# Operating Systems

# Lab Report

# (KCS-451)

# Session 2022-23


## Department of Allied Computational Science & Engineering

G L Bajaj Institute of Technology and Management
Greater Noida – 201306

**Submitted To:**
Mr. Sachin Chawla

**Submitted By:**
Himanshu Kumar
(2101921640017)

# Index

| S No. | Name of Program | CO | Date of Coding | Date of Submission | Signature |
|---|---|---|---|---|---|
| 1. | Study of hardware and software requirements of different operating systems (UNIX, LINUX, WINDOWS XP, WINDOWS7/8) | | | | |
| 2. | Execute various UNIX system calls for<br>i. Process management<br>ii. File management<br>iii. Input/output Systems calls | | | | |
| 3. | Implement CPU Scheduling Policies:<br>i. SJF<br>ii. FCFS | | | | |
| 4. | Implement CPU Scheduling Policies:<br>i. Round Robin<br>ii. Priority | | | | |
| 5. | Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance. | | | | |
| 6. | Implement file storage allocation technique:<br>i. Contiguous(using array)<br>ii. Linked-list(using linked-list)<br>iii. Indirect allocation (indexing) | | | | |
| 7. | Calculation of external and internal fragmentation:<br>i. Free space list of blocks from system<br>ii. List process file from the system | | | | |
| 8. | Implementation of contiguous allocation techniques:<br>i. Worst-Fit<br>ii. Best-Fit<br>iii. First-Fit | | | | |
| 9. | Implement the solution for Bounded Buffer (producer-consumer) problem using inter process communication techniques-Semaphores. | | | | |
| 10. | Implementation of resource allocation graph (RAG). | | | | |

# PROGRAM – 1

**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:

**1. Windows 10**

- Processor: 1 gigahertz (GHz) or faster processor or SoC.
- RAM: 1 gigabyte (GB) for 32-bit or 2 GB for 64-bit.
- Hard disk space: 16 GB for 32-bit OS or 20 GB for 64-bit OS.
- Graphics card: DirectX 9 or later with WDDM 1.0 driver.
- Display: 800 x 6009 with WDDM driver.

**2. Windows XP**

The minimum hardware requirements for Windows XP Home Edition are:

i. Pentium 233-megahertz (MHz) processor or faster (300 MHz is recommended).
ii. At least 64 megabytes (MB) of RAM (128 MB is recommended).
iii. At least 1.5 gigabytes (GB) of available space on the hard disk.
iv. CD-ROM or DVD-ROM drive.
v. Keyboard and a Microsoft Mouse or some other compatible pointing device.
vi. Video adapter and monitor with Super VGA (800 x 600)or higher resolution.
vii. Sound card.
viii. Speakers or headphones.

**3. UINX OS**

i. RAM: 1 GB
ii. Processor: IBM 604e processor with a clock speed of 375 MHz or faster.
iii. Free disk space: /temp must have 1 GB free disk space. If Tivoli Identity Manager installs WebSphere Application Server, (WAS HOME) must have 800 MB free disk space and /var must have 300 MB free disk space. Allocate 500 MB for /itim45.

**4. LINUX**

i. 32-bit Intel-compatible processor running at 2 GHz or greater.
ii. 512 MB RAM.
iii. Disk space: 2.5 GB for Pipeline Pilot server plus components.
iv. A DVD-ROM drive.

# PROGRAM – 2

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

a) Process management
b) File management
c) Input/Output System 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 |
| fork() | A new process is created using this command |
| chroot() | This changes the root directory of a file. |

| System Call | Description |
|---|---|
| exit() | The exit system call is used to exit a process. |

| | | |
|---|---|---|
| File Structure Related Calls | Creating a Channel | creat() |
| | | open() |
| | | close() |
| | Input/Output | read() |
| | | write() |
| | Random Access | lseek() |
| | Channel Duplication | dup() |
| | Aliasing and Removing Files | link() |
| | | unlink() |
| | File Status | stat() |
| | | fstat() |
| | Access Control | access() |
| | | chmod() |
| | | chown() |
| | | umask() |
| | Device Control | ioctl() |
| Process Related Calls | Process Creation and Termination | exec() |
| | | fork() |
| | | wait() |
| | | exit() |
| | Process Owner and Group | getuid() |
| | | geteuid() |
| | | getgid() |
| | | getegid() |
| | Process Identity | getpid() |
| | | getppid() |
| | Process Control | signal() |
| | | kill() |
| | | alarm() |
| | Change Working Directory | chdir() |
| Interprocess Communication | Pipelines | pipe() |
| | Messages | msgget() |
| | | msgsnd() |
| | | msgrcv() |
| | | msgctl() |
| | Semaphores | semget() |
| | | semop() |
| | Shared Memory | shmget() |
| | | shmat() |
| | | shmdt() |

// print all error message using “perror()”

```
#include<stdio.h>
int main( )
{
        int i;
        extern int erno, sys_nerr;
        for ( int i=0 ; i<sys_nerr ; ++i)
        {
                fprintf ( stderr, "%3d",i) ;
                errno = I ;
                perror(" ");
        }
        exit (0):
}
```

```
//print all system error messages using the global error message 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>          // defines types used by sys/stat.n
#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 constant 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_IWGRP 0000020             /* 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 you open a file for 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 file's access permissions if 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_WRONLY 1          /* Open the file for writing only */
#define O_RDWR 2            /* Open the file for both reading and writing*/
#define O_NDELAY 04         /* Non-blocking I/O */
#define O_APPEND 010        /* append (writes guaranteed at the end) */
#define O_CREAT 00400       /*open with file create (uses third open arg) */
#define O_TRUNC 01000       /* open with truncation */
#define O_EXCL 02000        /* exclusive open */
```

# PROGRAM – 3

**Objective:** Implement CPU Scheduling Policies:

a) SJF
b) FCFS

## 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 wait in 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 be performed. Each process will be executed according to the length of its hurst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

## Program:

### 1. FCFS SCHEDULING

i. Start
ii. Declare the array size
iii. Read the number of processes to be inserted 4. Read the Burst times of processes
iv. Calculate the waiting time of each process wt[i+1]=bt[i]+wt[i]
v. Calculate the turnaround time of each process tt[i+1]=tt[i]+bt[i+1]
vi. Calculate the average waiting time and average turnaround time.
vii. Display the values
viii. Stop

```
#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];
        }
        aw=w1/n;
        at=t1/n;
        printf("\nbt\t wt\t tt\n");
        for(i=0;i<n;i++)
        {
                wl=w1+wt[i];
                tl=t1+tt[i];
                printf("%d\t%d\t %d\n",bt[i], wt[i],ut[i])
                printf("aw=%fn,at=%\n",aw,at);
                getch();
        }
    }
}
```

**INPUT**

Enter no of processes
3
enter bursttime
12
8
20

**OUTPUT**

| bt | wt | tt |
|---|---|---|
| 12 | 0 | 12 |
| 8 | 12 | 20 |
| 20 | 20 | 40 |

aw=10.666670
at=24.00000

## 2. SJF SCHEDULING

i. Start
ii. Declare the array size
iii. Read the number of processes to be inserted
iv. Read the Burst times of processes
v. Sort the Burst times in ascending order and process with shortest burst time is first executed.
vi. calculate the waiting time of each process . wt[i + 1] = bt[i] + wt[i]
vii. calculate the turnaround time of each process . tt[i + 1] = t[i] + bt[i + 1]
viii. Calculate the average waiting time and average turnaround time.
ix. Display the values
x. Stop

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

```
#include<conio.h>
void main()
{
	int i,j,bt[10],t,n,wt[10], [10], w1=0,11=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]);
	}

	wi[0]=0;
	ut[0]=bt[0];

	for(i=0;i<n;i++)
	{
		wt[i+1]=bt[i]+wt[i];
		tt[i+1]=tt[i]+bt[i+1]:
		w1=w1+wt[i];
		t1=tl+[i];
	}

	aw=w1/n;
	at = t1/n;
```

```
        printf("\nbt\t wit it\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
```

# PROGRAM – 4

**Objective:** Implement CPU Scheduling Policies:

a) Round robin

b) Priority

## Content:

**ROUND ROBIN CPU SCHEDULING ALGORITHM**

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. 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 the system, 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 then 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. ROUND ROBIN SCHEDULING

i. Start
ii. Declare the array size
iii. Read the number of processes to be inserted
iv. Read the burst times of the processes
v. Read the Time Quantum
vi. If, the burst time of a process is greater than time Quantum then subtract time quantum form the burst time
Else, assign the burst time to time quantum.
vii. Calculate the average waiting time and turnaround time of the processes.
viii. Display the values
ix. Stop

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

void main()
{
        int st[10], bt[10], wt[10], tat[10], n, tq;
        int i, count = 0, swt = 0, stat = 0, temp, sq = 0;
        float awt = 0.0, atat = 0.0;
        clrscr();

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

```
printf("Enter burst time for sequences:");
for (i = 0; i < n; i++)
{
	scanf("%d", &bt[i]);
	st[i] = bt[i];
}

printf("Enter time quantum:");
scanf("%d", &tq);

while (1)
{
	for (i = 0, count = 0; i < n; i++)
	{
		temp = tq;
		if (st[i] == 0)
		{
			count++;
			continue;
		}

		if (st[i] > tq)
			st[i] = st[i] - tq;
		else if (st[i] >= 0)
		{
			temp = st[i];
			st[i] = 0;
		}

		sq = sq + temp;
		tat[i] = sq;
	}

	if (n == count)
		break;
}

for (i = 0; i < n; i++)
{
	wt[i] = tat[i] - bt[i];
	swt = swt + wt[i];
	stat = stat + tat[i];
}

awt = (float)swt / n;
atat = (float)stat / n;

printf("Process_no\tBurst time\tWait time\tTurn around time");
for (i = 0; i < n; i++)
```

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

        printf("\nAvg wait time is %f Avg turn around time is %f", awt, atat);

        getch();
}
```

**INPUT:**
Enter number of processes: 4
Enter burst time for sequences: 5 7 3 2
Enter time quantum: 2

**OUTPUT:**

| Process_no | Burst time | Wait time | Turn around time |
|---|---|---|---|
| 1 | 5 | 7 | 12 |
| 2 | 7 | 12 | 19 |
| 3 | 3 | 11 | 14 |
| 4 | 2 | 13 | 15 |

Avg wait time is 10.750000 Avg turn around time is 15.000000

## 2. PRIORITY SCHEDULING

i. Start
ii. Declare the array size
iii. Read the number of processes to be inserted
iv. Read the Priorities of processes
v. Sort the priorities and Burst times in ascending order
vi. Calculate the waiting time of each process wi[i+1]=bt[i]+wt[i]
vii. Calculate the turnaround time of each process u[i+1]=[i]+bt[i+1]
viii. Calculate the average waiting time and average turnaround time.
ix. Display the values
x. Stop

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

void main()
{
        int i, j, n;
        int pno[10], prior[10], bt[10], wt[10], tt[10];
        int w1 = 0, tl = 0;
        float aw, at;

        clrscr();
        printf("Enter the number of processes: ");
        scanf("%d", &n);

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

```
{
	printf("\nThe process %d\n", i + 1);
	printf("Enter the burst time of the process: ");
	scanf("%d", &bt[i]);
	printf("Enter the priority of the process: ");
	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])
		{
			int 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;
		}
	}
}

wt[0] = 0;
tt[0] = bt[0];

for (i = 0; i < n; i++)
{
	wt[i + 1] = bt[i] + wt[i];
	tt[i + 1] = tt[i] + bt[i + 1];
	w1 = w1 + wt[i];
	tl = tl + tt[i];
}

aw = (float)w1 / n;
at = (float)tl / n;

printf("\nJob\tBT\tWT\tTAT\tPriority\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("Average waiting time (aw) = %f\n", aw);
```

```
        printf("Average turnaround time (at) = %f\n", at);

        getch();
}
```

**INPUT:**

```
Enter the number of processes: 3

The process 1
Enter the burst time of the process: 5
Enter the priority of the process: 2

The process 2
Enter the burst time of the process: 3
Enter the priority of the process: 1

The process 3
Enter the burst time of the process: 6
Enter the priority of the process: 3
```

**OUTPUT:**

| Job | BT | WT | TAT | Priority |
|---|---|---|---|---|
| 2 | 3 | 0 | 3 | 1 |
| 1 | 5 | 3 | 8 | 2 |
| 3 | 6 | 8 | 14 | 3 |

```
Average waiting time (aw) = 3.666667
Average turnaround time (at) = 8.333333
```

# PROGRAM – 5

**Objective:** Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

## Content:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.
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.

Available:

- It is a 1-d array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type R;

MAX

- It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process P, may request at most 'k' instances of resource type Rj.

Allocation:

- It is a 2-d array of size 'n*m' that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process P; is currently allocated 'k' instances of resource type Rj

Need:

- It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.
- Need [i, j] = k means process P; currently need 'k' instances of resource type Rj for its execution.
- Need [i, j] = Max [i, j]- Allocation [i, j]

## Program:

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

void main()
{
   int n, r, k, i, p, j, u = 0, s, m;
   int block[10], run[10], active[10], newreq[10];
   int max[10][10], resalloc[10][10], resreq[10][10];
   int totalloc[10], totext[10], simalloc[10];
   //clrscr();
```

```
printf("Enter the number of processes: ");
scanf("%d", &n);

printf("Enter the number of resource classes: ");
scanf("%d", &r);

printf("Enter the total existing resource in each class: ");
for (k = 1; k <= r; k++)
   scanf("%d", &totext[k]);

printf("Enter the allocated resources: ");
for (i = 1; i <= n; i++)
   for (k = 1; k <= r; k++)
      scanf("%d", &resalloc[i][k]);

printf("Enter the process making the new request: ");
scanf("%d", &p);

printf("Enter the requested resource: ");
for (k = 1; k <= r; k++)
   scanf("%d", &newreq[k]);

printf("Enter the processes which are blocked or running:\n");
for (i = 1; i <= n; i++)
   for (k = 1; k <= r; k++)
   {
      if (i != p)
      {
         printf("Process %d\n", i + 1);
         scanf("%d %d", &block[i], &run[i]);
      }
   }

block[p] = 0;
run[p] = 0;

for (k = 1; k <= r; k++)
{
   j = 0;
   for (i = 1; i <= n; i++)
   {
      totalloc[k] = j + resalloc[i][k];
      j = totalloc[k];
   }
}

for (i = 1; i <= n; i++)
{
   if (block[i] == 1 || run[i] == 1)
      active[i] = 1;
```

```
    else
        active[i] = 0;
}

for (k = 1; k <= r; k++)
{
    resalloc[p][k] += newreq[k];
    totalloc[k] += newreq[k];
}

for (k = 1; k <= r; k++)
{
    if (totext[k] < totalloc[k])
    {
        u = 1;
        break;
    }
}

if (u == 0)
{
    for (k = 1; k <= r; k++)
        simalloc[k] = totalloc[k];

    for (s = 1; s <= n; s++)
        for (i = 1; i <= n; i++)
        {
            if (active[i] == 1)
            {
                j = 0;
                for (k = 1; k <= r; k++)
                {
                    if ((totext[k] - simalloc[k]) < (max[i][k] - resalloc[i][k]))
                    {
                        j = 1;
                        break;
                    }
                }

                if (j == 0)
                    active[i] = 0;

                for (k = 1; k <= r; k++)
                    simalloc[k] -= resalloc[i][k];
            }
        }

    m = 0;
    for (k = 1; k <= r; k++)
        resreq[p][k] = newreq[k];
```

```
      printf("Deadlock will not occur");
   }
   else
   {
      for (k = 1; k <= r; k++)
      {
         resalloc[p][k] -= newreq[k];
         totalloc[k] = newreq[k];
      }

      printf("Deadlock will occur");
   }

   getch();
}
```

**INPUT:**

```
Enter the number of processes: 3
Enter the number of resource classes: 2
Enter the total existing resource in each class: 5 3
Enter the allocated resources:
2 1
1 2
3 0
Enter the process making the new request: 2
Enter the requested resource: 1 1
Enter the processes which are blocked or running:
Process 1
0 1
Process 3
1 0
```

**OUTPUT:**

```
Deadlock will occur
```

# PROGRAM – 6

**Objective:** Implement file storage allocation technique:

i. Contiguous(using array)
ii. Linked-list(using linked-list)
iii. 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.

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation

The main idea behind these methods is to provide:

- Efficient disk space utilization.
- Fast access to the file blocks.

## Program:

### SEQUENTIAL FILE ALLOCATION

```
#include<stdio.h>

int main()
{
	int f[50], i, j, st, len, c;

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

X:
	printf("\nEnter the starting block & length of the file: ");
	scanf("%d %d", &st, &len);

	for(j=st; j<(st+len); j++)
	{
		if(f[j] == 0)
		{
			f[j] = 1;
			printf("%d-%d ", j, f[j]);
		}
		else
		{
			printf("Block already allocated\n");
			break;
		}
	}
```

```
        if(j == (st+len))
                printf("\nThe file is allocated to disk");

        printf("\nIf you want to enter more files? (y-1/n-0): ");
        scanf("%d", &c);

        if(c == 1)
                goto X;
        else
                return 0;
}
```

**OUTPUT**

```
Enter the starting block & length of the file: 3 5
3-1 4-1 5-1 6-1 7-1
The file is allocated to disk

If you want to enter more files? (y-1/n-0): 1

Enter the starting block & length of the file: 8 3
8-1 9-1 10-1
The file is allocated to disk

If you want to enter more files? (y-1/n-0): 0
```

## LINKED FILE ALLOCATION

```
#include <stdio.h>

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

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

        printf("Enter how many blocks are already allocated: ");
        scanf("%d", &p);

        printf("\nEnter the block numbers 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("\nIf you want to enter one more file? (yes-1/no-0): ");
        scanf("%d", &c);

        if (c == 1)
                goto X;
        else
                return 0;
}
```

**OUTPUT**

```
Enter how many blocks are already allocated: 3

Enter the block numbers that are already allocated: 5 10 15

Enter the starting index block & length: 7 5
7-1
8-1
9-1
10 -> File is already allocated
11-1

If you want to enter one more file? (yes-1/no-0): 0
```

## INDEXED ALLOCATION TECHNIQUE

```
#include<stdio.h>

int f[50], kj, inde[50], n, c, count = 0, p;

int main()
{
   clrscr();
   for (int i = 0; i < 50; i++)
```

```
    f[i] = 0;

X:
   printf("Enter index block: ");
   scanf("%d", &p);

   if (f[p] == 0)
   {
      f[p] = 1;
      printf("Enter number of files on index: ");
      scanf("%d", &n);
   }
   else
   {
      printf("Block already allocated\n");
      goto X;
   }

   for (int i = 0; i < n; i++)
      scanf("%d", &inde[i]);

   for (int i = 0; i < n; i++)
   {
      if (f[inde[i]] == 1)
      {
         printf("Block already allocated");
         goto X;
      }
   }

   for (int j = 0; j < n; j++)
      f[inde[j]] = 1;

   printf("%d allocated\n", n);
   printf("File indexed:\n");

   for (int k = 0; k < n; k++)
      printf("%d-%d: %d\n", p, inde[k], f[inde[k]]);

   printf("Enter 1 to enter more files and 0 to exit: ");
   scanf("%d", &c);

   if (c == 1)
      goto X;
   else
      exit();

   getch();
}
```

**OUTPUT**

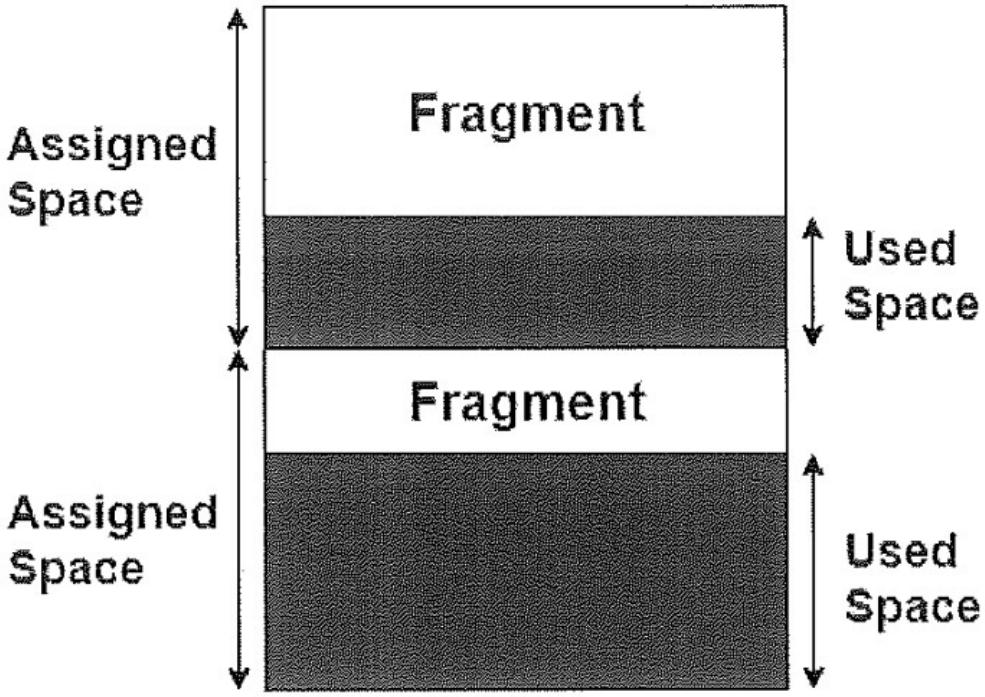
```
Enter index block: 2
Enter number of files on index: 3
8
12
15
3 allocated
File indexed:
2-8: 1
2-12: 1
2-15: 1
Enter 1 to enter more files and 0 to exit: 1
Enter index block: 5
Enter number of files on index: 2
10
20
2 allocated
File indexed:
5-10: 1
5-20: 1
Enter 1 to enter more files and 0 to exit: 0
```

# PROGRAM – 7

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

**Content:** There are two types of fragmentation in OS which are given as: Internal fragmentation, and External fragmentation.

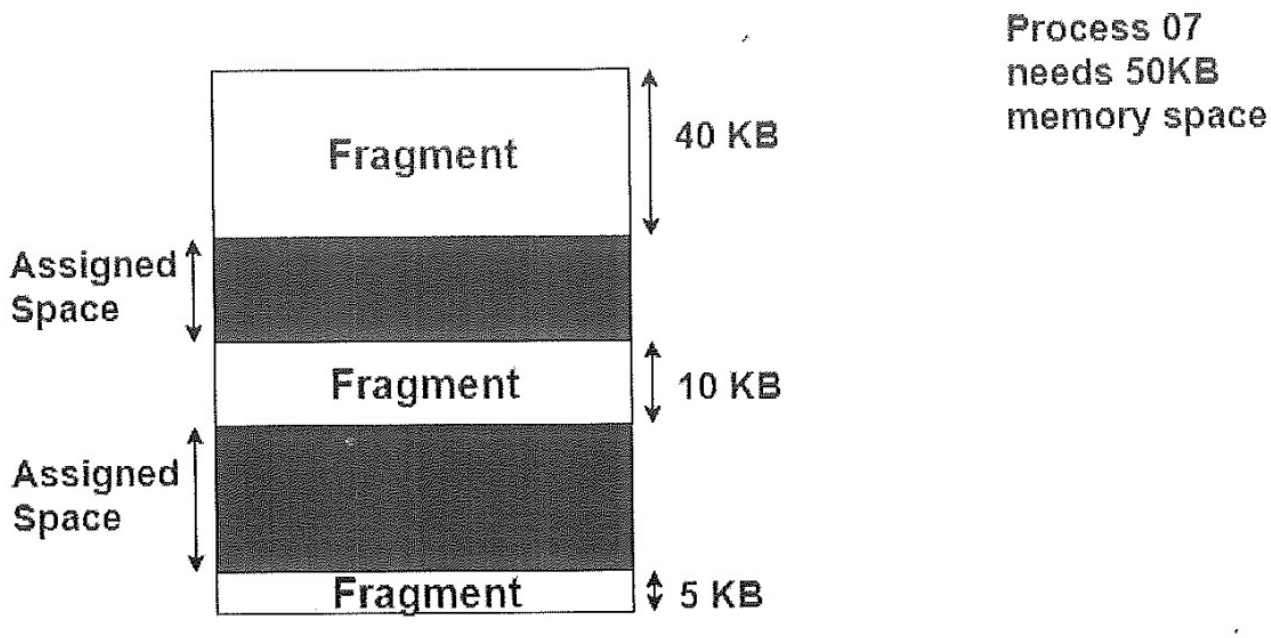
**Internal Fragmentation:**
Internal fragmentation happens when the memory is split into mounted sized blocks. Whenever a method request for the memory, the mounted sized block is allotted to the method. just in case the memory allotted to the method is somewhat larger than the memory requested, then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




Internal Fragmentation

The above diagram clearly shows the internal fragmentation because the difference between memory allocated and required space or memory is called 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 or best-fit memory allocation strategy it'll cause external fragmentation.




In above diagram, we can see that, there is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous. Here, we use compaction, paging or segmentation to use the free space to run a process.

**Difference between Internal fragmentation and External fragmentation:**

| S.NO | Internal fragmentation | External fragmentation |
|---|---|---|
| 1. | In internal fragmentation fixed-sized memory, blocks square measure appointed to process. | In external fragmentation, variable-sized memory blocks square measure appointed to method. |
| 2. | Internal fragmentation happens when the method or process is larger than the memory. | External fragmentation happens when the method or process is removed. |
| 3. | The solution of internal fragmentation is best-fit block. | Solution of external fragmentation is compaction, paging and segmentation. |
| 4. | Internal fragmentation occurs when memory is divided into fixed sized partitions. | External fragmentation occurs when memory is divided into variable size partitions based on the size of processes. |
| 5. | The difference between memory allocated and required space or memory is called Internal fragmentation. | The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, is called External fragmentation. |

# PROGRAM – 8

**Objective:** Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit
b) Best-fit
c) 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, when a partition 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 arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap 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. Worst-fit chooses the largest available block.

## Program:

### 1. First Fit algorithm in Memory Management

**Implementation:**

1. Input memory blocks with size and processes with size.
2. Initialize all memory blocks as free.
3. Start by picking each process and check if it can be assigned to current block.
4. If size-of-process <= size-of-block if yes then assign and check for next process.
5. If not then keep checking the further blocks.

```
// C++ implementation of First-Fit algorithm
#include <iostream>
using namespace std;

void firstFit(int blockSize[], int m, int processSize[], int n)
{
   int allocation[n];
   memset(allocation, -1, sizeof(allocation));

   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < m; j++)
      {
         if (blockSize[j] >= processSize[i])
         {
            allocation[i] = j;
            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" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "No Allocated";
        cout << endl;
    }
}

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);

    return 0;
}
```

**OUTPUT**

| Process No. | Process Size | Block no. |
|---|---|---|
| 1 | 212 | 1 |
| 2 | 417 | 2 |
| 3 | 112 | 1 |
| 4 | 426 | No Allocated |

## 2. Best Fit algorithm in Memory Management

**Implementation:**

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and find the minimum block size that can be assigned to current process i.e., find min(bockSize[1], blockSize[2].....blockSize[n]) > processSize[current], if found then assign it to the current process.
4. If not then leave that process and keep checking the further processes.

```
// C++ implementation of Best-Fit algorithm
#include <iostream>
using namespace std;
```

```
void bestFit(int blockSize[], int m, int processSize[], int n)
{
   int allocation[n];
   memset(allocation, -1, sizeof(allocation));

   for (int i = 0; i < n; i++)
   {
      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 (bestIdx != -1)
      {
         allocation[i] = bestIdx;
         blockSize[bestIdx] -= 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;
   }
}

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;
}
```

**OUTPUT**

| Process No. | Process Size | Block no. |
|---|---|---|
| 1 | 212 | 1 |
| 2 | 417 | 2 |
| 3 | 112 | 1 |
| 4 | 426 | Not Allocated |

## 3. Worst Fit algorithm in Memory Management

**Implementation:**

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(bockSize[1], blockSize[2].....blockSize[n]) > processSize[current], if found then assign it to the current process.
4. If not then leave that process and keep checking the further processes.

```
// C++ implementation of worst-Fit algorithm
#include <iostream>
using namespace std;

void worstFit(int blockSize[], int m, int processSize[], int n)
{
   int allocation[n];
   memset(allocation, -1, sizeof(allocation));

   for (int i = 0; i < n; i++)
   {
      int wstIdx = -1;
      for (int j = 0; j < m; j++)
      {
         if (blockSize[j] >= processSize[i])
         {
            if (wstIdx == -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;
    }
}

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;
}
```

**OUTPUT**

| Process No. | Process Size | Block no. |
|---|---|---|
| 1 | 212 | 5 |
| 2 | 417 | 2 |
| 3 | 112 | 2 |
| 4 | 426 | Not Allocated |

# PROGRAM – 9

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

## Content:

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores.
A semaphore S is an integer variable that can be accessed only through two standard operations: wait() and signal().
The wait() operation reduces the value of semaphore by I and the signal() operation increases its value by 1.

```
wait(S) {
        while($<=0); // busy waiting
        S--;
}
```

```
signal(S) {
        S++;
}
```

Semaphores are of two types:

**1. Binary Semaphore-** This is similar to mutex lock but not the same thing. It can have only two values- 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

**2. Counting Semaphore-** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

## Program:

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

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

int wait(int);
int signal(int);
void producer();
void consumer();

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

```
      case 1:
         if (mutex == 1 && empty != 0)
            producer();
         else
            printf("Buffer is full!!\n");
         break;
      case 2:
         if (mutex == 1 && full != 0)
            consumer();
         else
            printf("Buffer is empty!!\n");
         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\n", x);
   mutex = signal(mutex);
}

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

**Output:**

1. Producer
2. Consumer
3. Exit

Enter your choice: 1

Producer produces the item 1

Enter your choice: 2

Consumer consumes item 1

Enter your choice: 1

Producer produces the item 2

Enter your choice: 1

Producer produces the item 3

Enter your choice: 2

Consumer consumes item 2

Enter your choice: 3

# PROGRAM – 10

**Objective:** Implementation of Resource Allocation Graph (RAG).

## Content:

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

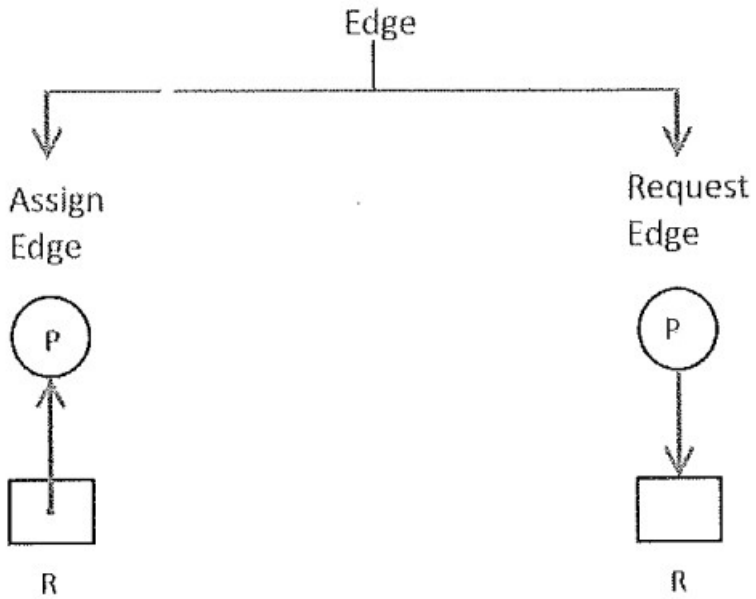
RAG also contains vertices and edges, in RAG vertices are two type -

**1. Process vertex -** Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
**2. Resource vertex -** Every resource will be represented as a resource vertex. It is also two type -

- **Single instance type resource -** It represents as a box, inside the box, there will be one dot. So the number of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource -** It also represents as a box, inside the box, there will be many dots present.

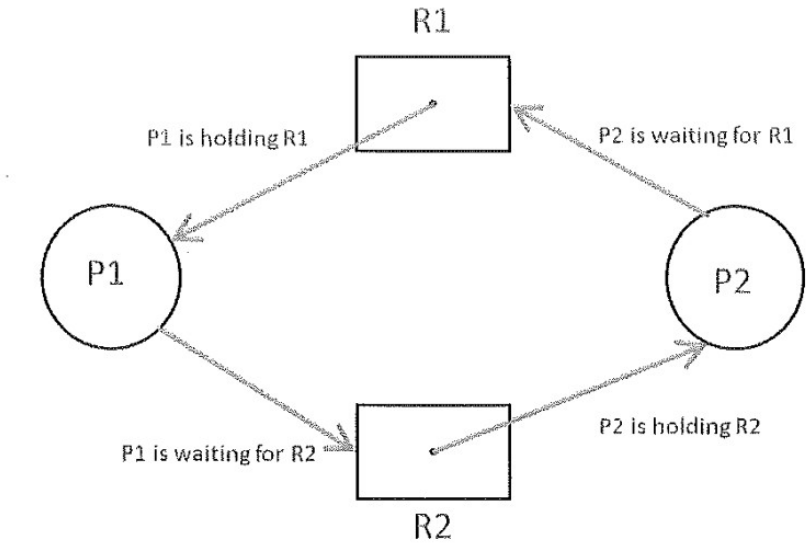
Now coming to the edges of RAG. There are two types of edges in RAG-

**1. Assign Edge -** If you already assign a resource to a process then it is called Assign edge.
**2. Request Edge -** It means in future the process might want some resource to complete the execution, that is called request edge.



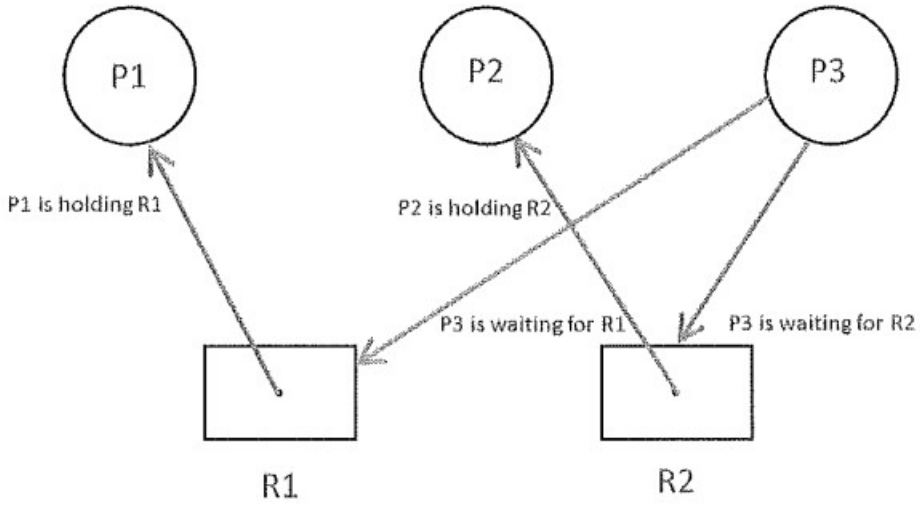

So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

**Example 1: (Single-instances RAG)**




SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

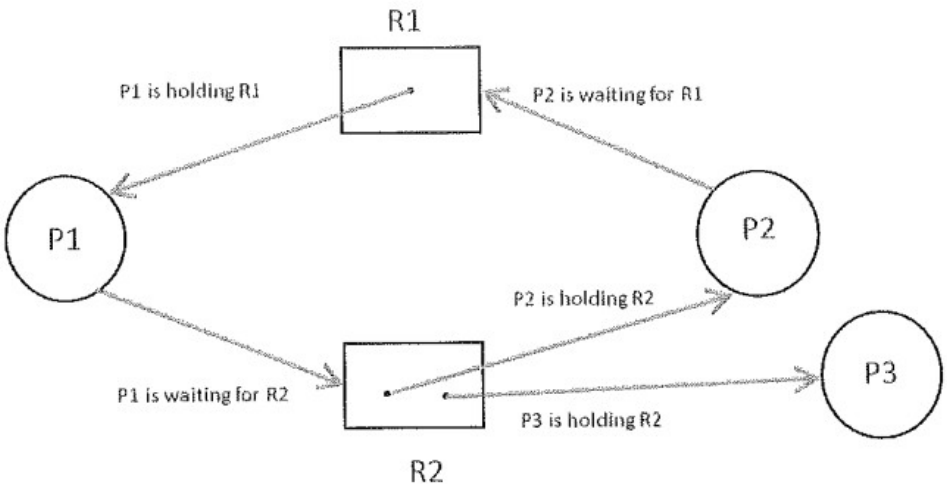
If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.




SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.
So cycle in single-instance resource type is the sufficient condition for deadlock.

**Example 2: (Multi-instances RAG)**




MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix

| Process | Allocation | | Request | |
|---|---|---|---|---|
| | Resource | | Resource | |
| | R1 | R2 | R1 | R2 |
| P1 | 1 | 0 | 0 | 1 |
| P2 | 0 | 1 | 1 | 0 |
| P3 | 0 | 1 | 0 | 0 |

**Allocation matrix -**

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.
- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

**Request matrix -**

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

So now available resource is = (0, 0).

**Checking deadlock (safe or not) –**

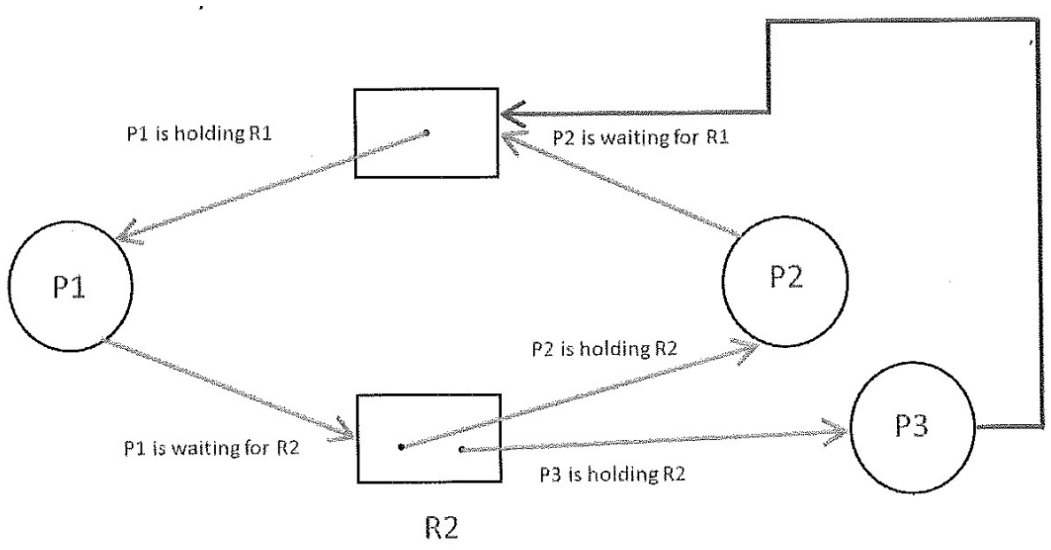
Available = 0 0 (As P3 does not require any extra resource to complete the execution and after
P3 0 1 completion P3 release its own resource)

New Available = 0 1 (As using new available resource we can satisfy the requirment of process P1
P1 1 0 and P1 also release its prevous resource)

New Available = 1 1 (Now easily we can satisfy the requirement of process P2)
P2 0 1

New Available = 1 2

So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore in multi-instance resource cycle is not sufficient condition for deadlock.




MULTI INSTANCES WITH DEADLOCK

Above example is the same as the previous example except that, the process P3 requesting for resource R1.

So the table becomes as shown in below.

| Process | Allocation | | Request | |
|---|---|---|---|---|
| | Resource | | Resource | |
| | R1 | R2 | R1 | R2 |
| P1 | 1 | 0 | 0 | 1 |
| P2 | 0 | 1 | 1 | 0 |
| P3 | 0 | 1 | 1 | 0 |

So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement. Therefore, it is in deadlock.
Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

## 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);
		}
	}

	// Using waitpid() and printing exit status of children.
	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;
}
```