SARASWATI Education Society's

SARASWATI College of Engineering

Learn Live Achieve and Contribute

Kharghar, Navi Mumbai - 410 210.

NAAC Accredited

Department of Computer Engineering

Name: Shaikh Adnan Shaukat Ali

Rno: 55

Subject: Operating System (OS)

Class/Sem: SE/IV

SARASWATI Education Society's
SARASWATI College of Engineering
Learn Live Achieve and Contribute
Kharghar, Navi Mumbai - 410 210.
NAAC Accredited

Department of Computer Engineering

Subject :Operating System **Class/Sem: SE/IV**

INDEX

Expt. No.	Date	Name of the Experiment	Page No
1	8/02/2021	FCFS	4-7
2	8/02/2021	SJF	8-12
3	18/02/2021	Round-Robin	13-17
4	22/03/2021	Producer-Consumer	18-22
5	5/04/2021	Banker's Algorithm	23-26
6	5/04/2021	Dynamic Partitioning	27-35
7	19/04/2021	Page Replacement	36-42
8	19/04/2021	Disk Scheduling	43-52
9	15/05/2021	Fork() System Call	53-61
10	15/05/2021	Linux commands-Case Study	62-71

Arpita Saxena
Subject I/C

SARASWATI Education Society's
SARASWATI College of Engineering
Learn Live Achieve and Contribute
Kharghar, Navi Mumbai - 410 210.
NAAC Accredited

Department of Computer Engineering

Subject :Operating System **Class/Sem:SE/IV**

INDEX

Expt. No.	Date	Name of the Experiment	Page No
1	20/3/2021	Assignment 1	72-92
2	14/5/2021	Assignment 2	93-107

Arpita Saxena
Subject I/C

EXPERIMENT NO. 1

AIM: Write a program to implement FCFS scheduling algorithm for process management.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python Compiler.

THEORY:

FCFS (First Come First Serve):

First Come First Serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

1. Completion Time (CT): Time at which process completes it's execution.
2. Turn Around Time (TAT): Time difference between completion and arrival time. Turn Around Time = Completion Time – Arrival Time.
3. Waiting Time (WT): Time difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time.

Problems with the FCFS scheduling algorithm:

1. It is Non Pre-emptive algorithm, which means the **process priority** doesn't matter.

   If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper scheduling.

2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc) utilization.

CONCLUSION: Hence, we have implemented a program on FCFS scheduling algorithm on process management.

CODE:

```
from prettytable import PrettyTable
def fcfs():
    pid,at,bt,ct,tt,wt = [],[],[],[],[],[]
    print()
    z = int(input("Enter number of Process: "))
    print()
    for i in range(0,z):
        pid.append(int(input("Enter Proccess id:")))
        print()
        at.append(int(input("Enter arrival time:")))
        print()
        bt.append(int(input("Enter burst time:")))
        print()
    for i in range(0,z):
        min = [pid[i],at[i],bt[i]]
        j = i-1
        while(j>=0 and at[j]>min[1]):
            at[j+1],pid[j+1],bt[j+1] = at[j],pid[j],bt[j]
            j = j-1
        pid[j+1],at[j+1],bt[j+1] = min[0],min[1],min[2]
    ct.append(at[0]+bt[0])
    tt.append(ct[0]-at[0])
    wt.append(tt[0]-bt[0])
    for i in range(1,z):
```

```
        ct.append(at[i]+bt[i]) if (at[i] > ct[i-1]) else  ct.append(ct[i-1]+bt[i])
        tt.append(ct[i]-at[i])
        wt.append(tt[i]-bt[i])


    for i in range(0,z):
        min = [pid[i],at[i],bt[i],ct[i],tt[i],wt[i]]
        j = i-1
        while(j>=0 and pid[j]>min[0]):
            at[j+1],pid[j+1],bt[j+1],ct[j+1],tt[j+1],wt[j+1] =
at[j],pid[j],bt[j],ct[j],tt[j],wt[j]
            j = j-1
        pid[j+1],at[j+1],bt[j+1],ct[j+1],tt[j+1],wt[j+1] =
min[0],min[1],min[2],min[3],min[4],min[5]



    x = PrettyTable()
    x.field_names = ["Process id","Arrival Time","Burst Time","Completion
Time","Turnaround Time","Waiting Time"]
    for a,b,c,d,e,f in zip(pid,at,bt,ct,tt,wt):
        x.add_row([a,b,c,d,e,f])
    print(x)
    print("Total turnaround time: "+str(sum(tt))+"\nTotal waiting time:
"+str(sum(wt)))
    print("Average turnaround time: "+str(sum(tt)/z)+"\nAverage waiting time:
"+str(sum(wt)/z))
if __name__ == "__main__":
    print("55_Adnan_Shaikh")
    fcfs()
}
```

OUTPUT:

```
Command Prompt
C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Scheduling algorithm>python fcfs.py
55_Adnan_Shaikh

Enter number of Process: 6

Enter Proccess id:1

Enter arrival time:6

Enter burst time:3

Enter Proccess id:2

Enter arrival time:3

Enter burst time:1

Enter Proccess id:3

Enter arrival time:4

Enter burst time:3

Enter Proccess id:4

Enter arrival time:2

Enter burst time:2

Enter Proccess id:5

Enter arrival time:1

Enter burst time:4

Enter Proccess id:6

Enter arrival time:5

Enter burst time:5
```

```
Command Prompt

+------------+--------------+------------+-----------------+-----------------+--------------+
| Process id | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time |
+------------+--------------+------------+-----------------+-----------------+--------------+
|     1      |      6       |     3      |       19        |       13        |      10      |
|     2      |      3       |     1      |       8         |       5         |      4       |
|     3      |      4       |     3      |       11        |       7         |      4       |
|     4      |      2       |     2      |       7         |       5         |      3       |
|     5      |      1       |     4      |       5         |       4         |      0       |
|     6      |      5       |     5      |       16        |       11        |      6       |
+------------+--------------+------------+-----------------+-----------------+--------------+
Total turnaround time: 45
Total waiting time: 27
Average turnaround time: 7.5
Average waiting time: 4.5
```

EXPERIMENT NO. 2

AIM: Write a program to implement SJF scheduling algorithm for process management.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python compiler

THEORY:

Shortest Job First (SJF):

Shortest Job First (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non pre-emptive algorithm.

- Shortest Job First has the advantage of having minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using concept of aging.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

CONCLUSION: Hence, we have implemented a program on SJF scheduling algorithm on process management.

CODE:

```
from prettytable import PrettyTable


def sjf():

    processes = [[1, 6, 1], [2, 8, 1], [3, 7, 2], [4, 3, 3]]
#[processid,burst_time,arrival_time]
```

```
n = len(processes)
rt = [0] * n
wt = [0] * n
tat = [0] * n


for i in range(n):
    rt[i] = processes[i][1]
complete = 0
t = 0
minm = 999999999
short = 0
check = False


while (complete != n):

    for j in range(n):
        if ((processes[j][2] <= t) and (rt[j] < minm) and rt[j] > 0):
            minm = rt[j]
            short = j
            check = True
    if (check == False):
        t += 1
        continue
```

```
rt[short] -= 1


minm = rt[short]
if (minm == 0):
    minm = 999999999


if (rt[short] == 0):

    complete += 1
    check = False


    fint = t + 1


    wt[short] = (fint - processes[short][1] - processes[short][2])

    if (wt[short] < 0):
        wt[short] = 0

t += 1
for i in range(n):
    tat[i] = processes[i][1] + wt[i]
```

```
    print("Processes    Arrival Time    Burst Time     Waiting",
                "Time     Turn-Around Time")
    total_wt = 0
    total_tat = 0
    for i in range(n):

        total_wt = total_wt + wt[i]
        total_tat = total_tat + tat[i]
        print(" ", processes[i][0], "\t\t",
                processes[i][2], "\t\t",
                processes[i][1], "\t\t",
                wt[i], "\t\t", tat[i])

    print("\nAverage waiting time = %.5f "%(total_wt /n) )
    print("Average turn around time = ", total_tat / n)

if __name__ == "__main__":
    print("55_Adnan_Shaikh")
    sjf()
```

OUTPUT:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Scheduling algorithm>python sjf.py
55_Adnan_Shaikh
Processes    Arrival Time    Burst Time     Waiting Time     Turn-Around Time
  1              1               6               3               9
  2              1               8               16              24
  3              2               7               8               15
  4              3               3               0               3

Average waiting time = 6.75000
Average turn around time =  12.75
```

EXPERIMENT NO. 3

AIM: Write a program to implement Round Robin scheduling algorithm for process management.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python compiler.

THEORY:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.

One of the most commonly used technique in CPU scheduling as a core.

It is pre-emptive as processes are assigned CPU only for a fixed slice of time at most.

The disadvantage of it is more overhead of context switching.

Steps to find waiting times of all processes:

1. Create an array **rem_bt[]** to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array).
2. Create another array **wt[]** to store waiting times of process. Initialize this array as 0.
3. Initialize time: t = 0.
4. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.
    a. If rem_bt[i] > quantum
        i. t = t +quantum
        ii. rem_bt[i] -= quantum
    b. Else
        i. t = t + rem_bt[i]
        ii. wt[i] = t – bt[i]
        iii. rem_bt[i] = 0; // This process is over.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,

The average waiting time will be, 11 ms.

CONCLUSION: Hence, we have implemented a program on Round Robin scheduling algorithm on process management.

CODE:

```
from prettytable import PrettyTable
def rr():
    pid,at,bt,tt,wt = [],[],[],[],[]
    print()
    z = int(input("Enter number of Process: "))
    ct = [0]*z
    quantum = int(input("Enter the quantum time: "))
    print()

    for i in range(0,z):
        pid.append(int(input("Enter Proccess id:")))
        print()
        at.append(int(input("Enter arrival time:")))
```

```
    print()
    bt.append(int(input("Enter burst time:")))
    print()
for i in range(0,z):
    min = [pid[i],at[i],bt[i]]
    j = i-1
    while(j>=0 and at[j]>min[1]):
        at[j+1],pid[j+1],bt[j+1] = at[j],pid[j],bt[j]
        j = j-1
    pid[j+1],at[j+1],bt[j+1] = min[0],min[1],min[2]

rem_bt = bt.copy()
tot = 0
while(True):
    status = True
    for x in range(0,z):
        if rem_bt[x] > 0:
            status = False

            if (rem_bt[x]-quantum)>0:
                rem_bt[x] -= quantum
                tot += quantum
            else:
                tot += rem_bt[x]
                ct[x] = tot+1
                rem_bt[x] = 0
    if status:
```

```
            break
    tt.append(ct[0]-at[0])
    wt.append(tt[0]-bt[0])
    for i in range(1,z):
        tt.append(ct[i]-at[i])
        wt.append(tt[i]-bt[i])
    x = PrettyTable()
    x.field_names = ["Process id","Arrival Time","Burst Time","Completion
Time","Turnaround Time","Waiting Time"]


    for a,b,c,d,e,f in zip(pid,at,bt,ct,tt,wt):
        x.add_row([a,b,c,d,e,f])


    print(x)


    print("Total turnaround time: "+str(sum(tt))+"\nTotal waiting time:
"+str(sum(wt)))
    print("Average turnaround time: "+str(sum(tt)/z)+"\nAverage waiting time:
"+str(sum(wt)/z))


if __name__ == "__main__":
    print("55_Adnan_Shaikh")
    rr()
```

OUTPUT:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Scheduling algorithm>python roundrobin.py
55_Adnan_Shaikh

Enter number of Process: 4
Enter the quantum time: 3

Enter Proccess id:1

Enter arrival time:1

Enter burst time:10

Enter Proccess id:2

Enter arrival time:2

Enter burst time:4

Enter Proccess id:3

Enter arrival time:3

Enter burst time:5

Enter Proccess id:4

Enter arrival time:4

Enter burst time:3

+------------+--------------+------------+-----------------+-----------------+--------------+
| Process id | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time |
+------------+--------------+------------+-----------------+-----------------+--------------+
|     1      |      1       |     10     |       23        |       22        |     12       |
|     2      |      2       |     4      |       17        |       15        |     11       |
|     3      |      3       |     5      |       19        |       16        |     11       |
|     4      |      4       |     3      |       13        |       9         |     6        |
+------------+--------------+------------+-----------------+-----------------+--------------+
Total turnaround time: 62
Total waiting time: 40
Average turnaround time: 15.5
Average waiting time: 10.0
```

EXPERIMENT NO. 4

AIM: Write a program to implement the solution of Producer Consumer problem through Semaphore.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python Compiler.

THEORY:

<u>THE PRODUCER CONSUMER PROBLEM:</u>

- There are one or more producers generating some type of data (records, characters) and placing these in a buffer.
- There is a single consumer that is taking items out of the buffer one at a time.
- The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full, and that the consumer won't try to remove data from an empty buffer.

To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

```
producer:
while (true) {
      /* produce item v */;
      b[in] = v;
      in++;
}
```

```
consumer:
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */;
}
```

CONCLUSION: Hence, we have implemented a program on Producer Consumer problem through semaphore.

CODE:

```
mutex,full,empty,x = 1,0,3,0
def wait(s):
    global mutex,full,empty,x
    s -=1
    return s

def signal(s):
    global mutex,full,empty,x
    s += 1
    return s
```

```
def producer():
    global mutex,full,empty,x
    mutex = wait(mutex)
    full = signal(full)
    empty = wait(empty)
    x +=1
    print("Producer produces the item: "+str(x))
    mutex = signal(mutex)

def consumer():
    global mutex,full,empty,x
    mutex = wait(mutex)
    full = wait(full)
    empty = signal(empty)
    print("Consumer consumes item: "+str(x))
    x -=1
    mutex = signal(mutex)

if __name__ == "__main__":
    print("55_Adnan Shaikh")
    print("1.Producer \n2.Consumer \n3.Exit")

    while(1):
        n = int(input("Enter your choice: "))

        if n == 1:
```

```
    if mutex == 1 and empty != 0:
        producer()
    else:
        print("Buffer is full")
elif n == 2:
    if  mutex == 1 and full != 0:
        consumer()
    else:
        print("Buffer is empty!!")
elif n == 3:
    break
else:
    print("You pressed wrong key please try")
```

OUTPUT:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Semaphore algorithm>python prodcons.py
55_Adnan Shaikh
1.Producer
2.Consumer
3.Exit
Enter your choice: 1
Producer produces the item: 1
Enter your choice: 1
Producer produces the item: 2
Enter your choice: 1
Producer produces the item: 3
Enter your choice: 1
Buffer is full
Enter your choice: 2
Consumer consumes item: 3
Enter your choice: 1
Producer produces the item: 3
Enter your choice: 1
Buffer is full
Enter your choice: 2
Consumer consumes item: 3
Enter your choice: 2
Consumer consumes item: 2
Enter your choice: 2
Consumer consumes item: 1
Enter your choice: 2
Buffer is empty!!
Enter your choice: 2
Buffer is empty!!
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: 1
Enter your choice: 3
```

EXPERIMENT NO. 5

AIM: Write a program to demonstrate Deadlock Avoidance through Banker's Algorithm

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python Compiler.

THEORY:

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.

We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

- Available – A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
- Max. An n × m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj .
- Allocation. An n × m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj .
- Need. An n × m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

This algorithm can be described as follows:

(1) LetWork and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.

(2) Find an index i such that both

- Finish[i] == false
- Needi ≤Work
- If no such i exists, go to step 4.

(3) Work =Work + Allocation i

Finish[i] = true

Go to step 2.

(4) If Finish[i] == true for all i, then the system is in a safe state.

CONCLUSION: Hence, we have demonstrated a program on deadlock avoidance through Banker's Algorithm.

CODE:

```
def banker():
    processes = int(input("number of processes : "))
    resources = int(input("number of resources : "))
    max_resources = [int(i) for i in input("maximum resources : ").split()]
    print("\n-- allocated resources for each process --")
    currently_allocated = [[int(i) for i in input(f"process {j + 1} : ").split()] for j
in range(processes)]
    print("\n-- maximum resources for each process --")
    max_need = [[int(i) for i in input(f"process {j + 1} : ").split()] for j in
range(processes)]


    allocated = [0] * resources
    for i in range(processes):
        for j in range(resources):
```

```
        allocated[j] += currently_allocated[i][j]
print(f"\ntotal allocated resources : {allocated}")
available = [max_resources[i] - allocated[i] for i in range(resources)]
print(f"total available resources : {available}\n")
running = [True] * processes
count = processes
while count != 0:
    safe = False
    for i in range(processes):
        if running[i]:
            executing = True
            for j in range(resources):
                if max_need[i][j] - currently_allocated[i][j] > available[j]:
                    executing = False
                    break
            if executing:
                print(f"process {i + 1} is executing")
                running[i] = False
                count -= 1
                safe = True
                for j in range(resources):
                    available[j] += currently_allocated[i][j]
                break
    if not safe:
        print("the processes are in an unsafe state.")
        break
    print(f"the process is in a safe state.\navailable resources : {available}\n")
```

```
if __name__ == '__main__':
    print("55_AdnanShaikh")
    banker()
```

OUTPUT:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\BankerAlgo>python bankeralgorithm.py
55_AdnanShaikh
number of processes : 5
number of resources : 3
maximum resources : 10 5 7

-- allocated resources for each process --
process 1 : 0 1 0
process 2 : 2 0 0
process 3 : 3 0 2
process 4 : 2 1 1
process 5 : 0 0 2

-- maximum resources for each process --
process 1 : 7 5 3
process 2 : 3 2 2
process 3 : 9 0 2
process 4 : 2 2 2
process 5 : 4 3 3

total allocated resources : [7, 2, 5]
total available resources : [3, 3, 2]

process 2 is executing
the process is in a safe state.
available resources : [5, 3, 2]

process 4 is executing
the process is in a safe state.
available resources : [7, 4, 3]

process 1 is executing
the process is in a safe state.
available resources : [7, 5, 3]

process 3 is executing
the process is in a safe state.
available resources : [10, 5, 5]

process 5 is executing
the process is in a safe state.
available resources : [10, 5, 7]
```

EXPERIMENT NO. 6

AIM: Write a program to implement Dynamic Memory partitioning algorithm i.e. First fit, Best fit, Worst fit for memory management.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python Compiler.

THEORY:

Memory Partitioning:

1. **Fixed Partitioning:**

Main memory is divided into a no. of static partitions at system generation time. A process may be loaded into a partition of equal or greater size. Memory Manager will allocate a region to a process that best fits it unused memory within an allocated partition called internal fragmentation

Advantages:

Simple to implement Little OS overhead

Disadvantages:

It may lead to inefficient use of memory due to internal fragmentation. Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as internal fragmentation.

There can be two possibilities as follows:

1. Equal size partitioning
2. Unequal size Partition

Not suitable for systems in which process memory requirements not known ahead of time; i.e. timesharing systems. When the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3. Here small jobs have to wait to get into memory; even though plenty of memory is free an alternative organization is to maintain a single queue as in

Figure (b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

Fig: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

2. Dynamic/Variable Partitioning:

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. The partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

An example, using 64 Mbytes of main memory, is shown in figure. Eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, the data memory becomes more and more fragmented, and memory utilization declines.

This phenomenon is referred to as external fragmentation, indicating that the memory that is external to all partitions becomes increasingly fragmented. One technique for overcoming external fragmentation is compaction.

From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. There are various dynamic memory partitioning algorithms such as Best Fit, Worst Fit and First Fit for memory management.

1. **<u>Best Fit:</u>**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire

list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage:

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage:

It is slower and may even tend to fill up memory with tiny useless holes.

2. **Worst fit:**

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage:

Reduces the rate of production of small gaps.

Disadvantage:

If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

3. **First Fit:**

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage:

Fastest algorithm because it searches as little as possible.

Disadvantage:

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus, request for larger memory requirement cannot be accomplished.

CONCLUSION: Hence, we have implemented programs on Dynamic Memory partitioning algorithm with First fit, Best fit, Worst fit for memory management.

CODE:

1. Best Fit:

```
import sys
def printf(format, *args):
    sys.stdout.write(format % args)

class BestFit:
    def __init__(self):
        return None

    def bestFit(self,block,process):

        m,n = len(block),len(process)

        allocation = [-1 for x in range(n)]

        for x in range(0,n):
            best_index = -1
            for y in range(0,m):
                if block[y] >= process[x]:
                    if best_index == -1:
                        best_index = y
                    else:
                        if block[y]<block[best_index]:
                            best_index = y

            if best_index != -1:
```

```
            allocation[x] = best_index
            block[best_index] -=process[x]

        printf("\nProcess no. \t\t Process size \t\t Block No. \n")

        for x in range(0,n):
            if allocation[x] == -1:
                printf("%10d \t\t %10d \t\t Not Allocated\n",(x+1),process[x])
            else:
                printf("%5d \t\t %15d \t\t %8d\n",(x+1),process[x],(1+allocation[x]))
        return None



if __name__=="__main__":
    print("55_Adnan_Shaikh")
    block = [100,500,200,300,600]
    process = [212,417,112,426]
    bestf = BestFit()
    bestf.bestFit(block,process)
```

2. Worst Fit:

```
import sys
def printf(format, *args):
    sys.stdout.write(format % args)

class WorstFit:
```

```
def __init__(self):
    return None

def worstFit(self,block,process):

    m,n = len(block),len(process)

    allocation = [-1 for x in range(n)]

    for x in range(0,n):
        worst_index = -1
        for y in range(0,m):
            if block[y] >= process[x]:
                if worst_index == -1:
                    worst_index = y
                else:
                    if block[y]>block[worst_index]:
                        worst_index = y

        if worst_index != -1:
            allocation[x] = worst_index
            block[worst_index] -=process[x]

    printf("\nProcess no. \t\t Process size \t\t Block No.\n")

    for x in range(0,n):
        if allocation[x] == -1:
```

```
            printf("%5d \t\t %15d \t\t Not Allocated\n",(x+1),process[x])
        else:
            printf("%5d \t\t %15d \t\t %5d\n",(x+1),process[x],(1+allocation[x]))
    return None



if __name__=="__main__":
    print("55_Adnan_Shaikh")
    block = [100,500,200,300]
    process = [500,400,600,426]
    worstf = WorstFit()
    worstf.worstFit(block,process)
```

3. First Fit:

```
import sys
def printf(format, *args):
    sys.stdout.write(format % args)

class FirstFit:
    def __init__(self):
        return None

    def firstFit(self,block,process):

        m,n = len(block),len(process)
```

```
    allocation = [-1 for x in range(n)]

    for x in range(0,n):
        for y in range(0,m):
            if block[y] >= process[x]:
                allocation[x] = y
                block[y] -=process[x]
                break



    printf("\nProcess no. \t\t Process size \t\t Block No.\n")

    for x in range(0,n):
        if allocation[x] == -1:
            printf("%5d \t\t %15d \t\t Not Allocated\n",(x+1),process[x])
        else:
            printf("%5d \t\t %15d \t\t %5d\n",(x+1),process[x],(1+allocation[x]))
    return None



if __name__=="__main__":
    print("55_Adnan_Shaikh")
    block = [100,500,200,300,600]
    process = [500,90,112,426]
    firstf = FirstFit()
```

```
firstf.firstFit(block,process)
```

OUTPUT:

1. Best Fit:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Fitting algorithm>python bestfit.py
55_Adnan_Shaikh

Process no.             Process size            Block No.
    1                       212                     4
    2                       417                     2
    3                       112                     3
    4                       426                     5
```

2. Worst Fit:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Fitting algorithm>python worstfit.py
55_Adnan_Shaikh

Process no.             Process size            Block No.
    1                       500                     2
    2                       400                 Not Allocated
    3                       600                 Not Allocated
    4                       426                 Not Allocated
```

3. First Fit:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Fitting algorithm>python firstfit.py
55_Adnan_Shaikh

Process no.             Process size            Block No.
    1                       500                     2
    2                        90                     1
    3                       112                     3
    4                       426                     5
```

EXPERIMENT NO. 7

AIM: Write a program to demonstrate Page Replacement policies i.e. FIFO, LRU for handling page faults.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python Compiler.

THEORY:

1.FIFO Algorithm:

The simplest page-replacement algorithm is a FIFO algorithm. The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little book-keeping on the part of the operating system. The idea is obvious from the name – the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences Bélády's anomaly.

FIFO page replacement algorithm is used by the VAX/VMS operating system, with some modifications. Partial second chance is provided by skipping a limited number of entries with valid translation table references and additionally, pages are displaced from process working set to a system wide pool from which they can be recovered if not already re-used.

Treats page frames allocated to a process as a circular buffer:

When the buffer is full, the oldest page is replaced. Hence first-in, first-out:

A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO.

Simple to implement, requires only a pointer that circles through the page frames of the process.

Consider the Example:

2.LRU Algorithm:

The least recently used page (LRU) replacement algorithm, though similar in name to NRU, differs in the fact that LRU keeps track of page usage over a short period, while NRU just looks at the usage in the last clock interval. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory (almost as good as Adaptive Replacement Cache), it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.

The most expensive method is the linked list method, which uses a linked list containing all the pages in memory. At the back of this list is the least recently used page, and at the front is the most recently used page. The cost of this implementation lies in the fact that items in the list must be moved about every memory reference, which is a very time-consuming process.

Because of implementation costs, one may consider algorithms (like those that follow) that are like LRU, but which offer cheaper implementations.

One important advantage of the LRU algorithm is that it is amenable to full statistical analysis. It has been proven, for example, that LRU can never result in more than N-times more page faults than OPT algorithm, where N is proportional to the number of pages in the managed pool.

On the other hand, LRU's weakness is that its performance tends to degenerate under many quite common reference patterns. For example, if there are N pages in the LRU pool, an application executing a loop over array of N + 1 pages will cause a page fault on each access. As loops over large arrays are common, much effort has been put into modifying LRU to work better in such situations. Many of the proposed LRU modifications try to detect looping reference patterns and to switch into suitable replacement algorithm, like Most Recently Used (MRU).

LRU Page Replacement Algorithm:

- Replaces the page that has not been referenced for the longest time:

  By the principle of locality, this should be the page least likely to be referenced soon performs nearly as well as the optimal policy.

CONCLUSION: Hence, we have implemented a program on FIFO and LRU page replacement policies for handling page faults.

CODE:

1. FIFO:

```
class FIFO:
    def __init__(self,p,fs):
        self.p = p
        self.fs = fs
        self.n = len(self.p)

    def search(self,key,f):
        for i in range(self.fs):
            if f[i] == key:
                return True
        return False

    def fifo(self):
        f = [-1 for i in range(self.fs)]
        hit = 0
        k = 0
```

```
        for i in range(self.n):
            if(self.search(self.p[i],f)):
                hit += 1
                print(str(self.p[i])+"\t\t  No")
                continue
            if k<self.fs:
                f[k] = self.p[i]
                k += 1

            if k == self.fs:
                k =0
            print(str(self.p[i])+"\t"+str(f[0])+" "+str(f[1])+" "+str(f[2])+"\t  Yes")

        print("Number of hits = "+str(hit))
        print("Number of miss = "+str(self.n-hit))

if __name__ == '__main__':
    print("55_Adnan Shaikh")
    x = input("Enter elements of reference string use comma: ")
    p = [int(i.replace(" ","")) for i in x.split(',')]
    fs= int(input("Enter the frame size: "))

    fif = FIFO(p,fs)
    print("PAGE\tFRAME\tPAGE FAULT")
    fif.fifo()
```

2. LRU:

```
def lru():
    print("Enter the number of frames: ",end="")
    capacity = int(input())
    f,st,fault,pf = [],[],0,'No'
    print("Enter the reference string: ",end="")
    s = list(map(int,input().strip().split()))
    print("\nString|Frame →\t",end="")
    for i in range(capacity):
        print(i,end=' ')
    print("Fault\n   ↓\n")
    for i in s:
        if i not in f:
            if len(f)<capacity:
                f.append(i)
                st.append(len(f)-1)
            else:
                ind = st.pop(0)
                f[ind] = i
                st.append(ind)
            pf = 'Yes'
            fault += 1
        else:
            st.append(st.pop(st.index(f.index(i))))
            pf = 'No'
        print("   %d\t\t"%i,end="")
        for x in f:
```

```
        print(x,end=' ')
    for x in range(capacity-len(f)):
        print(' ',end=' ')
    print(" %s"%pf)
  print("\nTotal Requests: %d\nTotal Page Hit: %d\nTotal Page Faults: %d\nFault Rate: %0.2f%%"%(len(s),len(s)-fault,fault,(fault/len(s))*100))


if __name__ == "__main__":
  print("55_Adnan Shaikh")
  lru()
```

OUTPUT:

1. FIFO:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS>cd "Page Replacement"

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Page Replacement>python fifo.py
55_Adnan Shaikh
Enter elements of reference string use comma: 6,0,5,2,0,3,0,4,2,3,0,3,2,5,2,0,5,6,0,5
Enter the frame size: 3
PAGE    FRAME   PAGE FAULT
6       6 -1 -1   Yes
0       6 0 -1    Yes
5       6 0 5     Yes
2       2 0 5     Yes
0                 No
3       2 3 5     Yes
0       2 3 0     Yes
4       4 3 0     Yes
2       4 2 0     Yes
3       4 2 3     Yes
0       0 2 3     Yes
3                 No
2                 No
5       0 5 3     Yes
2       0 5 2     Yes
0                 No
5                 No
6       6 5 2     Yes
0       6 0 2     Yes
5       6 0 5     Yes
Number of hits = 5
Number of miss = 15
```

2. LRU:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Page Replacement>python lru.py
55_Adnan Shaikh
Enter the number of frames: 3
Enter the reference string: 6 0 5 2 0 3 0 4 2 3 0 3 2 5 2 0 5 6 0 5

String|Frame →  0 1 2 Fault
   ↓

   6            6      Yes
   0            6 0    Yes
   5            6 0 5  Yes
   2            2 0 5  Yes
   0            2 0 5  No
   3            2 0 3  Yes
   0            2 0 3  No
   4            4 0 3  Yes
   2            4 0 2  Yes
   3            4 3 2  Yes
   0            0 3 2  Yes
   3            0 3 2  No
   2            0 3 2  No
   5            5 3 2  Yes
   2            5 3 2  No
   0            5 0 2  Yes
   5            5 0 2  No
   6            5 0 6  Yes
   0            5 0 6  No
   5            5 0 6  No

Total Requests: 20
Total Page Hit: 8
Total Page Faults: 12
Fault Rate: 60.00%
```

EXPERIMENT NO. 8

AIM: Write a program to demonstrate Disk Scheduling Algorithms i.e FCFS, SCAN, C-SCAN.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Python Compiler.

THEORY:

DISK SCHEDULING:

One of the responsibilities of the operating system is the use the hardware efficiently. For the disk drivers, meeting this responsibility entails having fast access time and large disk bandwidth.

For magnetic disks, the access time has two major components: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

The disk **bandwidth** is the total number of the bytes transferred, divided by the total time between the first request for the service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced.

1.FCFS Scheduling:

The simplest form of disk scheduling is, of course, the first-come, first-serve (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

Consider, for example, a disk queue with requests for I/O to blocks on cylinders :- 98, 183, 37, 122, 14, 124, 65, 67 in the order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for total head movements of 640 cylinders.

2.SCAN:

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves towards 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 arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Let's return to out example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position.

Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183. If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just

behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

3.C-SCAN:

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.

When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

CONCLUSION: Hence, we have implemented the programs of Disk Scheduling Algorithms using FCFS, SCAN, C-SCAN.

CODE:

1. FCFS:

```
def fcfs():
    thm = 0
    req = int(input("Enter the Number of Requests : "))
    hp = int(input("Enter the Initial Head Position : "))
    srate = int(input("Enter the Seek Rate : "))
    print("Enter the Requests : ")
    arr = [ int(input()) for i in range(req)]
    thm = thm + abs(hp - arr[0])
    print(str(hp)+" -> "+str(arr[0]), end=")
```

```
    for i in range(1, req):
        thm = thm + abs(arr[i] - arr[i-1])
        print(" -> "+str(arr[i]), end=")
    stime = srate * thm
    print("\nThe Total Head Movement is",thm)
    print("The Seek Time is",stime)

if __name__ == "__main__":
    print("55_Adnan Shaikh")
    fcfs()
```

2. SCAN:

```
def scan():
    thm = 0
    req = int(input("Enter the Number of Requests : "))
    hp = int(input("Enter the Initial Head Position : "))
    pos = hp
    srate = int(input("Enter the Seek Rate : "))
    print("Enter the Requests : ")
    arr = [ int(input()) for i in range(req)]
    start = 0
    end = int(input("Enter ending position of disk: "))
    print(hp, end=")
    if(hp<100):
        for i in range(pos, start-1, -1):
            if i in arr:
                thm+= abs(pos-i)
```

```
            pos = i
            print(" -> ",i, end=")
            arr.remove(i)
    thm+= abs(pos-start)
    pos = start
    print(" -> ", start, end=")
    for i in range(pos, end+1):
        if i in arr:
            thm+= abs(pos-i)
            pos = i
            print(" -> ", i, end=")
            arr.remove(i)
else:
    for i in range(pos, end+1):
        if i in arr:
            thm+= abs(pos-i)
            pos = i
            print(" -> ",i, end=")
            arr.remove(i)
    thm+= abs(pos-end)
    pos = end
    print(" -> ", end, end=")
    for i in range(pos, start-1,-1):
        if i in arr:
            thm+= abs(pos-i)
            pos = i
            print(" -> ", i, end=")
```

```
            arr.remove(i)
    stime = thm * srate
    print("\nThe Total Head Movement is",thm)
    print("The Seek Time is",stime)

if __name__ == "__main__":
    print("55_Adnan Shaikh")
    scan()
```

3. C-SCAN:

```
def CSCAN(arr, head,disk_size,size):
        seek_count = 0
        distance = 0
        cur_track = 0
        left = []
        right = []
        seek_sequence = []
        left.append(0)
        right.append(disk_size - 1)
        for i in range(size):
                if (arr[i] < head):
                        left.append(arr[i])
                if (arr[i] > head):
                        right.append(arr[i])
        left.sort()
        right.sort()
```

```
        for i in range(len(right)):
                cur_track = right[i]
                seek_sequence.append(cur_track)
                distance = abs(cur_track - head)
                seek_count += distance
                head = cur_track
        head = 0
        seek_count += (disk_size - 1)

        for i in range(len(left)):
                cur_track = left[i]
                seek_sequence.append(cur_track)
                distance = abs(cur_track - head)
                seek_count += distance
                head = cur_track
        print("Total number of seek operations =",
                seek_count)
        print("Seek Sequence is")
        print(*seek_sequence, sep="\n")

if __name__ == "__main__":
    print("55_Adnan Shaikh")
    arr = list(map(int,input("Enter requests: ").strip().split()))
    head = int(input("Enter head position: "))
    disk_size = int(input("Enter ending position of disk: ")) + 1
    print("Initial position of head:", head)
    CSCAN(arr, head, disk_size,len(arr))
```

OUTPUT:

1. FCFS:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Disk Scheduling algorithm>python fcfs.py
55_Adnan Shaikh
Enter the Number of Requests : 8
Enter the Initial Head Position : 143
Enter the Seek Rate : 1
Enter the Requests :
80
1470
913
1777
948
1022
1750
130
143 -> 80 -> 1470 -> 913 -> 1777 -> 948 -> 1022 -> 1750 -> 130
The Total Head Movement is 6125
The Seek Time is 6125
```

2. SCAN:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Disk Scheduling algorithm>python scan.py
55_Adnan Shaikh
Enter the Number of Requests : 8
Enter the Initial Head Position : 143
Enter the Seek Rate : 1
Enter the Requests :
80
1470
913
1777
948
1022
1750
130
Enter ending position of disk: 4999
143 ->  913 ->  948 ->  1022 ->  1470 ->  1750 ->  1777 ->  4999 ->  130 ->  80
The Total Head Movement is 9775
The Seek Time is 9775
```

3. C-SCAN:

```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\OS\Disk Scheduling algorithm>python cscan.py
55_Adnan Shaikh
Enter requests: 80 1470 913 1777 948 1022 1750 130
Enter head position: 143
Enter ending position of disk: 4999
Initial position of head: 143
Total number of seek operations = 9985
Seek Sequence is
913
948
1022
1470
1750
1777
4999
0
80
130
```

EXPERIMENT NO. 9

AIM: To create child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection.

S/W Requirements: Linux Batch, Python compiler(already included in Linux).

THEORY:

System call **fork ()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork ()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork ()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork ()**:

- If **fork ()** returns a negative value, the creation of a child process was unsuccessful.
- **fork ()** returns a zero to the newly created child process.
- **fork ()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork ()**, a simple test can tell which process is the child. Please note that Unix will make an exact copy of the parent's address

space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

Suppose the above program executes up to the point of the call to **fork ()** (marked in red color):

If the call to **fork ()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:

Both processes start their execution right after the system call **fork ()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork ()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others.

In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork ()** calls will not be affected even though they have identical variable names.

If you run this program, you might see the following on the screen:

```
.................
This line is from pid 3456, value 13
This line is from pid 3456, value 14
.................
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
.................
This line is from pid 3456, value 21
This line is from pid 3456, value 22
.................
```

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

Consider one more simple example, which distinguishes the parent from the child.

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


#define   MAX_COUNT  200
```

```
void  ChildProcess(void);              /* child process prototype  */
void  ParentProcess(void);             /* parent process prototype */

void  main(void)
{
    pid_t  pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("   This line is from child, value = %d\n", i);
    printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
```

```
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable **i**. For simplicity, **printf()** is used.

When the main program executes **fork ()**, an identical copy of its address space, including the program and all data, is created. System call **fork ()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.

Now both programs (*i.e.*, the parent and child) will execute independent of each other starting at the next statement:

In the parent, since **pid** is non-zero, it calls function **ParentProcess()**. On the other hand, the child has a zero **pid** and calls **ChildProcess()** as shown below:

Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of **MAX_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

CONCLUSION: Hence, we have successfully created child process and obtain the process id in Linux Operating system using fork() and getpid() system calls.

Code:

```
import os
import time

def child():
  print('\nA new child ',  os.getpid())
  time.sleep(0.5)
  os._exit(0)

def parent():
  while True:
    newpid = os.fork()
    if newpid == 0:
      child()
    else:
      pids = (os.getpid(), newpid)
      print("\nparent: %d, child: %d\n" % pids)
      time.sleep(0.5)
    reply = input("\nq for quit / c for new fork \n")
    if reply.lower() == 'c':
       continue
    else:
       break
```

```
if __name__ == "__main__":
    print("55_Adnan Shaikh")
    parent()
```

Output:

```
adnan@adnan-VirtualBox: ~/Desktop
adnan@adnan-VirtualBox:~/Desktop$ python3 systemcalls.py
55_AdnanShaikh

parent: 3248, child: 3249


A new child:  3249

press Q for quit and C for new fork
c

parent: 3248, child: 3250


A new child:  3250

press Q for quit and C for new fork
C

parent: 3248, child: 3251


A new child:  3251

press Q for quit and C for new fork
Q
adnan@adnan-VirtualBox:~/Desktop$
```

EXPERIMENT NO. 10

AIM: Case Study of Linux Commands

RESOURCES REQUIRED:

H/W Requirements: P-IV and above, Ram 128 MB, Printer, Internet Connection**.**

S/W Requirements: Linux Operating System.

THEORY:

Commands	Typical Use
ls (list)	listing all the folders and files in current or specified directory.
pwd (print working directory)	Printing current working directory.
cd (change directory)	Changing current directory.
mkdir (make directory)	To create new directory (folder).
nano	To open existing file or creating and opening new file.
cat (Concatenate)	Commonly use to print content of file there are lots of different uses too.
rm (remove)	Deleting files.
rmdir (remove directory)	Deleting directory (folder).
echo	To insert or append into text into files.
sudo	To get superuser access.
apt	Use for upgrading, installing, listing, searching, etc apps
cp (copy)	Copying files from current directory to specify directory.
grep	To check the string content in specified files.
free	To get the information of all the memory.
lshw (list hardware)	To get information of system hardware.
history	History of all the commands used.

ping (packet internet groper)	This command is used to check the network connectivity between host and server/host.
hostname	Return hostname
ps (process state)	To get the Process state
mv (move)	To move files from one directory to another or renaming the files.
clear	To clear the terminal.

Conclusion: Hence, we have successfully study and used essential Linux commands.

Output :

Basic use of pwd, ls, cd and mkdir:

nano command:

Before using **nano**, Testing(directory) is empty :

```
adnan@adnan-VirtualBox:~/Desktop/Testing$ ls
adnan@adnan-VirtualBox:~/Desktop/Testing$ nano test.py
```

Afte using **nano** to create test.py file:

Created newfile

```
Activities   Terminal                                   May 21  16:04
                                   adnan@adnan-VirtualBox: ~/Desktop/Testing
  GNU nano 5.2                                                  test.py
#!/usr/bin/python3

print("Hello world!!!")
```

ls command:

Using **ls** to check if the file is created

```
adnan@adnan-VirtualBox:~/Desktop/Testing$ ls
test.py
```

rm command:

Using **rm** to remove test.py and checking if it removed successfully using **ls**:

```
adnan@adnan-VirtualBox:~/Desktop/Testing$ rm test.py
adnan@adnan-VirtualBox:~/Desktop/Testing$ ls
adnan@adnan-VirtualBox:~/Desktop/Testing$
```

echo and cat command:

Creating file mdhash256texting.txt using **nano** and appending string into the using **echo** and checking the content of file after appending using **cat:**

```
adnan@adnan-VirtualBox:~/Desktop/Testing$ nano mdhash256testing.txt
adnan@adnan-VirtualBox:~/Desktop/Testing$ echo "I, Giorno Giovanna, have a dream" >> mdhash256testing.txt
adnan@adnan-VirtualBox:~/Desktop/Testing$ cat mdhash256testing.txt

I, Giorno Giovanna, have a dream
adnan@adnan-VirtualBox:~/Desktop/Testing$
```

More functionality of cat:

Creating file1.txt and file2.txt and writing text through terminal into them using **cat** command and merging them in file3.txt using same:

```
Activities   Terminal   May 21 16:33
adnan@adnan-VirtualBox: ~/Desktop
adnan@adnan-VirtualBox:~$ cd Desktop
adnan@adnan-VirtualBox:~/Desktop$ mkdir temporary
adnan@adnan-VirtualBox:~/Desktop$ ls temporary
adnan@adnan-VirtualBox:~/Desktop$ cat > file1.txt
Hello, This is file1 it will be merge to file2 to create new file3 #1
adnan@adnan-VirtualBox:~/Desktop$ cat > file2.txt
Hello, this is file2 #1
adnan@adnan-VirtualBox:~/Desktop$ cat file1.txt file2.txt > file3.txt
adnan@adnan-VirtualBox:~/Desktop$ cat file3.txt
Hello, This is file1 it will be merge to file2 to create new file3 #1
Hello, this is file2 #1
adnan@adnan-VirtualBox:~/Desktop$
```

Getting CPU and OS information using **cat:**

```
Activities   Terminal   May 21 16:53
adnan@adnan-VirtualBox: ~/Desktop/temporary
adnan@adnan-VirtualBox:~/Desktop/temporary$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 165
model name      : Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
stepping        : 2
microcode       : 0xffffffff
cpu MHz         : 2591.990
cache size      : 12288 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_
tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni ssse3 pcid sse4_1 sse4_2 hypervisor lahf_lm invpcid_single ibrs_enhanced fsgsbase in
vpcid md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs itlb_multihit
bogomips        : 5183.98
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 165
model name      : Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
```

```
Activities   Terminal   May 22 11:53
adnan@adnan-VirtualBox: ~/Desktop
adnan@adnan-VirtualBox:~/Desktop$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.10 (Groovy Gorilla)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.10"
VERSION_ID="20.10"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=groovy
UBUNTU_CODENAME=groovy
adnan@adnan-VirtualBox:~/Desktop$
```

More functionality of ls:

ls –l will give directories and files name with their details (The file type, The file permissions, Number of hard links to the file, File owner, File group, File size, Date and Time, File name) and ls –al use to list hidden files and folders.

ls –p to check content inside directory, **ls –r** to list files and folders in reverse alphabetical order, **ls –s** to get the size of each file and folder with total size. We can combine above command as **ls –psr** or **ls –p –s –r** (sequence can be change) :

```
adnan@adnan-VirtualBox:~$ ls -l
total 36
drwxr-xr-x 4 adnan adnan 4096 May 21 16:02 Desktop
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Documents
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Downloads
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Music
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Pictures
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Public
drwxr-xr-x 4 adnan adnan 4096 May 20 15:31 snap
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Templates
drwxr-xr-x 2 adnan adnan 4096 May 19 19:08 Videos
adnan@adnan-VirtualBox:~$ ls -r -p
Videos/  Templates/  snap/  Public/  Pictures/  Music/  Downloads/  Documents/  Desktop/
adnan@adnan-VirtualBox:~$ ls -r -p -s
total 36
4 Videos/  4 Templates/  4 snap/  4 Public/  4 Pictures/  4 Music/  4 Downloads/  4 Documents/  4 Desktop/
adnan@adnan-VirtualBox:~$
```

cp command:

Copying file sample.txt from current directory to Desktop using **cp** and checking content using **cat** after copying:

```
Activities   Terminal▾                May 22 11:12
adnan@adnan-VirtualBox: ~/Desktop
adnan@adnan-VirtualBox:~/Desktop/temporary$ cp sample.txt ~/Desktop
adnan@adnan-VirtualBox:~/Desktop/temporary$ cd ..
adnan@adnan-VirtualBox:~/Desktop$ ls
python  sample.txt  temporary  Testing
adnan@adnan-VirtualBox:~/Desktop$ cat sample.txt

What is Lorem Ipsum?

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a g
alley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It
was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versi
ons of Lorem Ipsum.
Why do we use it?
```

mv command:

Moving file1.txt, file2.txt and file3.txt from Desktop to temporary folder using **mv**:

```
Activities  Terminal  May 21 16:35
adnan@adnan-VirtualBox: ~/Desktop/temporary
adnan@adnan-VirtualBox:~/Desktop$ ls
file1.txt  file2.txt  file3.txt  python  temporary  Testing
adnan@adnan-VirtualBox:~/Desktop$ mv file1.txt file2.txt file3.txt "temporary"
adnan@adnan-VirtualBox:~/Desktop$ ls
python  temporary  Testing
adnan@adnan-VirtualBox:~/Desktop$ cd temporary
adnan@adnan-VirtualBox:~/Desktop/temporary$ ls
file1.txt  file2.txt  file3.txt
adnan@adnan-VirtualBox:~/Desktop/temporary$
```

Renaming file3.txt to merged.txt using **mv**:

```
Activities  Terminal  May 21 16:37
adnan@adnan-VirtualBox: ~/De
adnan@adnan-VirtualBox:~/Desktop/temporary$ mv file3.txt merged.txt
adnan@adnan-VirtualBox:~/Desktop/temporary$ ls
file1.txt  file2.txt  merged.txt
adnan@adnan-VirtualBox:~/Desktop/temporary$
```

history command:

```
Activities  Terminal  May 21 16:50
adnan@adnan-VirtualBox: ~/Desktop/temporary
adnan@adnan-VirtualBox:~/Desktop/temporary$ history
    1  man sudo_root
    2  clear
    3  ls
    4  exit
    5  ls
    6  cd desktop
    7  cd Desktop
    8  ls
    9  mkdir python
   10  nano systemcalls.py
   11  python systemcalls.py
   12  python3 systemcalls.py
   13  clear
   14  python3 systemcalls.py
   15  ls
   16  cd ..
   17  ls
   18  cd ..
   19  l
   20  ls
   21  cd bin
   22  ls
   23  /usr/lib/nux/unity_support_test -p
   24  cd ..
   25  cd /usr/lib/nux/unity_support_test
   26  /usr/lib/nux/unity_support_test -p
   27  cd ..
   28  /usr/lib/nux/unity_support_test -p
   29  /usr/lib
   30  /usr
   31  cd /user
   32  ls
   33  /lib/nux/unity_support_test -p
   34  /lib/nux/unity_support_test -p
```

grep command:

grep –n “string” sample.txt return appearance of words with line no

grep –I used for removing case sensitivity

grep –e use for defining string expression multiple expression can be use using **–e** option

Output of combine of above options with **grep** command is given below:

String match is highlighted in red color

```
Activities   Terminal   May 22 11:09
adnan@adnan-VirtualBox: ~/Desktop/temporary
adnan@adnan-VirtualBox:~/Desktop/temporary$ ls
file1.txt  file2.txt  merged.txt  sample.txt
adnan@adnan-VirtualBox:~/Desktop/temporary$ grep -n -i -e "lorem" -e "ipsum" sample.txt
2:What is Lorem Ipsum?
4:Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a
 galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. I
t was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including ver
sions of Lorem Ipsum.
7:It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-le
ss normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use
Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by ac
cident, sometimes on purpose (injected humour and the like).
11:Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock
, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the wor
d in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by C
icero, written in 45 BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from
a line in section 1.10.32.
13:The standard chunk of Lorem Ipsum used since the 1500s is reproduced below for those interested. Sections 1.10.32 and 1.10.33 from "de Finibus Bonorum et Malorum" by Cicero are also
reproduced in their exact original form, accompanied by English versions from the 1914 translation by H. Rackham.
16:There are many variations of passages of Lorem Ipsum available, but the majority have suffered alteration in some form, by injected humour, or randomised words which don't look even
slightly believable. If you are going to use a passage of Lorem Ipsum, you need to be sure there isn't anything embarrassing hidden in the middle of text. All the Lorem Ipsum generators
 on the Internet tend to repeat predefined chunks as necessary, making this the first true generator on the Internet. It uses a dictionary of over 200 Latin words, combined with a handf
ul of model sentence structures, to generate Lorem Ipsum which looks reasonable. The generated Lorem Ipsum is therefore always free from repetition, injected humour, or non-characterist
ic words etc.
adnan@adnan-VirtualBox:~/Desktop/temporary$
```

ps command:

getting process id, parent id their path, memory usage and CPU usage and sorting them in descending order using **ps** with options **–e –o** :

```
Activities   Terminal   May 22 11:55
adnan@adnan-VirtualBox: ~/Desktop
adnan@adnan-VirtualBox:~/Desktop$ ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head
    PID    PPID CMD                         %MEM %CPU
   1065     818 /usr/bin/gnome-shell         8.1  2.4
   1321     818 /snap/snap-store/518/usr/bi  6.6  0.2
   1226    1045 /usr/libexec/evolution-data  2.0  0.0
    836     834 /usr/lib/xorg/Xorg vt2 -dis  2.0  2.6
   2120     818 /usr/libexec/gnome-terminal  1.6  0.5
    897     818 /usr/libexec/goa-daemon      1.1  0.0
   1143     818 /usr/libexec/evolution-cale  1.0  0.0
   1831    1045 update-notifier              0.9  0.0
   1165     818 /usr/libexec/evolution-addr  0.9  0.0
adnan@adnan-VirtualBox:~/Desktop$
```

free and lshw command:

free use return information about memory state **--human** option will format output in human readable format.

lshw list hardware information **–short** option is used to get information in brief.

hostname command:

return hostname of system:

sudo and apt command:

sudo use for super user, apt list return applications present in a system

--upgradeable option use to check if any application needed an update, apt install will install specified application to system from host. (snap can be use instead of apt to install application). :

ping command:

Assignment 1

Roll no: 5511

Sub: O.S

Class/Sem: S.E/IV

1)) Explain the evolution of operating system.

Ans)) 1)) The developments that have occured in computing facilities in the last 4-5 decades. In 1960's people were using mainframe computer system as a computing facilities.

2)) The mainframe computer system would be normally kept in a computer center with a controlled environment.

3)) The user uses to prepare their program as a desk of punched card with encoded list of program instructions. Among the deck, header cards which were "Job control cards" would specify computer needed to compile the program.

4) The entire processing was batched processing on the basis of set of jobs. There was no user interactions in processing the jobs. The processor would remain busy at a time in processing one program.

5)) If any I/O required then the processor would wait as processor speed high with compare to I/O speed. As a consequence of this processor utilization was poor and would remain for idle for most time. In this scenarios only one program was kept in the memory at the time of execution.

6)) Int the decades of 1970's the operating system designed by designers developed the concept of multiprogramming. In multiprogramming, physical memory was decided in many partitions. Each partition was holding exactly one program and operating system was residing in exactly one partition.

7)) Along with the policy of memory allocation, a policy was needed to switch the processor from one program to other. It has solved the problem of processor remaining idle while I/o or program is going on.

8)) In the early decades of 1980's system designers added a features to give interactive access to the system. In this period time sharing system came into picture.

9)) The time sharing idea is to let use fill that all system resources is given his program while execution. To achieve it each user program was allocated a slice of time of processors. In this allocations time, user program partly complete the execution.

10)) when time slice ends the processor switches to other program one by one and again comback to first program in very short time. This give the illusion to user that, the system is only available to him alone.

11)) In the decade of 1970-80 there was an exceptional growth in bulk storage. It has help to realize the concept to propose the idea of extended storage. Again the concept of "virtual storage" helped to utilize this extended storage as an enchanced address space.

12)) By swapping in active part of the program in memory from the disk and swapping out suspended program to the disk supported to enhance the concept of multiprogramming. This led to the large number of users using the system.

13)) In the mid of 1970 decade, M.I.T developed first time sharing system called compatible time sharing system (CTSS).

14)) In the decade of 1980's microcomputers came into scene and CP/M was first operating system for this platform. It was developed on Intel 8080 in 1974. MP/M the version of CP/M was designed as multiuser, timesharing with real time capabilities.

15)) After the arrival of IBM P.C based Intel 8086, P.C-DOS of IBM and M.S DOS was developed. After the arrival of IBM 286, IBM and Microsoft developed OS/2 for 286 and 386 Machines. It was multiuser system.

16)) After the arrival of 386 & 486 based computers, to facilitate and reduce the development time, Microsoft developed Graphical User Interface (G.U.I) based M.S WINDOWS. It was

used on top of D.O.S to provide user friendly G.U.I.

17)) After arrival of pentium processors, Microsoft developed multiuser Windows N.T and Single user Windows 95 then to windows 98 and windows XP. After the Windows 2000 was developed as a single user operating System for desktop users and as a multiusers system for business users, to combine the both streams.

18)) Now Linux is emerged as most successful operating system which is variant of UNIX below table summarize the evolution of operating system.

S.No.	Decade	Machines	Operating System & features.
1.	1970	Main frames	MULTICS, Multiuser, time Sharing
2.	1980	Mini computers	UNIX, Multi-user, time Sharing & Networking.
3.	1990	Desktop computers	DOS, MS-Windows, UNIX, LINUX etc - Multiuser, time Sharing, networked, Clustered.
4	2000	Hand held Computers.	Windows XP, Windows 2000, Linux - Multiprocessors, Multiuser and Networked.

2) Write a Short note on Linux Kernel.

Ans)) 1)) The design of Linux looks like any other traditional, non-monoKernel UNIX implementation. It is a multiuser, multitasking system consisting of UNIX compatible tools.

2)) The file system of Linux sticks onto traditional UNIX semantics. Linux also completely implements standard UNIX networking model.

3)) The internal details of Linux's design have been influenced heavily by the history of this O.S development. Initially Linux was developed solely on P.C architecture from the beginning of the design Linux tries to acquire as much functionality as possible from limited resources.

4)) UNIX can run on multiprocessor system having main memory in several megabytes with diskspace of several G.B. Apart from this it can run on system with 4 M.B of R.A.M.

5)) With the enhancement in machine configuration & efficiency Linux Kernel's grow to implement more UNIX functionalities. Apart from speed and efficiency, the current work and design passed on standardization.

6)) Due to variety of UNIX implementation at present available, the problem faced are source code written for one may not essentially compile or run correctly or another.

7)) Linux is designed to be agreeable with the relation POSIX Documents atleast two linux distribution have detained official POSIX Certification.

Layer in Linux System

a) System Hardware :- It consists of cpu, memory, disks, terminal & other devices.

b) Linux O.S :- Its function is to contribute hardware, and make available a system called interface to all the programs. These systems calls permit user programs to create, delete and manage processes, files & others resources.

c) Standard library :- Program issues a trap instruction to switch from user mode to Kernel mode and makes a system call by putting arguments in stack or register. Library contains procedures implemented in assembly language for these system calls.

d)) Standard Utilities:-

These programs are invoked by users while working on terminal. It includes the Shell, compilers, editors, text processing programs and file manipulation utilities.

e)) Users:- All the users interacting with system. There are three interfaces, that is true system call interface, the library interface, and the, interface formed by the set of standard utility programs.

Q.)) Explain the layered operating system structure.

Ans)) ① With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS & UNIX systems.

② A System can be made modular in many ways. One method is the layered approach, in which the O.S is broken into a number of layers (level). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

← A layered operating system.

③ The main advantage of the layered approach is simplicity of construction & debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging & system verification.

④ The first layer can be debugged without any concern for the rest of the system, because by definition, it uses only the basic hardware (layer 0) to implement its functions.

⑤ once the first layer is debugged, its correct functioning can be assumed while the second layer debugged

and so on.

(6) If error is found during the debugging of a particular layer, the error must be on that layer, because the lower layers below it are already debugged.

(7) The lower layer is abstracted from upper layer. upper layer will only know what operations & services can lower layer provide but doesn't know how it is providing.

(8) The major difficulty with layered approach involves appropiate defining the various layers. Because a layer can use only lower level layers, careful planning is necessary

(9) A final problem with layered implentations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to I/O layer, which calls the memory management layer, which in turns calls the CPU-scheduling layer, which is then passed to hardware.

(10) At each layer, the parameters maybe modified, data may need to be passed, & so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a nonlayered system.

4.)) What is the role of process control block?

Ans)) 1)) Any process is identified by its process control block (P.C.B). P.C.B is a data structure used by the operating system to keep track on the process.

2)) All the information associated with process is kept in P.C.B. There is seperate P.C.B for each process.

3)) Traffic control module keep track on the status of the processes PC.B's of the processes which are in the same state (ready, executing, waiting, etc) are linked together giving a specific name to the list such as ready list.

4)) If many processes wait for the same device, then PCB's of all these processes are linked together in chain waiting for that device.

5)) If the device become free than traffic controller checks the P.C.B chain to see if any process is waiting for the device.

6)) If the processes are available in the device chain then again it is placed back to ready state to request that device again.

P.C.B chain waiting for processor in ready queue.

7)) Process State:- The state maybe new, ready running, waiting, halted and so on.

8)) Program Counter:- The counter indicates the address of the next instruction to be executed for this process.

9)) C.P.U registers:- The register vary in numbers and type, depending on the computer architecture. They include accumulators, index registers, Stack Pointers, & general purpose registers, plus any condition code information.

10)) CPU-scheduling information:- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

11)) Memory-management information:- This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the O.S.

12)) Accounting information:- This information includes the amount of C.P.U and real time used, time limits, account numbers, job or process numbers, and so on.

13)) I/O Status information:- This information includes the list of I/O devices allocated to the processes, a list of open files, and so on.

Page :
Date : / /

Q.5) Explain the concept of multi-threading.

Ans) <<1>> Multi threading is a concept in which more than one thread are executed in a system, where the execution of these threads can be of two different types such as concurrent and parallel multithreaded executions.

2>> A thread can be defined as a chunk or unit of a process that can be identified as either an user level thread or kernel level thread.

3>> It is usually used for its essential charateristics like it uses the system resources efficiently, high performance, greatly responsive & also its parallel executionability.

4>> Multithreading models are of 3-types:-

<<a>> Many-to-Many:-

Any number of user threads can interact with equal or lesser number of kernel threads.

many-to-many model

Page :
Date : / /

b) Many-to-one :-

It maps many user level threads to one kernel level thread.

Many-to-one model

c) One-to-one :-

One user level thread is mapped to only one kernel level thread.
It supports multiple threads to execute in parallel on processors. This model provides more concurrency than the many to one model.

One-to-one model

Page :
Date : / /

Q6) What is the difference between Pre-Emptive and Non-Pre-Emptive Scheduling.

Ans)

Pre-Emptive scheduling	Non-Pre-Emptive scheduling
1)) In pre-emptive scheduling the C.P.U is allocated to the processes for limited time.	In non pre-emptive scheduling the C.P.U is allocated to the process till it terminates or switches to waiting state.
2)) The executing process in pre-emptive scheduling is interrupted in the middle of execution.	The executing process in non-pre-emptive scheduling is not interrupted in the middle of execution.
3)) Pre-emptive scheduling has the overhead of switching the process from ready state to running state, vicevexse and maintaining the ready queue.	Non-pre emptive scheduling has no overhead of the switching the process from running state to ready state.
4)) If a process with high priority frequently arrives in the ready queue then the process with low priority have to wait for a long & it may have to starve	In non pre-emptive scheduling if C.P.U is allocated to the process with larger burst time then the processes with small burst time may have to starve.
5) Pre-emptive scheduling is quite flexible because the critical processes are allowed to access C.P.U as they arrive into the ready queue, no matter what process is executing currently.	Non pre-emptive scheduling is rigid as even if a critical ~~section~~ process enters the ready queue, the process running on C.P.U is not disturbed.

Page:
Date: / /

Q.7) How do Processess communicate with each other? Explain.

Ans) ① Inter Process communication (I.P.C) refers to a mechanism, where the O.S allows various Process to communicate with each other. This involves synchronizing their actions and managing shared data.

② Communication can be of two types:-

a) Between related processes initiating from only one Process such as parent & child processes.

b) Between unrelated processes or two or more different processess.

③ Pipes:- Communicating between two unrelated Processes the mechanism is half duplex meaning the first Process communicates with 2nd process. To achieve a full duplex pipe is required.

④ FIFO:- Communication between 2 unrelated Processes FIFO is a full duplex, meaning the 1st Process can communicate with 2nd Process & viceverse at the same time.

⑤ Message Queues:- Comunication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message & retrieving it out of the queue. Once, retrieved the message is no longer available in the queue.

⑥ Shared Memory:- Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all processes.

Page :
Date : / /

7)) Semaphores:- Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory it needs to be locked & released when the access is removed. This needs to be repeated by all the processes to secure data.

8)) Signals:- Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal and the destination process will handle it accordingly.

Q.8)) What are the methods of deadlock recovery.

Ans)) When a deadlock detection algorithm determines that a deadlock has occured in a system, the system must recover from that deadlock. There are two approaches of breaking a deadlock.

(A) Process Termination:-
To eliminate the deadlock we can simply kill one or more processes. For this we use two methods:-

(I) Abort all the deadlocked processes:-
Aborting all the processes will certainly break the deadlock but with a great expenses. The deadlock processes maybe computed for a long time & the result of these partial computations must be discarded & there is a probability to recalculate them later.

Page:
Date: / /

II) Abort one process at a time until deadlock is eliminated:-

Abort one deadlocked process at a time, until deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead because after aborting each process we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

B) Resource Pre-emption:-
To eliminate deadlock using resource preemption, we pre-empt some resources from processes & give these resources to other processes. This method will raise three issues:-

I) Selecting a Victim:-
We must determine which resources ~~and~~ and which process are to be pre-empted and also the order to minimize the cost.

II) Roll-back:-

We must determine what should be done with the process from which resources are pre-empted. One simple idea is total roll back. That means abort the process & restart it.

Page:
Date: / /

III Starvation:-

In a system it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called Starvation & must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

Q.9) What is the Producer-Consumer Problem?

Ans)) 1) The Producer consumer problem is a classic problem this is used for multi-process synchronization i.e; Sychronization between more than one processes.

2) In the Producer-Consumer problem, there is once producer that is ~~providing~~ producing something and there is one consumer that is consuming the products produced by the producer.

3)) The producer and consumer share the same memory buffer that is of fixed size.

4)) The job of producer is to generate data, put it into the buffer & again ~~st~~ start generating data. While the job of consumer is to consume the data from buffer.

5)) Problems that occur in producer-consumer problem are:-

- The producer should produce data only when the buffer is not full. ~~The~~ If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.

- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer should n't be allowed to take any data from the buffer.
- The producer & consumer shouldn't access the buffer at the same time.

6) The above problems can be solved with the help of semaphores. In the producer-consumer problems, we use three semaphore variables:-

(I) Semaphore S:-
This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either producer or consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.

(II) Semaphore E:-
This semaphore variable is used to define the empty space in the buffer. Initially it is set to the whole space of the buffer i.e 'n' because the buffer is initially empty.

(III) Semaphore F:-
This semaphore variable is used to define the space that is filled by the producer. Initially it is set to '0' because there is no space filled by the producer initially.

Q.10) What is the dining Philosphers Problem?

Ans)) The dining Philosphers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

Problem Statement:-

Consider there are five philosphers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in below figure:-

At any instant, a philosopher is either eating or thinking when a philosopher wants to eat he uses two chopsticks- one from their left & one from right. When a philosopher wants to think he keeps down both chopsticks at their original place.

Solution:-

It is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking & eating.

Page :
Date : / /

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and pick up that chopstick. Then he waits for the right chopstick to be available & then pick up when it is available. After eating he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solution for this are:-

(1) A philosopher must be allowed to pickup the chopsticks only if both the left & right chopsticks are available.

(2) Allow only four philosphers to sit at the table. That way, if all the four philosphers pick up four chopsticks, there will be ~~on~~ one chopstick left on the table. So, one philospher can start eating & eventually two chopsticks will be available. In this way deadlocks can be avoided.

Date : / /

Assignment no. 2

1.) Explain memory management with fixed and variable partitions.

Ans) ① The earliest and one of the simplest technique which can be used to load more than one process into main memory is fixed partitioning or contiguous memory location.

② In this technique, the main memory is divided into partitions of equal or different sizes.

③ The operating system always resides in the first partition while other partitions can be used to store user processes.

④ The memory is assigned to the processes in contigious way.

⑤ In fixed partioning,

1. The partitions cannot overlap.
2. A process must be contiguously present in a partition for the execution.

⑥ There are various cons of using this technique:

① Internal fragmentation :- If size of the process is lesser than the total size of the partition then some size of the partition get wasted and remain unused. This is wastage of the memory & called internal fragmentation.

Date: / /

2.) External fragmentation:- The total unused space of various partitions cannot be used to load the processes even though there is space available but not in contiguous form.

3.) Limitation on the size of the process:- If the process size is larger than the size of maximum partition then that process cannot be loaded into the memory. Therefore, we have to imposed restriction on process size.

4.) Degree of multiprogramming decreases because of internal fragmentation.

Page:
Date: / /

(7) Dynamic or variable partitioning.

(1) Dynamic Partitioning tries to overcome the problems caused by fixed Partitioning. In this technique, the partition size is not declared initially. It is declared at the time of process loading.
(2) The first partition is reserved for the operating system. The remaining space is divided into parts.
(3) The partition size will be equal to size of the process. The size of each partition
(4) The partition size varies according to the need of the process so that internal fragmentation can be avoided.

(8) Advantages of Dynamic Partitioning:-

(i) No internal fragmentation.
(ii) No limitation on the size of the process
(iii) High degree of multiprogramming relative to fixed partitioning.

(6) Only disadvantages of dynamic partitionig are external fragmentation and complexity of memory allocation.

There are different algorithms used for Dynamic Partitioning :-

① Best fit :- Find the minimum space where process can be fit.
② Worst fit :- find the maximum space where process can be fit.
③ First fit :- The first suitable space is allocated to process.

Q.2) Calculate number of page fault and page hit for the page replacement policies FIFO optimal and LRU for a given reference string.

Ans) Reference String = 6, 0, 5, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 5, 2, 0, 5, 6, 0, 5

from size = 3

① FIFO

6	0	5	2	0	3	0	4	2	3	0	3	2	5	2	0	5	6	0	5
		5	5	5	5	0	0	0	3	3	3	3	3	2	2	2	2	2	5
	0	0	0	0	3	3	3	2	2	2	2	2	5	5	5	5	5	0	0
6	6	6	2	2	2	2	4	4	4	0	0	0	0	0	0	0	6	6	6
F	F	F	F		F	F	F	F	F	F			F	F			F	F	F

Page Fault = 15
Page Hit = 5.

Page :
Date : / /

2) Optimal :-

6	0	5	2	0	3	0	4	2	3	0	3	2	5	2	0	5	6	0	5
		5	5	5	3	3	3	3	3	3	3	3	5	5	5	5	5	5	5
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
6	6	6	2	2	2	2	2	2	2	2	2	2	2	2	2	2	6	6	6
F	F	F	F		F		F			F			F				F		

Page fault = 9
Page hit = 11

③ Least Recently used :-

6	0	5	2	0	3	0	4	2	3	0	3	2	5	2	0	5	6	0	5
		5	5	5	3	3	3	2	2	2	2	2	2	2	2	2	6	6	6
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
6	6	6	2	2	2	2	4	4	4	0	0	0	5	5	5	5	5	5	5
F	F	F	F		F		F	F	F	F			F		F		F		

Page fault = 12
Page Hit = 8

Page:
Date: / /

Q.3) Explain different types of file organization.

Ans) ① We use the term file organization to refer to the logical structuring of the records as determined by the way in which they are accessed. The physical organization of the file on the secondary storage depends on the blocking strategy and the file allocation strategy.

② In choosing a file, several criteria are important:
- Short access time.
- Ease of update
- Economy of storage
- Simple maintenance
- Reliability.

③ The five organizations are as follows:-
(i) The pile
(ii) The sequential file.
(iii) The indexed sequential file.
(iv) The indexed file.
(v) The direct, or hashed, file.

④ The pile order:- ① The least complicated form of file organization may be termed the pile. Data are collected in order in which they arrive. Each record consists of one burst of data.
② The purpose of the pile is simply to accumulate the mass of data & save it. Records may have different fields, or similar fields in different orders. Thus, each field should be self-describing, including a field name as well as value. The length of each field must be implicitly indicated.

by delimiters, explicitly included as a subfield, or known as default for that field type.

(3) A pile file doesn't have any structure, so exhaustive search is used to access any record.

(4) The search inspects each record in the file ~~unit~~ until the desired record is located or the complete file has been searched.

(5) The complete file must be searched to locate all records of particular fied & value.

(5) The Sequential file:-

(1) It is most commonly used structure and records have fixed format.

(2) Each and every record in the file has same length with equal number of fixed-length fields in particular order.

(3) The first field in each record is considered as key field & it is use to identify field.

(4) As a result, the value of key in each record are always different. The records are stored in the logical order of key.

(5) The sequential file can be organized as a linked list. Each physical block contains one or more records.

(6) Each block on disk hold a pointer to next block.

(7) If a new recorded is inserted, a pointer needs to be manipulated only & new record would not copy a particular physical block position.

Page:
Date: / /

⑥ The indexed file:-

1) The sequential file and the indexed sequential file can search the record based on a single field of the file. A part from using key field they cannot search record using other attribute of record.

2) To overcome this difficulty and accomplish flexibility, a structure contain multiple indexes are for each type of field is required.

3) In indexed file, the records are accessed only through their indices.

4) As a result, there is now no limit on the placing of record providing a pointer in a minimum one index refers to that record. In addition, variable-length records can be used.

5) In this structure, two kind of indices are used. An exhaustive index holds are entry for each record in main file.

6) In order to search easily, the index itself is organized as a sequential file.

7) A partial index holds entries to records when the field of concern exists.

⑦ The indexed-sequential file:-

1) It eliminate drawback of sequential file. Similar to sequential file, records are organized in logical order of a key field.

2) It contains index to the file to support random access and an overflow file. The index offer searching ability to reach ~~the~~ the required record quickly.

Page :
Date : / /

3) The overflow file is used in order that a record in the overflow is searched by following pointer from its ancestor.

4) In a single level of indexing, the index is a simple sequential file. In the index file, each record contains two field. A key field identical to key field in the main file, and a pointer into main file.

5) To search a particular field, the index is searched to locate the largest key value that is equal to or preceedes the key value to be searched. The search comes on in the main file at the location denoted by the pointer.

6) The indexed sequential file really decreases the time needed to access a single record, without giving up the sequential nature of file.

8) <u>The Direct or hash file:-</u>

1) This type of file make use of the ability found on disks to access straight any block of known address. There is no idea of sequential ordering like sequential or indexed sequential file.

2) A hashing operation is performed on the key value using hashing function.

3) Direct files are frequently used where very fast access is necessary, where there is use of fixed length records & when records are always accessed one by one.

Page:
Date: / /

Q4) Explain DMA transfer.

Ans) ① The DMA unit is capable of mimicking the processor and indeed, of taking over control of the system bus just like a processor.

② It needs to do this to transfer data to and from memory over the system bus.

③ ~~It~~ DMA means CPU grants I/O module authority to read from or write to memory without involvement.

④ DMA needs a special hardware called DMA controller (DMAC) that manages the data transfer and arbitrates access to the system bus.

⑤ The controller are programmed with source and destination pointer, counters to track, the number of transfer bytes, and setting which includes I/O & memory types, interrupts and states for the CPU cycles.

⑥ DMA techniques works as follows:-

- When a processor wishes to read or write a block of data, it issues a command to the DMA module by sending the following information:-

① Whether a read or write is requested, using the read or write control line between the processor & the DMA module.

② The data of the I/O devices involved, communicated on data line.

③ The starting location in memory to read from or write to, communicate on the data lines and stored by the DMA module in its address register.

Page:
Date: / /

4) The number of words to be read or write, again communicated via the data line and stored in the data count register.
- The processor then continues with other work. It has deletegated this I/O operation to the DMA.
- When DMA transfer is completed, it send interrupt signal to processor. Thus, processor is only involved at the beggginning and at the end.

Page :
Date : / /

5.) Suppose that the head of a moving disk with 200 tracks, numbered 0 to 199, is currently serving a request at 143 and has just finished a request track 125. The queue of requests is kept in the order:
86, 147, 91, 177, 94, 150, 102, 175, 130.
What is the total number of head movements needed to satisfy these requests for the following disk scheduling algorithms: (i) FCFS (ii) SSTF (iii) Elevator.

(i) FCFS :- Head → 143

Total track movements made by r/w head:-

$= (143-86) + (147-86) + (147-91) + (177-91) + (177-94) + (150-94) + (150-102) + (175-102) + (175-130)$

$= 565.$

Page :
Date : / /

(ii) SSTF :- Head → 143

Request order :- 147, 189, 130, 102, 94, 91, 86, 175, 177

⇒ Total movements = (147 − 143) + (150 − 147) + (150 − 130) + (130 − 102) + (102 − 94) + (94 − 91) + (91 − 86) + (175 − 86) + (177 − 175)

= 162.

(iii) Elevator :- Head → 143

request order :- 147, 150, 175, 177, 130, 102, 94, 91, 86

Total movement = 172 + (199 - 177) + (199 - 130)

Page:
Date: / /

Total movements $= (199 - 143) + (199 - 130) + (130 - 86)$
$= 69$