscr(); for(i=0;i<50;i++) f[i]=0; x:

printf(“enter index block\t”);

scanf(“%d”,&p); if(f[p]==0)

{

f[p]=1; printf(“enter no of fileson index\t”); scanf(“%d”,&n);

}

else

{

Printf(“Block already allocated\n”); goto x;

}

for(i=0;i<n;i++) scanf(“%d”,&inde[i]); for(i=0;i<n;i++) if(f[inde[i]]==1)

{

printf(“Block already allocated”); goto x;

}

for(j=0;j<n;j++) f[inde[j]]=1; printf(“\n file allocated”); printf(“\n file indexed”); for(k=0;k<n;k++) printf(“\n %d:%d”,p,inde[k],f[inde[k]]); printf(“Enter 1 to enter more files and 0 to exit\t”); scanf(“%d”,&c); if(c==1) goto x; else exit(); getch();

## EXPERIMENT :- 05

***Objective:*** Implementation of contiguous allocation techniques:

1. Worst-Fit
2. Best-Fit iii. First-Fit

***Content:***One of the simplest methods for memory allocation is to divide memory into several fixed sized partitions. Each partition may contain exactly one process. In this multiple-partition method, whwn a partion is free, a process is selected from the input queue and is loaded into the free partition . When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrivesand needs memory, a memory section large enough for this processis provided. When it is time to load orswap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best fit strategy chooses the block that is closest in size to the request. First fit chooses the first available block that is large enough. Worstfit chooses the largest available block.

***Program:***

1) **FIRST-FIT**

//C++ implementation of First-fit algorithm

#include<bits/stdc++.h>

Using namespace std;

//Function to allocate memory to //blocks as per First fit algorithm void firstFit(int blockSize[],int m, int processSize[],int n)

{

//Stores block id of the //block allocated to a process int allocation[n];

//Initially no block is assigned to any process memset(allocation,-1,sizeof (allocation ));

//pick each process and find suitable blocks //according to its size ad assign to it for (int i=0;i<n; i++)

{

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

{

if(blockSize[j]>=processSize[i])

{

//allocate block j to p[i] process allocation[i]=j;

//Reduce available memory in this block.

blockSize[j]-=processSize[i]; break;

}

}

}

cout<<”\nProcess No.\tProcess Size\tBlock no.\n”; for(int i=0;i<n;i++)

{

cout<<””<<i+1<<”\t\t”; if(allocation[i]!=-1) cout<<allocation[i]+1; else cout<<”Not Allocated”; cout<<endl;}

}

//Driver code int main()

{

int blockSize[]={100,500,200,300,600}; int processSize[]={212,417,112,426}; int m=sizeof(blockSize)/sizeof(blockSize[0]); int n=sizeof(processSize)/sizeof(processSize[0]); firstFit(blockSize,m,processSize,n); return0;

}

2) **BEST –FIT** #include<bits/stdc++.h> using namespace std; //Function to allocate memory to blocks as per Best fit algorithm void bestFit(int blockSize[],int m,int processSize[],int n)

{

//stores block id of the block allocated to a process int allocation[n];

//Initially no block is assigned to any process memset(allocation,-1,sizeof(allocation));

//pick each process and find suitable blocks according to its size ad assign to it for(int i=0;i<n;i++)

{

//Find the best fit block for current process int bestIdx=-1; for (int j=0;j<m;j++)

{

if (blockSize[j]>=processSize[i])

{

if(bestIdx==-1) bestIdx=j; else if (blockSize[bestIdx]>blockSize[j]) bestIdx=j;

}

}

//If we could find a block for current process

If(bestIdx!=-1)

{

//allocate block j to p[i] process

Allocation[i]=bestIdx;

//Reduce available memory in this block. blockSize[bestIdx]-=processSize[i];

}

}

cout<<”\nProcess no.\tProcess Size\tBlock no.\n”; for(int i=0;i<n;i++)

{

cout<<””<<i+1<<i+1<<”\t\t”<<processSize[i]<<’\t\t”; if (allocation[i]!=-1) cout<<allocation[i]+1; else cout<<”Not Allocated”; cour<<endl;

}

}

//Driver code int main()

{

int blockSize[]={100,500,200,300,600}; int processSize[]={212,417,112,426}; int m=sizeof(blockSize)/sizeof(blockSize[0]); int n=sizeof(processSize)/sizeof(processSize[0]); bestFit(blockSize,m,processSize,n); return 0;}

3) **WORST FIT**

#include<bits/stdc++.h>

Using namespace std;

//Function to allocate memory to blocks as per worst fit algorithm void worstFit(int blockSize[],int m,int processSize[],

{

//stores block id of the block allocated to a process int allocation[n];

//Initially no block is assigned to any process memset(allocation,-1,sizeof(allocation));

//pick each process and find suitable blocks according to its size ad assign it for (int i=0;i<n;i++)

{

int wstIdx=-1; for(int j=0;j<m;j++) {

if(BlockSize[j]>=processSize[i])

{

if(wstIsdx==-1) wstIdx=j; else if (blockSize[wstIdx]<blockSize[j]) wstIdx=j;

}

}

if(wstIdx!=-1)

{

allocation[i]=wstIdx; blockSize[wstIdx]-=processSize[i];

}

}

cout<<”\nProcess no.\tProcess Size\tBlock no.\n”; for(int i=0;i<n;i++)

{

cout<<””<<i+1<<”\t\t”<<processSize[i]<<”\t\t”; if (allocation[i]!=-1) cout<<allocation[i]+1; else

cout<<”Not Allocated”; cout<<endl;

}

}

//Driver code int main()

{

int blockSize[]={100,500,200,300,600}; int processSize[]={212,417,112,426}; int m=sizeof(blockSize)/sizeof(blockSize[0]); int n=sizeof(processSize)/sizeof(processSize[0]); worstFit(blockSize,m,processSize,n); return 0;

}

## EXPERIMENT :- 06

***Objective:*** Calculation of external and internal fragmentation

1. Free space list of blocks from system
2. List process file from the system

**Internal Fragmentation:**

Internal fragmentation happens when the memory is split into mounted sized blocks. Whenever a method request for the memory, the mounted sized block is allotted to the method. Just in case the memory allotted to the method is somewhat larger than the memory requested, then the distinction between allotted and requested memory is that the Internal fragmentation.

**External Fragmentation:**

External Fragmentation happens when there`s a sufficient quantity of area within the memory to satisfy the memory request of a method. However the process`s memory request cannot be fulfilled because the memory offered is during a non-contiguous manner. Either you apply first-fit memory allocation strategy it’ll cause external fragmentation.

**Experiment :-07**

**Objective:**

Implementation of compaction for the continually changing memory layout and calculate total movement of data

**Content:**

Compaction is performed after deallocation in the size class affected by the deallocation request. It implies movement of only one memory object in the affected size-class. Before presenting the algorithm, we state two invariants and two rules that are related to our com paction strategy.

Not every deallocation request requires moving of a memory object. The cases when no moving is necessary are:

* The deallocated memory object is in the unique notfull page of the size-class. This case imposes no work except when the deallocated memory object is the only memory object in the page. Then the page is removed from the size-class.
* There is no not-full page in the size-class where deallocation happened. In this case only a fixed number of list-reference updates is needed in order that the affected page becomes the last page in the size-class list.

**Program:**

void compaction(size\_class, affected\_page){ if(affected\_page != last\_page) { if (is\_full(last\_page)) { set\_last(affected\_page);

} else {

move(object, last\_page, affected\_page); abstract\_address\_space\_update(object);

}

}

}

## EXPERIMENT:- 08

***Objective:*** Implementation of resource allocation graph(RAG) **Program:**

//C program to demonstrate waitpid()

#include<stdio.h>

#include<stdlib.h>

#include<sys/wait.h> #include<unistd.h> void waitexample()

{

int I,stat; pid\_t pid[5]; for(i=0;i<5;i++)

{

if((pid[i]=fork())==0)

{

sleep(1); exit(100+i);

}

}

for(i=0;i<5;i++)

{

pid\_t cpid=waitpid(pid[i],&stat,0); if(WIFEXITED(stat)) printf(“Child %d terminated with status:%d\n”, cpid,WEXITSTATUS(stat));

}

}

//Driver code int main()

{

Waitexample(); return 0;

}

**Experiment:- 09**

**Objective:**

Implementation of Banker’s algorithm **Content:**

Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

**Program:**

// Banker's Algorithm #include <iostream> using namespace std;

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k; n = 5; // Number of processes m = 3; // Number of resources

int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0; for (k = 0; k < n; k++) { f[k] = 0;

}

int need[n][m]; for (i = 0; i < n; i++) { for (j = 0; j < m; j++) need[i][j] = max[i][j] - alloc[i][j];

} int y = 0; for (k = 0; k < 5; k++) { for (i = 0; i < n; i++) { if (f[i] == 0) {

int flag = 0; for (j = 0; j < m; j++) { if (need[i][j] > avail[j]){ flag = 1; break;

}

}

if (flag == 0) {

ans[ind++] = i; for (y = 0; y < m; y++) avail[y] += alloc[i][y]; f[i] = 1;

}

}

}

}

cout << "Following is the SAFE Sequence" << endl; for (i = 0; i < n - 1; i++) cout << " P" << ans[i] << " ->"; cout << " P" << ans[n - 1] <<endl;

return (0);

}

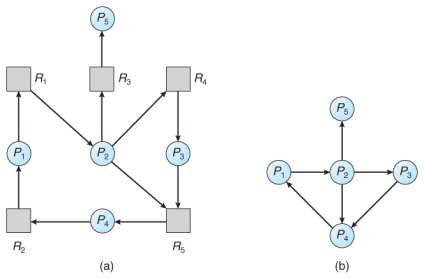
**Experiment :- 10**

**Objective:**

Conversion of resource allocation graph (RAG) to wait for graph (WFG) for each type of method used for storing graph.

**Content:**

One such deadlock detection algorithm makes use of a wait-for graph to track which other processes a process is currently blocking on. In a waitfor graph, processes are represented as nodes, and an edge from process Pi to Pj implies Pj is holding a resource that Pi needs and thus Pi is



waiting for Pj to release its lock on that resource. There may be processes waiting for more than a single resource to become available. Graph cycles imply the possibility of a deadlock.

* If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a *wait-for graph*.

Figure 7.9 - (a) Resource allocation graph. (b) Corresponding wait-for graph

* A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
* An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holdingAs before, cycles in the wait-for graph indicate deadlocks.
* This algorithm must maintain the wait-for graph, and periodically search it for cycles.

**Experiment :-11**

**Objective:**

Implement the solution for Bounded Buffer (producer-consumer) problem using inter process communication techniques-Semaphores

**Content:**

Producer consumer problem is also known as bounded buffer problem. In this problem we have two processes, producer and consumer, who share a fixed size buffer. Producer work is to produce data or items and put in buffer. Consumer work is to remove data from buffer and consume it. We have to make sure that producer do not produce data when buffer is full and consumer do not remove data when buffer is empty.

The producer should go to sleep when buffer is full. Next time when consumer removes data it notifies the producer and producer starts producing data again. The consumer should go to sleep when buffer is empty. Next time when producer add data it notifies the consumer and consumer starts consuming data. This solution can be achieved using semaphores.

**Program:**

#include<stdio.h> #include<stdlib.h> int mutex=1,full=0,empty=3,x=0; int main() { int n; void producer(); void consumer();

int wait(int); int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit"); while(1) {

printf("\nEnter your choice:"); scanf("%d",&n); switch(n) {

case 1: if((mutex==1)&&(empty!=0)) producer(); else printf("Buffer is full!!"); break;

case 2: if((mutex==1)&&(full!=0)) consumer(); else printf("Buffer is empty!!"); break; case 3: exit(0); break; }

} return 0; }

int wait(int s) { return (--s);

}

int signal(int s) { return(++s); } void producer()

{ mutex=wait(mutex); full=signal(full); empty=wait(empty); x++;

printf("\nProducer produces the item %d",x); mutex=signal(mutex);

} void consumer() { mutex=wait(mutex); full=wait(full); empty=signal(empty); printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

**Experiment :-12**

**Objective:**

Implement the solutions for Readers-Writer’s problem using inter process communication technique -Semaphore **Content:**

**Writer process:**

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

**Reader process:**

1. Reader requests the entry to critical section.
2. If allowed:
   * it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
   * It then, signals mutex as any other reader is allowed to enter while others are already reading.
   * After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

**Program:**

**Writer code** do {

// writer requests for critical section wait(wrt);

// performs the write // leaves the critical section signal(wrt);

} while(true);

**Reader Code:**

do {

// Reader wants to enter the critical section wait(mutex);

// The number of readers has now increased by 1

readcnt++;

// there is atleast one reader in the critical section

**// this ensure no writer can enter if there is even one reader**

**// thus we give preference to readers here**

if (readcnt==1) wait(wrt);

// other readers can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex); // a reader wants to leave

readcnt--;

// that is, no reader is left in the critical section, if (readcnt == 0)

signal(wrt); // writers can enter

signal(mutex); // reader leaves

} while(true);