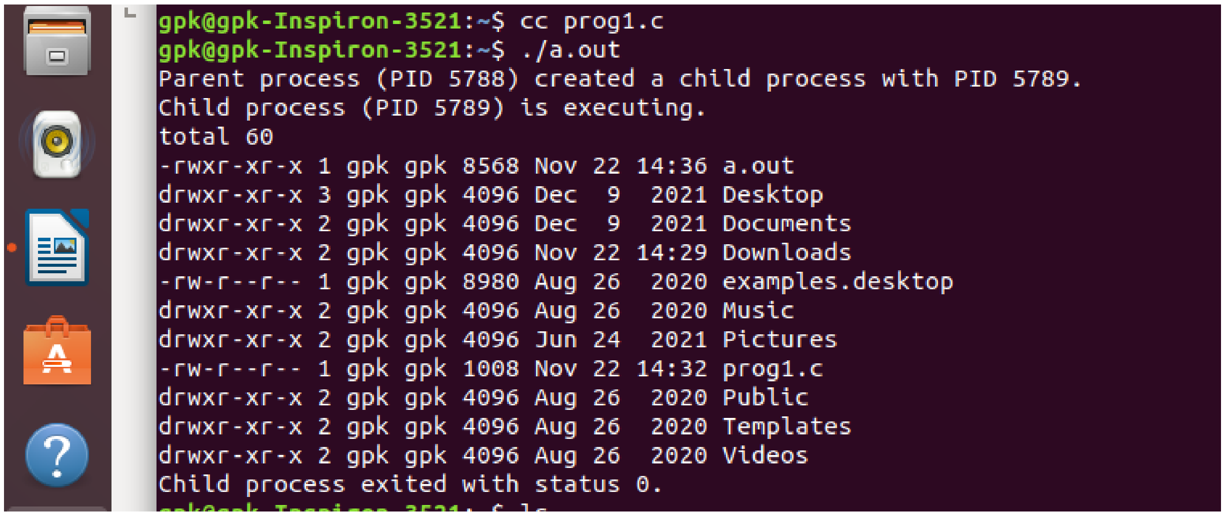
Prog 1: Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
   pid_t child_pid;

   // Create a child process using fork()
   child_pid = fork();

   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }

   if (child_pid == 0) {
      // This is the child process
      printf("Child process (PID %d) is executing.\n", getpid());

      // Execute a program using exec()
      char *args[] = {"/bin/ls", "-l", NULL};
      execv(args[0], args);

      // If execv() fails
      perror("Exec failed");
      return 1;
   } else {
      // This is the parent process
      printf("Parent process (PID %d) created a child process with PID %d.\n", getpid(), child_pid);

      // Wait for the child process to finish using wait()
      int status;
      wait(&status);

      if (WIFEXITED(status)) {
         printf("Child process exited with status %d.\n", WEXITSTATUS(status));
      }
   }

   return 0;
}
```

Explanation:

1. The program creates a child process using fork(). fork() creates an identical copy of the current process, and both the parent and child processes continue to execute from the point of the fork() call.
2. In the child process, we use execv() to replace the child process's memory with a new program (/bin/ls in this case). This effectively starts a new program in the child process.
3. In the parent process, we wait for the child process to finish using wait(). This allows the parent to synchronize its execution with the child.
4. After the child process exits, we check if it exited normally using WIFEXITED(). If it did, we print the exit status.

Compile and run this program using a C compiler. It will create a child process, execute the ls -l command in the child process, and then print the exit status when the child process finishes.



```
gpk@gpk-Inspiron-3521:~$ cc prog1.c
gpk@gpk-Inspiron-3521:~$ ./a.out
Parent process (PID 5788) created a child process with PID 5789.
Child process (PID 5789) is executing.
total 60
-rwxr-xr-x 1 gpk gpk 8568 Nov 22 14:36 a.out
drwxr-xr-x 3 gpk gpk 4096 Dec  9  2021 Desktop
drwxr-xr-x 2 gpk gpk 4096 Dec  9  2021 Documents
drwxr-xr-x 2 gpk gpk 4096 Nov 22 14:29 Downloads
-rw-r--r-- 1 gpk gpk 8980 Aug 26  2020 examples.desktop
drwxr-xr-x 2 gpk gpk 4096 Aug 26  2020 Music
drwxr-xr-x 2 gpk gpk 4096 Jun 24  2021 Pictures
-rw-r--r-- 1 gpk gpk 1008 Nov 22 14:32 prog1.c
drwxr-xr-x 2 gpk gpk 4096 Aug 26  2020 Public
drwxr-xr-x 2 gpk gpk 4096 Aug 26  2020 Templates
drwxr-xr-x 2 gpk gpk 4096 Aug 26  2020 Videos
Child process exited with status 0.
```

Prg 2: Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

DESCRIPTION

Assume all the processes arrive at the same time.

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 be 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.

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.

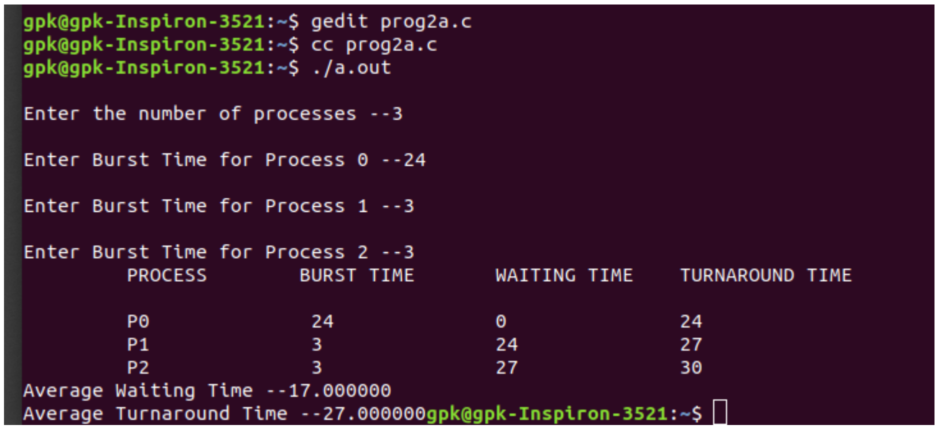
PRIORITY CPU SCHEDULING ALGORITHM

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.

FCFS CPU SCHEDULING ALGORITHM

```
#include<stdio.h>

int main()

{

int bt[20], wt[20], tat[20], i,n;

float wtavg, tatavg;

printf("\nEnter the number of processes --");

scanf("%d", &n);

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

{
```

```
printf("\nEnter Burst Time for Process %d --", i);

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

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1]+bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg +tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time --%f", wtavg/n);
printf("\nAverage Turnaround Time --%f", tatavg/n);
return 0;
}
```

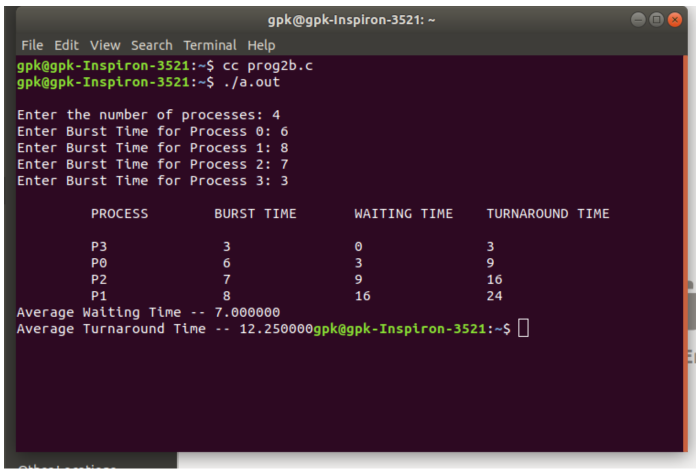


```
gpk@gpk-Inspiron-3521:~$ gedit prog2a.c
gpk@gpk-Inspiron-3521:~$ cc prog2a.c
gpk@gpk-Inspiron-3521:~$ ./a.out

Enter the number of processes --3

Enter Burst Time for Process 0 --24

Enter Burst Time for Process 1 --3

Enter Burst Time for Process 2 --3
         PROCESS        BURST TIME       WAITING TIME    TURNAROUND TIME

         P0              24              0               24
         P1              3               24              27
         P2              3               27              30
Average Waiting Time --17.000000
Average Turnaround Time --27.000000gpk@gpk-Inspiron-3521:~$
```

SJF CPU SCHEDULING ALGORITHM

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

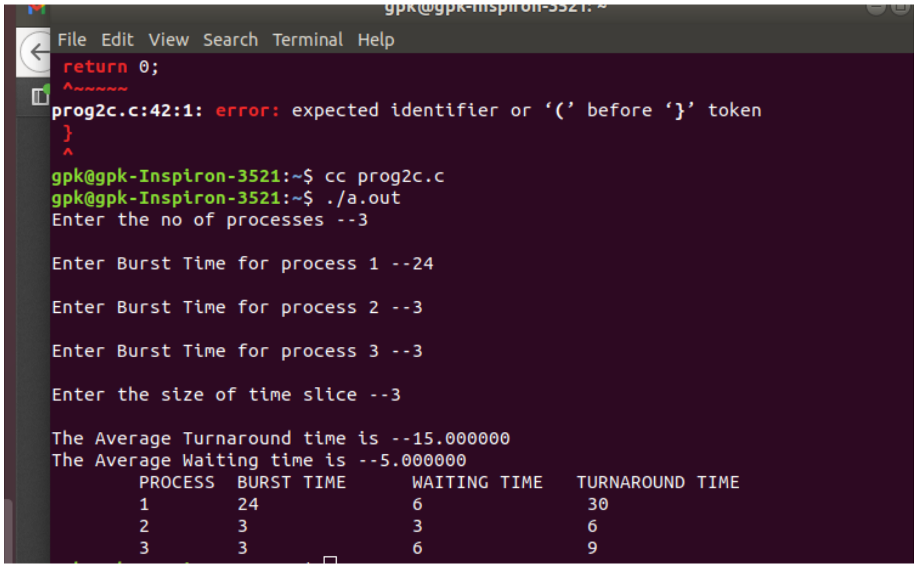
```
int main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
printf("\nEnter the number of processes --");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d --", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
 bt[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
temp=p[i];
 p[i]=p[k];
 p[k]=temp;
wt[i] = wt[i-1]+bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg +tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time --%f", wtavg/n);
printf("\nAverage Turnaround Time --%f", tatavg/n);
return 0;
}
```



```
gpk@gpk-Inspiron-3521:~$ cc prog2b.c
gpk@gpk-Inspiron-3521:~$ ./a.out

Enter the number of processes: 4
Enter Burst Time for Process 0: 6
Enter Burst Time for Process 1: 8
Enter Burst Time for Process 2: 7
Enter Burst Time for Process 3: 3

         PROCESS         BURST TIME       WAITING TIME     TURNAROUND TIME

         P3               3                0                3
         P0               6                3                9
         P2               7                9                16
         P1               8                16               24
Average Waiting Time -- 7.000000
Average Turnaround Time -- 12.250000gpk@gpk-Inspiron-3521:~$
```

ROUND ROBIN CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
int main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
printf("Enter the no of processes --");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d --", i+1);
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice --");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else{
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++)
{
wa[i]=tat[i]-ct[i];
att+=tat[i];
awt+=wa[i];
}
printf("\nThe Average Turnaround time is --%f",att/n);
printf("\nThe Average Waiting time is --%f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
return 0;
}
```

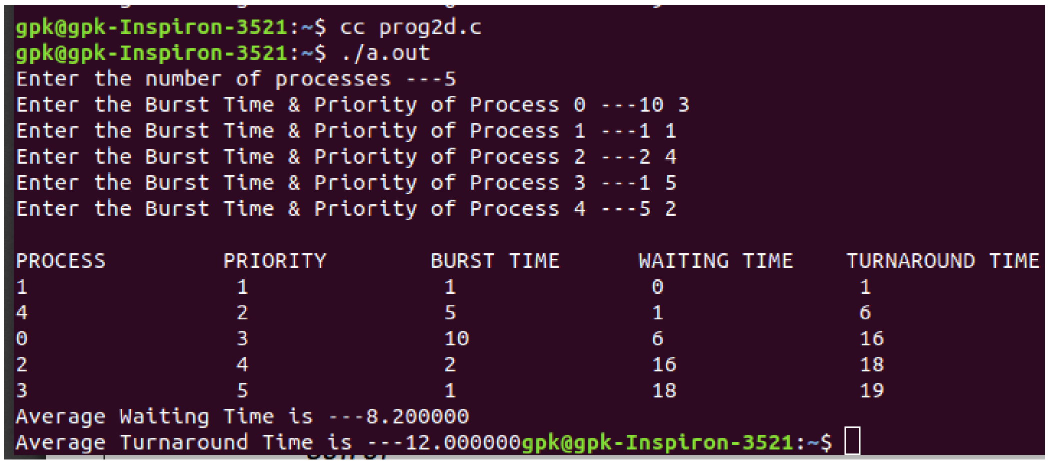


```
 return 0;
 ^~~~~~
prog2c.c:42:1: error: expected identifier or '(' before '}' token
 }
 ^
gpk@gpk-Inspiron-3521:~$ cc prog2c.c
gpk@gpk-Inspiron-3521:~$ ./a.out
Enter the no of processes --3

Enter Burst Time for process 1 --24

Enter Burst Time for process 2 --3

Enter Burst Time for process 3 --3

Enter the size of time slice --3

The Average Turnaround time is --15.000000
The Average Waiting time is --5.000000
        PROCESS  BURST TIME      WAITING TIME   TURNAROUND TIME
        1        24              6               30
        2        3               3               6
        3        3               6               9
```

PRIORITY CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
int main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg;
```

```
printf("Enter the number of processes ---");
scanf("%d",&n);
for(i=0;i<n;i++)
{
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d ---",i);
scanf("%d %d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k])
{
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is ---%f",wtavg/n);
printf("\nAverage Turnaround Time is ---%f",tatavg/n);
return 0;
}
```



```
gpk@gpk-Inspiron-3521:~$ cc prog2d.c
gpk@gpk-Inspiron-3521:~$ ./a.out
Enter the number of processes ---5
Enter the Burst Time & Priority of Process 0 ---10 3
Enter the Burst Time & Priority of Process 1 ---1 1
Enter the Burst Time & Priority of Process 2 ---2 4
Enter the Burst Time & Priority of Process 3 ---1 5
Enter the Burst Time & Priority of Process 4 ---5 2

PROCESS         PRIORITY        BURST TIME      WAITING TIME    TURNAROUND TIME
1                1               1               0               1
4                2               5               1               6
0                3               10              6               16
2                4               2               16              18
3                5               1               18              19
Average Waiting Time is ---8.200000
Average Turnaround Time is ---12.000000gpk@gpk-Inspiron-3521:~$
```

Prog 3: Develop a C program to simulate producer-consumer problem using semaphores.

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

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
sem_t mutex, full, empty;

void *producer(void *arg)
{
   int item = 0;
   while (1)
   {
      sem_wait(&empty);
      sem_wait(&mutex);

      // Produce item and add to buffer
      buffer[item % BUFFER_SIZE] = item;
      printf("Produced item %d\n", item);
      item++;

      sem_post(&mutex);
      sem_post(&full);
   }
}

void *consumer(void *arg)
{
   while (1)
   {
      sem_wait(&full);
      sem_wait(&mutex);

      // Consume item from buffer
      int item = buffer[item % BUFFER_SIZE];
      printf("Consumed item %d\n", item);

      sem_post(&mutex);
      sem_post(&empty);
   }
}

int main()
{
   sem_init(&mutex, 0, 1);
```

```
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

    pthread_t producer_thread, consumer_thread;

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);

    return 0;
}
```

The *sem_init*, *sem_wait*, *sem_post*, and *sem_destroy* functions to implement synchronization using semaphores. The producer produces items, adds them to the buffer, and the consumer consumes items from the buffer. The *mutex* semaphore ensures that only one thread accesses the critical section at a time, and *full* and *empty* semaphores control the number of items in the buffer.

Prog 4: Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

/*Make sure to compile and run the programs separately as writer and reader processes in different terminal windows. Ensure that the reader process is executed after the writer process to read the message from the FIFO.*/

```
/*Writer Process*/
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
   int fd;
   char buf[1024];
   /* create the FIFO (named pipe) */
   char *myfifo = "/tmp/myfifo";
   mkfifo(myfifo, 0666);
   printf("Run Reader process to read the FIFO File\n");
   fd = open(myfifo, O_WRONLY);
   write(fd, "Hi", sizeof("Hi") - 1); // Corrected size This ensures that only the actual characters in the
string are written to the FIFO.
   /* write "Hi" to the FIFO */
   close(fd);
   unlink(myfifo); /* remove the FIFO */
   return 0;
}

/*Reader Process*/
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_BUF 1024

int main()
{
   int fd;
   char *myfifo = "/tmp/myfifo";
   char buf[MAX_BUF];

   // Open the FIFO file for reading
   fd = open(myfifo, O_RDONLY);
```

```
    // Read the content from the FIFO into the buffer
    read(fd, buf, MAX_BUF);

    // Print the content
    printf("Writer: %s\n", buf);

    // Close the file descriptor
    close(fd);

    return 0;
}
```

Prog 5: Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

Initialization:

- Variables and arrays are initialized.
- User inputs are received.

Resource Allocation Simulation:

- The program simulates the allocation of new resources to process p.
- It checks if the system will enter a deadlock by simulating the resource allocation for each process.
- If the system will not enter a deadlock, it prints the message "Deadlock will not occur" and updates the resource allocation accordingly.
- If a deadlock will occur, it prints the message "Deadlock will occur" and updates the resource allocation accordingly.

Output:

- The final output indicates whether a deadlock will occur or not.

```
#include <stdio.h>
int main() {
  int n, r, i, j, k, p, u = 0, s = 0, 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];

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

  printf("Enter the total existing resources 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 resources:");
  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++) {
    if (i != p) {
        printf("Process %d (Blocked/Running): ", i);
        scanf("%d", &block[i]);
        printf("Process %d (Running): ", i);
        scanf("%d", &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] < 0) {
        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\n");
} else {
    for (k = 1; k <= r; k++) {
        resalloc[p][k] = newreq[k];
        totalloc[k] = newreq[k];
    }
    printf("Deadlock will occur\n");
}

return 0;
}
```

Variables:

- n: Number of processes.
- r: Number of resource classes.
- i, j, k: Loop control variables.
- p: The process making a new resource request.
- u, s, m: Flags and counters.
- Arrays:
    - block: Indicates whether a process is blocked.
    - run: Indicates whether a process is running.
    - active: Indicates whether a process is active.
    - newreq: The new resource request made by process p.
    - max: Maximum demand of each process for each resource.
    - resalloc: Currently allocated resources to each process.
    - resreq: Resources requested by each process.
    - totext: Total existing resources in each class.
    - totalloc: Total allocated resources in each class.
    - simalloc: Simulated allocation of resources.

User Input:

- The user is prompted to enter the number of processes (n), the number of resource classes (r), and the total existing resources in each class (totext).

- The user is then asked to input the currently allocated resources for each process (resalloc).
- The process making the new resource request (p) is specified, and the user provides the details of the new resource request (newreq).
- The user provides information about the processes that are blocked or running (block and run).

Resource Allocation Simulation:

- The program simulates the allocation of the new resources to process p.
- It checks whether the system will enter a deadlock or not based on the available resources.

Output:

- The program prints whether a deadlock will occur or not.

```
Enter the no of processes:4
Enter the no ofresource classes:3
Enter the total existed resource in each class:3 2 2
Enter the allocated resources:1 0 0 5 1 1 2 1 1 0 0 2
Enter the process making the new request:2
Enter the requested resource:1 1 2
Enter the process which are n blocked or running:process 2:
1 2
process 4:
1 0
process 5:
1 0
Deadlock will occur
```

Prog 6: Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION

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

a)WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - First Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
File No      File Size      Block No         Block Size        Fragment
1                  1          1                   5                   4
2                  4          3                   7                   3
```

b) Best-fit

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
clrscr();
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
if(lowest>temp)
{
ff[i]=j;

lowest=temp;
}
}
}
frag[i]=lowest;
bf[ff[i]]=1;
lowest=10000;
}
printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");
for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
File No         File Size       Block No        Block Size      Fragment
1                 1               2               2               1
2                 4               1               5               1
```

c) First-fit

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{

for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
File No          File Size        Block No        Block Size        Fragment
1                  1                 3               7                 6
2                  4                 1               5                 1
```

Prog 7: Develop a C program to simulate page replacement algorithms:
a) FIFO b) LRU

```
#include <stdio.h>

#define MAX_FRAMES 3 // Number of frames in memory

int fifoPageReplacement(int pages[], int n, int capacity) {
   int frame[MAX_FRAMES];
   int pageFaults = 0;
   int frameIndex = 0;

   for (int i = 0; i < MAX_FRAMES; ++i) {
      frame[i] = -1; // Initialize frames to -1 (empty frame)
   }

   for (int i = 0; i < n; ++i) {
      int page = pages[i];
      int found = 0;

      for (int j = 0; j < capacity; ++j) {
         if (frame[j] == page) {
            found = 1;
            break;
         }
      }

      if (found == 0) {
         frame[frameIndex] = page;
         frameIndex = (frameIndex + 1) % capacity; // Circular queue for FIFO
         ++pageFaults;
      }
   }

   return pageFaults;
}

int lruPageReplacement(int pages[], int n, int capacity) {
   int frame[MAX_FRAMES];
   int pageFaults = 0;
   int used[MAX_FRAMES];

   for (int i = 0; i < MAX_FRAMES; ++i) {
      frame[i] = -1;
      used[i] = 0; // Initialize used bits to 0
   }

   for (int i = 0; i < n; ++i) {
      int page = pages[i];
```

```
        int found = 0;

        for (int j = 0; j < capacity; ++j) {
            if (frame[j] == page) {
                found = 1;
                used[j] = i + 1; // Update the used time for the page
                break;
            }
        }

        if (found == 0) {
            int lruIndex = 0;
            for (int j = 1; j < capacity; ++j) {
                if (used[j] < used[lruIndex]) {
                    lruIndex = j; // Find the least recently used page
                }
            }
            frame[lruIndex] = page;
            used[lruIndex] = i + 1;
            ++pageFaults;
        }
    }

    return pageFaults;
}

int main() {
    int pages[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = MAX_FRAMES;

    printf("FIFO Page Replacement Algorithm - Page Faults: %d\n", fifoPageReplacement(pages, n, 
capacity));
    printf("LRU Page Replacement Algorithm - Page Faults: %d\n", lruPageReplacement(pages, n, 
capacity));

    return 0;
}
```

Prog 8: Simulate following File Organization Techniques

a) Single level directory b) Two level directory

PROGRAM

1. SINGLE LEVEL DIRECTORY ORGANIZATION

```
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;
void main()
{
int i,ch;
char f[30];
clrscr();
dir.fcnt = 0;
printf("\nEnter name of directory -- ");
scanf("%s", dir.dname);
while(1)
{
printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter your choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\n Enter the name of the file -- ");
scanf("%s",dir.fname[dir.fcnt]);
dir.fcnt++;
break;
case 2: printf("\n Enter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is deleted ",f);
strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
else
dir.fcnt--;
break;
case 3: printf("\n Enter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is found ", f);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
break;
case 4: if(dir.fcnt==0)
printf("\n Directory Empty");
else
{
printf("\n The Files are -- ");
for(i=0;i<dir.fcnt;i++)
printf("\t%s",dir.fname[i]);
}
break;
default: exit(0);
}
}
getch();
}
```

OUTPUT:

```
Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- A

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- B

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 4

The Files are -- A B C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 3

Enter the name of the file – ABC

File ABC not found
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 2

Enter the name of the file – B
File B is deleted

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 5
```

2. TWO LEVEL DIRECTORY ORGANIZATION

```
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir[10];
void main()
{
int i,ch,dcnt,k;
char f[30], d[30];
clrscr();
dcnt=0;
while(1)
{
printf("\n\n 1. Create Directory\t 2. Create File\t 3. Delete File");
printf("\n 4. Search File \t \t 5. Display \t 6. Exit \t Enter your choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\n Enter name of directory -- ");
scanf("%s", dir[dcnt].dname);
dir[dcnt].fcnt=0;
dcnt++;
printf("Directory created");
break;
case 2: printf("\n Enter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",dir[i].fname[dir[i].fcnt]);
dir[i].fcnt++;
printf("File created");
break;
}
if(i==dcnt)
printf("Directory %s not found",d);
break;
case 3: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is deleted ",f);
dir[i].fcnt--;
strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
goto jmp;
}
}
printf("File %s not found",f);
goto jmp;
}
}
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{

if(strcmp(d,dir[i].dname)==0)
{
printf("Enter the name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is found ",f);
goto jmp1;
}
}
printf("File %s not found",f);
goto jmp1;
}
}
printf("Directory %s not found",d);
jmp1: break;
case 5: if(dcnt==0)
printf("\nNo Directory's ");
else
{
printf("\nDirectory\tFiles");
for(i=0;i<dcnt;i++)
{
printf("\n%s\t\t",dir[i].dname);
for(k=0;k<dir[i].fcnt;k++)
printf("\t%s",dir[i].fname[k]);
}
}
break;
default:exit(0);
}
}
getch();
}
```

OUTPUT:

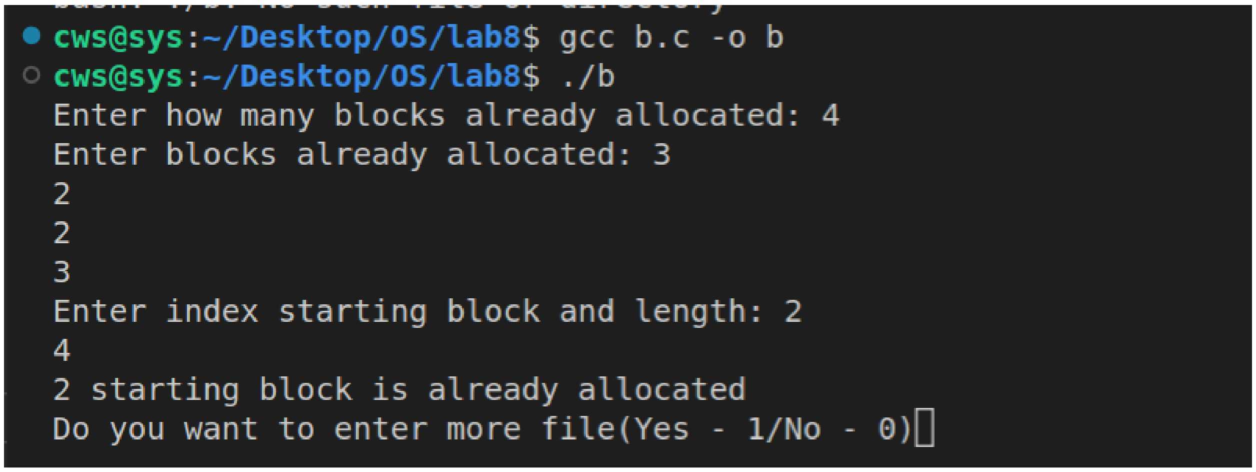
```
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR1
Directory created


1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR2

Directory created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A2
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR2
Enter name of the file -- B1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 5
Directory Files
DIR1 A1 A2
DIR2 B1

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 4

Enter name of the directory – DIR
Directory not found

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 3
Enter name of the directory – DIR1
Enter name of the file -- A2

File A2 is deleted
```

```
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice – 6
```

Prog 9: Develop a C program to simulate the Linked file allocation strategies.

Each file in a linked allocation is a linked list of disk blocks. A pointer to the file's first and, optionally, end block is present in the directory. For instance, a file of five blocks beginning at block 4 may go on to block 7, then block 16, then block 10, and then block 27.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int f[50], p, i, st, len, j, c, k, a;
for (i = 0; i < 50; i++)
f[i] = 0;
printf("Enter how many blocks already allocated: ");
scanf("%d", &p); printf("Enter blocks already allocated: ");
for (i = 0; i < p; i++)
{
scanf("%d", &a); f[a] = 1;
}
x:
printf("Enter index starting block and length: ");
scanf("%d%d", &st, &len);
k = len;
if (f[st] == 0)
{
for (j = st; j < (st + k); j++)
{
if (f[j] == 0)
{
f[j] = 1;
printf("%d-------->%d\n", j, f[j]);
}
else
{
printf("%d Block is already allocated \n", j);
k++;
}
}
}
else
printf("%d starting block is already allocated \n", st);
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if (c == 1)
goto x;
else
exit(0);
return 0;
}
```



```
cws@sys:~/Desktop/OS/lab8$ gcc b.c -o b
cws@sys:~/Desktop/OS/lab8$ ./b
Enter how many blocks already allocated: 4
Enter blocks already allocated: 3
2
2
3
Enter index starting block and length: 2
4
2 starting block is already allocated
Do you want to enter more file(Yes - 1/No - 0)
```

Prog 10: Develop a C program to simulate SCAN disk scheduling algorithm.

In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.

At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

The SCAN algorithm is sometimes called the elevator algorithm, since the disk arms behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

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

int main() {
  int i, j, sum = 0, n;
  int d[20];
  int disk; // loc of head
  int temp, max;
  int dloc; // loc of disk in array

  printf("Enter the number of locations: ");
  scanf("%d", &n);
  printf("Enter the position of the head: ");
  scanf("%d", &disk);
  printf("Enter the elements of the disk queue:\n");

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

  d[n] = disk;
  n = n + 1;

  for (i = 0; i < n; i++) {
    for (j = i; j < n; j++) {
      if (d[i] > d[j]) {
        temp = d[i];
        d[i] = d[j];
        d[j] = temp;
      }
    }
  }

  max = d[n - 1];

  for (i = 0; i < n; i++) {
        if (disk == d[i]) {
            dloc = i;
            break;
        }
    }

    printf("The path of the head movement:\n");

    for (i = dloc; i >= 0; i--) {
        printf("%d --> ", d[i]);
        sum += abs(disk - d[i]);
        disk = d[i];
    }
    printf("0 --> ");

    for (i = dloc + 1; i < n; i++) {
        printf("%d --> ", d[i]);
        sum += abs(disk - d[i]);
        disk = d[i];
    }

    printf("\nTotal cylinder movement: %d\n", sum);

    return 0;
}
```

```
Output:
Enter no of location 8
Enter position of head 53
Enter elements of disk queue
98
183
37
122
14
124
65
67
53->37->14->0->65->67->98->122->124->183->
Movement of total cylinders 236.
```