Department of Information Technology
BABU BANARASI DAS
Institute of Technology & Management, Lucknow
Recognized by AICTE, Govt.of India & Affiliated to Dr. A.P.J. Abdul Kalam Technical University,Lucknow,AKTU College Code=054

LAB MANUAL
OF
OPERATING SYSTEM

[BCS – 451]

B.TECH, 2ND Year, Semester -IV

Dr. A.P.J. Abdul Kalam Tech. University
Uttar Pradesh

2025-26

Department of Information Technology

Faculty Name: Mr. Arvind Goutam

(Assistant Professor)

Approved by: (Head of the Department)

MANUAL CONTENTS

This manual is intended for the 2nd year students of Information Technology in the subject of Operating System Lab. This manual typically contains practical/lab sessions related Operating System covering various aspects related to the subject of enhanced understanding.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

Good Luck for your Enjoyable Laboratory Sessions.

PREFACE

This practical manual will be helpful for students of Information Technology for understanding the course from the point of view of applied aspects. Though all the efforts have been made to make this manual error free, yet some errors might have crept in inadvertently. Suggestions from the readers for the improvement of the manual are most welcomed.

VISION OF THE DEPARTMENT

Vision of the department to transform the students to be skilled IT professionals, Innovative leaders, and environmentally receptive citizens.

MISSION OF THE DEPARTMENT

M1: To develop and deliver quality academic programs in emerging & innovative fields of engineering to empower the students to meet industry standards.

M2: To provide students with the ethical and professional tools to become productive & become continuous Learners.

M3: To inculcate professional behavior, positive attitude, and communication skills.

.

Program Educational Objectives(PEOs) of Department

- **PEO1:** To develop the ability among students to understand the concept of core Information Technology subjects that will facilitate understanding of new Technology.
- **PEO2:** To embed a strong foundation in the engineering fundamentals to solve, analyze and design real time engineering products.
- **PEO3:** To give exposures to emerging edge technologies, adequate training and opportunities to work as team on multidisciplinary projects with effective communication skills and leadership qualities

Program Outcomes:(PO)

PO 1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering Fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4. Conduct investigations of complex problems: Use research-based knowledge a d research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6. The engineer and society: Apply reasoning informed by the contextual knowled e to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear

PO 11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

Program Specific Outcomes(PSOs) of the Department

PSO1: Use and apply current technical concepts and practices in the core Information Technologies of human Machine Learning, Information management ,programming, Cloud computing.

PSO2: Design, develop and test software systems for worldwide network of computers to provide solutions to real world problem.

Course Evaluation Scheme

Sr No	Subject Code	Subject Name	Periods			Evaluation Scheme				Total	Credit
						Sessional Assessment			PE		
			L	T	P	CT	TA	PS			
1.	BCS451	Operating System Lab	0	0	2	-	-	50	50	100	1

Course Objectives:

The teacher will explain:

1.	Able to learn implementation of CPU scheduling.
2.	Able to implement memory management.
3.	Able to implement linear non linear data structure.

Pre- requisite: Fundamentals of C language.

Course Outcomes (COs)

Course Outcomes: The students should be able to:		**Bloom's Level**
CO1	Use modern operating systems calls and synchronization libraries in software/hardware interfaces.	L2
CO2	Implement different sorting and searching algorithms. Analyze and simulate CPU Scheduling Algorithms like FCFS,SJF, Priority, Round- Robin.	L3
CO3	Implement memory management schemes and page replacement schemes.	L3
CO4	Simulate file allocation and organization techniques and disk scheduling algorithms.	L3
CO5	Understand the concepts of deadlock in operating system and implement the multiprogramming system.	L3

CO-PO-PSO Mapping

COs	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**
CO1	3	2	3	-	3	-	-	-	-	-	-	2	2	1
CO2	3	3	2	-	2	-	-	-	-	-	-	2	2	2
CO3	3	3	3	-	2	-	-	-	-	-	-	2	2	1
CO4	3	3	3	-	2	-	-	-	-	-	-	2	2	1
CO5	3	3	3	-	3	-	-	-	-	-	-	2	2	1

The extent of mapping is as follows: 1 for low, 2 for moderate, 3 for high &"-" for No correlation between CO & PO.

Index

S. No	LIST OF EXPERIMENTS	COs	BL
1	To study of hardware and Software requirements of different operating systems (UNIX, LINUX, WINDOWS XP, WINDOWS 10)	CO1	L2
2	Execute various UNIX system calls for i. Process management ii. File management iii. Input/output Systems calls	CO2	L3
3	Implement CPU Scheduling Policies: i. SJF ii. Priority iii. FCFS iv. Multi-level Queue	CO3	L3
4	Implement file storage allocation technique: i. Contiguous(using array) ii. Linked –list(using linked-list) iii. Indirect allocation (indexing)	CO5	L3
5	Implementation of contiguous allocation techniques: i. Worst-Fit ii. Best- Fit iii. First- Fit	CO4	L3
6	Calculation of external and internal fragmentation i. Free space list of blocks from system ii. List process file from the system	CO4	L3
7	Implementation of compaction for the continually changing memory layout and calculate total movement of data	CO4	L3
8	Implementation of resource allocation graph RAG)	CO4	L3
9	Implementation of Banker‟s algorithm	CO4	L3
10	Conversion of resource allocation graph (RAG) to wait for graph (WFG) for each type of method used for storing graph.	CO4	L4
Program Beyond Syllabus			
1	Implement the solution for Bounded Buffer (producer-consumer)problem using inter process communication techniques-Semaphores	CO2	L3
2	Implement the solutions for Readers-Writers problem using inter process communication technique - Semaphore	CO2	L3

Softwares & Hardwares Required:

1. **Operating System:** Windows, Linux, macOS.
2. **Software:** Turbo C
3. **Hardware: Processor**: At least **Intel i3,RAM**: A minimum of **8 GB RAM**.

Experiment – 1

Objective :
To study of hardware and software requirements of different operating systems (UNIX, LINUX, WINDOWS XP, WINDOWS 10).

Content :
An Operating System acts as an intermediary between the user & computer hardware. The purpose of an operating system is to provide an environment at which user can execute programs conveniently and efficiently.
An Operating System is a program that controls the execution of application programs and acts as an interface between the user & a computer and the computer hardware.
There are various types of operating system used throughout the world and this depends mainly on the types of operations performed. Most commonly used operating systems are as follows :

- Windows 10, 8, 7, XP
- LINUX
- UNIX
- macOS

HARDWARE AND SOFTWARE MINIMUM REQUIREMENTS
All computer software needs certain hardware components or other software resources to be present on a computer.

Software Requirements
Software requirements deal with defining software resource requirements and prerequisites that need to be installed on a computer to provide optimal functioning of an application such as platform, APIs and drivers, web browser etc.

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.

1.1. UNIX :

a) Hardware Requirements →

- Processor : Depends on versions (SPARC, PowerPC, x86 etc)
- RAM : minimum 512 MB (modern versions need 1–2 GB+)
- Storage : 10 GB minimum
- Display : VGA compatible monitor
- Network card (for networking)

b) Software Requirements →

- UNIX OS distribution (eg: Solaris, AIX, HP-UX)

- Shell (sh, csh, ksh)
- C compiler (gcc)
- Text editors (vi, nano)

c) Features →

- Open source
- Secure and stable
- supports multiple file systems
- Used in servers, desktops, embedded systems

1.2. LINUX :

a) Hardware Requirements →

- Processor : 1 GHz or higher
- RAM : 1 GB minimum (2 GB recommended)
- Storage : 20 GB minimum
- Graphics card and monitor
- Keyboard and mouse

b) Software Requirements →

- Linux Distribution (Ubuntu, Fedora, Debian)
- Boot loader (GRUB)
- Package manager (APT, YUM)
- Shell (Bash)

c) Features →

- Open source
- Secure and stable
- supports multiple file systems
- Used in servers, desktops, embedded systems

1.3. Windows 10 :

a) Hardware Requirements →

- Processor : Dual-core 2 GHz or higher
- RAM : 4 GB or more
- Storage : 128 GB SSD (recommended)
- Graphics : Dedicated GPU (optional but better for gaming/design)

b) Software Requirements →

- Bootable USB/DVD with Windows 10 ISO
- Device drivers (automatically installed in most cases)
- Antivirus / Windows Defender
- Microsoft account (optional but recommended)

c) Features →

- Improved Start Menu (combination of Windows 7 + 8 style)
- Cortana virtual assistant
- Microsoft Edge browser
- Virtual desktops
- Touchscreen support
- Windows Defender security
- Regular automatic updates

1.4. Windows XP :

a) Hardware Requirements (Minimum) →

- Processor : 233 MHz
- RAM : 64 MB
- Storage : 1.5 GB free space
- Super VGA (800 × 600)
- CD-ROM drive

b) Software Requirements →

- Windows XP installation CD
- Device drivers

- Antivirus software

c) Features →

- User-friendly GUI
- Plug and play support
- NTFS file system support

1.5. Windows 7/8 →

a) Hardware Requirements (Minimum)
Windows 7 :

- Processor : 1 GHz
- RAM : 1 GB (32-bit) / 2 GB (64-bit)
- Storage : 16 GB (32-bit) / 20 GB (64-bit)
- DirectX 9 graphics device

Windows 8 :

- Processor : 1 GHz with PAE, NX, SSE2
- RAM : 1 GB (32-bit) / 2 GB (64-bit)
- Storage : 16–20 GB
- Touchscreen (optional)

b) Software Requirements :
Windows 7 :

- Windows 7 DVD or bootable USB
- Device drivers
- System software
- Utility software (recommended)
- Updates & Service Packs

Windows 8 :

- Windows 8 DVD or bootable USB
- Sound drivers, display/graphics drivers
- UEFI firmware (recommended), Windows Boot Manager

- Internet Explorer, Windows Defender, Windows Store apps
- Windows 8.1 update recommended, automatic updates enabled

c) Features →

- Improved security
- Better performance
- Touch support (Windows 8)
- Aero interface (Windows 7)

1.6. macOS →

a) Hardware Requirements →

- Supported Device : Apple Mac (MacBook, iMac, Mac Mini, Mac Studio, Mac Pro)
- Processor : Intel-based Mac or Apple Silicon (M1/M2/M3 chips)
- RAM : Minimum 4 GB (8 GB recommended)
- Storage : At least 20–35 GB free space
- Graphics : Built-in Apple GPU or supported graphics hardware
- macOS only runs officially on Apple hardware.*

b) Software Requirements →

- macOS installer (from App Store)
- Apple ID (for App Store & iCloud)
- Built-in apps : Safari, Finder, Terminal
- Xcode (for development, optional)

c) Features →

- UNIX-based architecture (stable & secure)
- Finder file management system
- Spotlight search
- Siri voice assistant

- Time Machine backup
- iCloud integration
- AirDrop file sharing
- Gatekeeper security system

Experiment – 2

Objective :
Executing various UNIX system calls for process management, file management and input/output system calls.

1.7.) Process Management System Calls :

System Call	Purpose
fork()	Create new process
exec()	Execute new program
wait()	Wait for child process
exit()	Terminate process
getpid()	Get process ID

Example Program (Process Creation)

```
#include <stdio.h>
#include <unistd.h>

int main()
{
   fork();
   printf("Hello from process\n");
   return 0;
}
```

1.8.) File Management System Calls :

System Call	Purpose
open()	Open file
read()	Read file
write()	Write to file
close()	Close file
unlink()	Delete file

Example Program

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
   int fd;

   fd = open("test.txt", O_CREAT | O_WRONLY, 0644);
   write(fd, "Hello File\n", 11);
   close(fd);

   return 0;
}
```

1.9.) Input / Output System Calls :

System Call	Purpose
read()	Read input
write()	Write output
scanf()	Input (library function)
printf()	Output (library function)

Example Program (I/O)

```
#include <unistd.h>

int main()
{
   char msg[] = "Hello World\n";
   write(1, msg, 12);
   return 0;
}
```

Experiment – 3

Objective :

To implement different CPU scheduling policies :

1. FCFS
2. Shortest Job First (SJF)
3. Priority Scheduling
4. Multi-level Queue Scheduling

1- FCFS (First Come First Serve)
Theory :

5. FCFS is the simplest scheduling algorithm.
6. Processes are executed in the order of their arrival.
7. It is a non-preemptive scheduling algorithm.
8. Once a process starts execution, it runs till completion.

Disadvantages :

9. Causes convoy effect
10. Average waiting time is high

Sample Input :
Process → P1, P2, P3
Burst Time → 5, 3, 8

Sample Output :

Process	Burst Time	Waiting Time	Turnaround Time
P1	5	0	5
P2	3	5	8
P3	8	8	16

2) Shortest Job First (SJF)
Theory :

11. In SJF scheduling, the process with the smallest burst time is selected first.
12. It can be preemptive or non-preemptive.
13. It provides minimum average waiting time.

Disadvantages :

14. Starvation may occur
15. Difficult to predict burst time

3) Priority Scheduling
Theory :

16. Each process is assigned a priority.
17. The process with highest priority is executed first.
18. Can be preemptive or non-preemptive.

Disadvantages :

19. Starvation of low priority processes

4) Multi-Level Queue Scheduling

Theory :

20. Ready queue is divided into multiple queues.

21. Each queue has its own scheduling algorithm.

22. Processes are assigned to queues based on type.

Example :

23. System processes → FCFS

24. User processes → Round Robin

Execution :

25. Queue 1 executes first

26. Then Queue 2

Experiment – 4

Objective :

To implement file storage allocation techniques :

27. Contiguous Allocation

28. Linked Allocation

29. Indexed Allocation

1.) Contiguous Allocation

Theory :

30. File occupies contiguous blocks on disk.

31. Directory stores starting block and length.

Advantages :

32. Simple implementation

33. Fast access

Disadvantages :

34. External fragmentation

35. Difficult to resize file

Sample Input :
Start block = 10
Length = 4
Sample Output :
10, 11, 12, 13
2) Linked Allocation
Theory :

36. File is stored as linked list of blocks.

37. Each block contains pointer to next block.

Advantages :

38. No external fragmentation

39. Dynamic allocation

Disadvantages :

40. Slow access

41. Pointer overhead

Sample Output :
5 → 9 → 14 → 20 → NULL

3) Indexed Allocation

Theory :

42. All block addresses are stored in index block.

43. Directory contains address of index block.

Advantages :

44. Direct access

45. No external fragmentation

Disadvantages :

46. Extra memory for index block

Sample Output :
2, 6, 8, 12, 15

Experiment – 5

Objective :
Implementation of contiguous allocation technique :
(i) Worst Fit
(ii) Best Fit
(iii) First Fit

Worst Fit Memory Allocation :
Algorithm →

1. Start
2. Input number of blocks and their sizes.
3. Input number of processes and their sizes.
4. For each process, find the largest block available.
5. Allocate that block to the process.
6. Update remaining block size.
7. If no block fits, mark process not allocated.
8. Display allocation result.
9. Stop

Program : #include <stdio.h>

```
int main()
{
   int blockSize[10], processSize[10], block[10];
   int i, j, nb, np, worst;

   printf("Enter number of blocks : ");
   scanf("%d", &nb);

   printf("Enter size of each block : \n");
   for(i=0; i<nb; i++)
      scanf("%d", &blockSize[i]);

   printf("Enter number of processes : ");
   scanf("%d", &np);

   printf("Enter size of each process : \n");
   for(i=0; i<np; i++)
      scanf("%d", &processSize[i]);

   for(i=0; i<np; i++)
      block[i] = -1;
```

```
    for(i=0; i<np; i++)
    {
        worst = -1;
        for(j=0; j<nb; j++)
        {
            if(blockSize[j] >= processSize[i])
            {
                if(worst == -1 || blockSize[j] > blockSize[worst])
                    worst = j;
            }
        }

        if(worst != -1)
        {
            block[i] = worst;
            blockSize[worst] -= processSize[i];
        }
    }

    printf("\nProcess No\tProcess Size\tBlock No\n");
    for(i=0; i<np; i++)
    {
        printf("%d\t\t%d\t\t", i+1, processSize[i]);
        if(block[i] != -1)
            printf("%d", block[i]+1);
        else
            printf("Not Allocated");
        printf("\n");
    }

    return 0;
}
```

Sample Output :

```
Enter number of blocks : 5
Enter size of each block :
100
500
200
300
600
Enter number of processes : 4
Enter size of each process :
212
417
112
426
Process No Process Size Block No
1 212 5
```

2 417 2
3 112 5
4 426 Not Allocated

2) Best Fit Memory Allocation :

Algorithm →

1. Start
2. Input number of blocks and their sizes.
3. Input number of processes and their sizes.
4. For each process, find the smallest block that can fit the process.
5. Allocate the block to the process.
6. Update remaining memory of the block.
7. If no block fits, mark process as NOT allocated.
8. Display allocation result.
9. Stop

Program →

```
#include <stdio.h>
int main()
{
   int blockSize[10], processSize[10], block[10];
   int i, j, nb, np, best;

   printf("Enter number of blocks : ");
   scanf("%d", &nb);

   printf("Enter size of each block : \n");
   for(i=0; i<nb; i++)
      scanf("%d", &blockSize[i]);

   printf("Enter number of processes : ");
   scanf("%d", &np);

   printf("Enter size of each process : \n");
   for(i=0; i<np; i++)
      scanf("%d", &processSize[i]);

   for(i=0; i<np; i++)
      block[i] = -1;

   for(i=0; i<np; i++)
```

```
    {
        best = -1;
        for(j=0; j<nb; j++)
        {
            if(blockSize[j] >= processSize[i])
            {
                if(best == -1 || blockSize[j] < blockSize[best])
                    best = j;
            }
        }

        if(best != -1)
        {
            block[i] = best;
            blockSize[best] -= processSize[i];
        }
    }

    printf("\nProcess No\tProcess Size\tBlock No\n");
    for(i=0; i<np; i++)
    {
        printf("%d\t\t%d\t\t", i+1, processSize[i]);
        if(block[i] != -1)
            printf("%d", block[i]+1);
        else
            printf("Not Allocated");
        printf("\n");
    }

    return 0;
}
```

Sample Output :

```
Enter number of blocks : 5
Enter size of each block :
100
500
200
300
600
Enter number of processes : 4
Enter size of each process :
212
417
112
426
Process No Process Size Block No
1 212 4
2 417 2
3 112 3
4 426 5
```

3) First Fit Memory Allocation :

Algorithm →

1. Start
2. Input number of blocks and their sizes.
3. Input number of processes and their sizes.
4. For each process, find the first block that can accommodate the process.
5. Allocate that block to the process.
6. Update remaining block size.
7. If no block fits, mark process NOT allocated.
8. Display allocation result.
9. Stop

Program →

```
#include <stdio.h>
int main() { int blockSize[10], processSize[10], block[10];
int i, j, nb, np; printf("Enter number of blocks : ");
 scanf("%d", &nb);
printf("Enter size of each block : \n");
 for(i = 0; i < nb; i++)
scanf("%d", &blockSize[i]);
 printf("Enter number of processes : ");
 scanf("%d", &np);
printf("Enter size of each process : \n");
 for(i = 0; i < np; i++) scanf("%d", &processSize[i]);
for(i = 0; i < np; i++) block[i] = -1; for(i = 0; i < np; i++)
{ for(j = 0; j < nb; j++) {
 if(blockSize[j] >= processSize[i])
{ block[i] = j;
 blockSize[j] -= processSize[i];
break; } } } printf("\nProcess No\tProcess Size\tBlock No\n");
for(i = 0; i < np; i++) {
 printf("%d\t\t%d\t\t", i+1, processSize[i]);
 if(block[i] != -1) printf("%d", block[i] + 1);
else printf("Not Allocated");
printf("\n"); } return 0; }
```

Sample Input :

```
Enter number of blocks : 5
Enter size of each block :
100
```

500
200
300
600
Enter number of processes : 4
Enter size of each process :
212
417
112
426

Sample Output :
Process No | Process Size | Block No
1 | 212 | 2
2 | 417 | 5
3 | 112 | 2
4 | 426 | Not Allocated

Experiment – 6
Objective :
Calculation of External and Internal Fragmentation of a memory allocation system.

Theory :
Internal fragmentation occurs when memory is allocated to a process but some part of it remains unused inside the allocated block.
External fragmentation occurs when total free memory is enough to satisfy a request but it is not contiguous.

Algorithm :
1. Start
2. Enter number of memory blocks
3. Enter size of memory blocks
4. Enter number of processes
5. Enter size of each process
6. Allocate processes using first fit
7. Calculate internal fragmentation
8. Find free block
9. Calculate external fragmentation
10. Display results

11. Stop

Given Data :
Blocks →
B1 = 100
B2 = 500
B3 = 200
B4 = 300
B5 = 600
Processes →
P1 = 212
P2 = 417
P3 = 112
P4 = 426

Calculation of Internal Fragmentation :

Process	Block	Block Size	Process Size	Internal Fragmentation
P1	B4	300	212	88
P2	B2	500	417	83
P3	B3	200	112	88
P4	B5	600	426	174

Total Internal Fragmentation = 433 KB

External Fragmentation :
Remaining free block →
B1 = 100 KB

Experiment – 7
Objective :
Implementation of compaction for contiguous memory allocation.

Theory :
Compaction is a memory management technique used to reduce external fragmentation. It collects all free spaces and combines them into one large block.

Algorithm :
1. Start
2. Enter number of memory blocks
3. Enter size of each memory block
4. Assume 0 represents a free block
5. Traverse memory array from beginning to end
6. Whenever a hole (0) is found, shift next occupied block

7. Count movements
8. Continue until all holes are moved to end
9. Display compacted memory layout
10. Display total movement of data
11. Stop

Program -

```
#include <stdio.h>
int main()
{
   int memory[20], n, i, j, temp;
   int movement = 0;

   printf("Enter number of memory blocks : ");
   scanf("%d", &n);

   printf("Enter memory layout (0 for hole):\n");
   for(i = 0; i < n; i++)
      scanf("%d", &memory[i]);

   for(i = 0; i < n; i++)
   {
      if(memory[i] == 0)
      {
         for(j = i + 1; j < n; j++)
         {
            if(memory[j] != 0)
            {
               memory[i] = memory[j];
               memory[j] = 0;
               movement++;
               break;
            }
         }
      }
   }

   printf("\nMemory layout after compaction :\n");
   for(i = 0; i < n; i++)
      printf("%d ", memory[i]);

   printf("\nTotal Data Movement = %d", movement);

   return 0;
```

```
}
```

Sample Input :
Enter number of memory blocks : 8
Enter memory layout (0 for hole) :
1 2 0 3 0 4 5 0

Sample Output :
Memory layout after compaction :
1 2 3 4 5 0 0 0
Total Data Movement = 4

Experiment – 8
Objective :
Implementation of Resource Allocation Graph (RAG)

Content :
A Resource Allocation Graph (RAG) is used to represent allocation of resources and detect deadlocks.

Algorithm :
1. Start
2. Input number of processes and resources
3. Enter allocation matrix
4. Enter request matrix
5. Construct graph
6. Check for cycle
7. If cycle exists → deadlock may occur
8. Else → system is safe
9. Display result
10. Stop

Sample Output :
Deadlock may occur (cycle detected)

Experiment – 9
Objective :
Implementation of Banker's Algorithm

Content :

Banker's Algorithm is used for deadlock avoidance. It checks whether the system is in a safe state or not.

Algorithm :

1. Start
2. Input number of processes and resources
3. Enter allocation matrix
4. Enter maximum matrix
5. Enter available resources
6. Calculate Need = Max – Allocation
7. Initialize Finish = 0
8. Find process where Need ≤ Available
9. Add allocation to available
10. Repeat until all processes finish
11. If all finish → safe state
12. Else → unsafe state
13. Stop

Program -

```
#include <stdio.h>
int main()
{
   int alloc[10][10], req[10][10];
   int i, j, p, s, deadlock = 0;

   printf("Enter number of processes : ");
   scanf("%d", &p);

   printf("Enter number of resources : ");
   scanf("%d", &s);

   printf("Enter Allocation Matrix : \n");
   for(i = 0; i < p; i++)
      for(j = 0; j < s; j++)
         scanf("%d", &alloc[i][j]);

   printf("Enter Request Matrix : \n");
   for(i = 0; i < p; i++)
```

```
        for(j = 0; j < s; j++)
            scanf("%d", &req[i][j]);

    for(i = 0; i < p; i++)
    {
        for(j = 0; j < s; j++)
        {
            if(req[i][j] == 1 && alloc[j][i] == 1)
                deadlock = 1;
        }
    }

    if(deadlock)
        printf("Deadlock may occur (cycle detected)\n");
    else
        printf("NO Deadlock (system is safe)\n");

    return 0;
}
```

Sample Input :
Processes = 5
Resources = 3
Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Maximum Matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Available Resources
3 3 2

Sample Output :
Safe sequence : P1 P3 P4 P0 P2

Experiment – 10

Objective :
Conversion of Resource Allocation Graph (RAG) to Wait For Graph (WFG) to check type of deadlock.

Content :

- Resource Allocation Graph (RAG) is used to represent allocation of resources in a system.
- It consists of processes and resources.

- Processes are represented by circles (P1, P2, …)
- Resources are represented by rectangles (R1, R2, …)

Types of Edges :

- Request edge : Process → Resource
- Allocation edge : Resource → Process

Wait For Graph (WFG) :

- A wait-for graph is derived from RAG by removing resource nodes and showing only process-to-process dependency.
- If a process is waiting for a resource held by another process →
  P1 → P2

Algorithm :

1. Take the given graph
2. Identify request edges (Process → Resource)
3. Find which process holds that resource
4. Replace
   Process → Resource → Process
   with
   Process → Process
5. Remove all resource nodes
6. The obtained graph is WFG

Example :
Given :
P1 → R1 (request)
R1 → P2 (allocated)
P2 → R2 (request)
R2 → P1 (allocated)

Result :
Cycle exists → Deadlock present

Experiment – 11

Objective :
Implementation of solution for Bounded Buffer (Producer-Consumer problem) using Inter Process Communication Technique.

Content :

- Producer produces data and places it into buffer.
- Consumer removes data from buffer.
- Synchronization is required to avoid race condition.
- Semaphores are used:

mutex → mutual exclusion
empty → counts empty slots
full → counts filled slots

Algorithm :
Producer →

7. Wait for an empty slot
8. Wait for mutex
9. Add item to buffer
10. Release mutex
11. Signal full

Consumer →

12. Wait for full slot
13. Wait for mutex
14. Remove item from buffer
15. Release mutex
16. Signal empty

Program →

```
#include <stdio.h>
#include <stdlib.h>

int mutex = 1, full = 0, empty = 3, x = 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 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);
}

int main()
{
   int choice;

   while(1)
   {
      printf("\n1. Producer\n2. Consumer\n3. Exit");
      printf("\nEnter your choice : ");
      scanf("%d", &choice);

      switch(choice)
      {
         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;

            default:
               printf("Invalid choice");
         }
      }

      return 0;
}
```

Sample Output :

17. Producer

18. Consumer

19. Exit

Enter your choice : 1
Producer produces item 1
Enter your choice : 1
Producer produces item 2
Enter your choice : 2
Consumer consumes item 2

Experiment – 12

Objective :

Implementation of Readers-Writers problem using semaphores.

Content :

- Multiple readers can read simultaneously
- Only one writer can write at a time
- No reader allowed when writer is writing

Semaphores used:
mutex → protects read count
wrt → controls writing

Algorithm :

Reader →

20. Enter critical section

21. First reader blocks writer

22. Read data

23. Last reader allows writer

Writer →

24. Wait for write semaphore

25. Write data

26. Release semaphore

Program →

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex, wrt;
int readcount = 0;

void *reader(void *arg)
{
   sem_wait(&mutex);
   readcount++;

   if(readcount == 1)
      sem_wait(&wrt);

   sem_post(&mutex);

   printf("Reader is reading\n");

   sem_wait(&mutex);
   readcount--;

   if(readcount == 0)
      sem_post(&wrt);

   sem_post(&mutex);

   return NULL;
}
```

```
void *writer(void *arg)
{
   sem_wait(&wrt);
   printf("Writer is writing\n");
   sem_post(&wrt);
   return NULL;
}

int main()
{
   pthread_t r1, r2, w1;

   sem_init(&mutex, 0, 1);
   sem_init(&wrt, 0, 1);

   pthread_create(&r1, NULL, reader, NULL);
   pthread_create(&r2, NULL, reader, NULL);
   pthread_create(&w1, NULL, writer, NULL);

   pthread_join(r1, NULL);
   pthread_join(r2, NULL);
   pthread_join(w1, NULL);

   sem_destroy(&mutex);
   sem_destroy(&wrt);

   return 0;
}
```

Output :

```
Reader is reading
Reader is reading
Writer is writing
```