## EXPERIMENT NO – 1

**Aim:-** To study different operating system and Linux commands.

**Lab Objective**:- To study Windows and Linux operating system and learn Linux commands

## Theory:-

Basics of Linux:

What is a command shell?

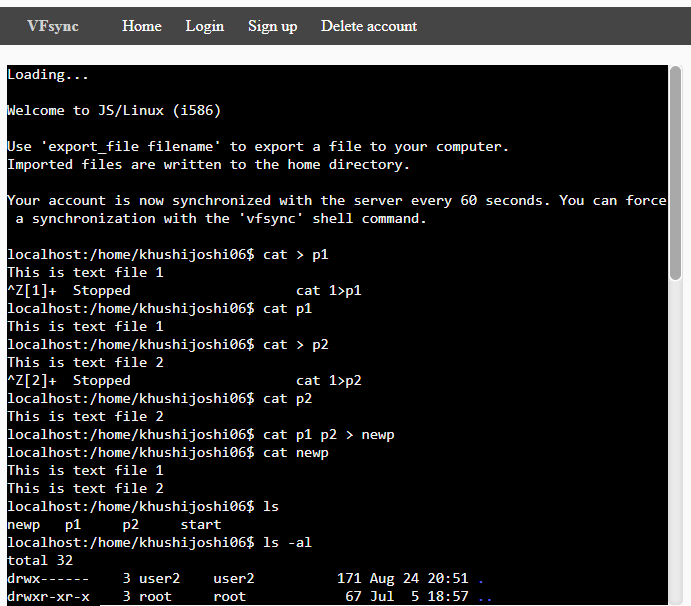
A program that interprets commands allows a user to execute commands by typing them manually at a terminal, or automatically in programs called shell scripts. A shell is not an operating system. It is a way to interface with the operating system and commands.

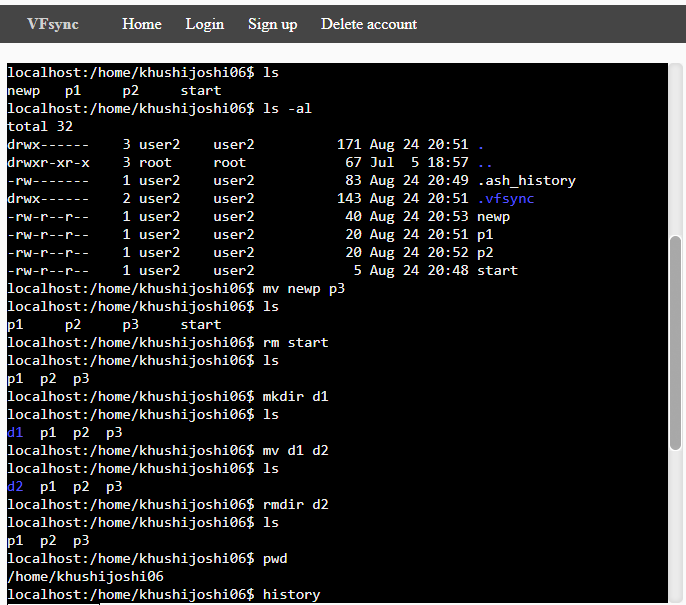
Linux is a Unix-Like operating system. All the Linux/Unix commands are run in the terminal provided by the Linux system. This terminal is just like the command prompt, **https://vfsync.org/vm.html** of Windows OS. Linux/Unix commands are *case-sensitive.* The terminal can be used to accomplish all Administrative tasks. This includes package installation, file manipulation, and user management. Linux terminal is user-interactive. The terminal outputs the results of commands which are specified by the user itself. Execution of typed command is done only after you press the Enter key.

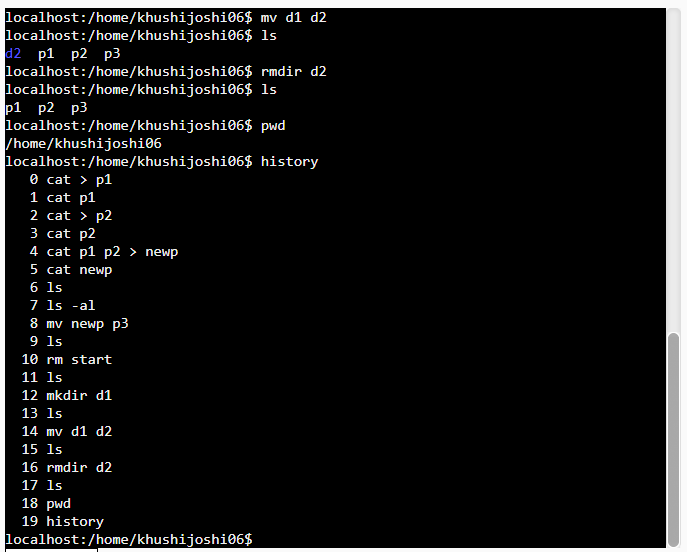
|  |  |  |
| --- | --- | --- |
| **Command** | **Syntax** | **Description** |
| *mkdir* | mkdir | This will create new directory (pascal) in the present directory |
| *rmdir* | rmdir | Removes the directory and all files in that directory |
| *Ls* | $ ls filename | To display the list of files in the dir |
| *Mv* | Mv file1.txt file2.txt | Moves the specified file to another directory |
| *chdir cd* | Chdir filename | Used to change the directory |
| *cat* | Cat file1.txt file2.txt | Appends the content of file.a to the end file.b |
| *Cp* | cp file.a file.b | This will create a duplicate of file.a under a new file name, file.b |
| *clear* | clear | This will clear your screen |
| *kill* | Kill process\_number | Kills the process specified by the Process ID Number |
| *more* | More | Paginates the specified file so it can be read line by line (using Enter key) or  | screen by screen |
| *less* | less | Less displays information a screen at a time, you can also page back and forth |
| *head* | head filename | Display the first 10 lines of the file |
| *tail* | tail filename | Display the last 10 lines of the file |
| *man* | Man topic\_number | Prints the manual page on the specific topic |
| *pwd* | pwd | Prints the current working directory |
| *uname* | uname | This will print to the screen the Linux Kernel in use on your system |

|  |  |  |
| --- | --- | --- |
| *cmp* | cmp file.a file.b | Compares 2 files of any type. |
| *cut* | cut |  |
| *join* | join |  |
| *paste* | paste |  |
| *echo* | echo text | To print the required text |
| *free* | free | Provides a snapshot of the system memory usage |
| *banner* | $ banner text |  |
| *who* | who | Lists currently logged on users username, port, and when  they logged in |
| *date* | Date | Used to display the Date |
| *time* | time | Used to display the Time |
| *mail* | Mail | To send & receive the e-Mails |
| *cal* | Cal month\_name  year | To display the calendar |
| *chmod* | chmod  filename=rwx file | This command gives Read - Write - Execute permission  to everyone |

## Linux commands output:-







**Conclusion:-**

Difference between LINUX operating system and windows operating system was understood. Basic LINUX terminal commands were understood and executed simultaneously.

**Lab Outcome**:- Student should be able to understand Windows and Linux operating system along with how to run Linux command on Terminal (Command Line Interface - CLI).

**EXPERIMENT NO – 2**

**Aim:-** To implement First Come First Served [FCFS] CPU scheduling algorithm.

**Lab objective**: - Describe Process & Process Management using CPU Scheduling Algorithm.

**Theory:-**

**First Come First Serve (FCFS)** is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue.

The average time is quite long under FCFS algorithm. The CPU time is allotted to processes as they arrive. The process arriving first will use the CPU and will execute fully. Meanwhile, if any process comes then it will be added in waiting queue. As soon as the execution of first process ends, the CPU is allotted to the process waiting in queue.

**Result:- Code:**

# Python3 program for implementation of FCFS scheduling # Function to find the waiting time for all processes

def findWaitingTime(processes, n,bt, wt): # waiting time for first process is 0

wt[0] = 0

# calculating waiting time for i in range(1, n ):

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

#Function to calculate turn around time

def findTurnAroundTime(processes, n,bt, wt, tat):

# calculating turnaround time by adding bt[i] + wt[i] for i in range(n):

tat[i] = bt[i] + wt[i]

# Function to calculate average time def findavgTime( processes, n, bt): wt = [0] \* n

tat = [0] \* n total\_wt = 0 total\_tat =

# Function to find waiting time of all processes findWaitingTime(processes, n, bt, wt)

# Function to find turn around time for all processes findTurnAroundTime(processes, n,bt, wt, tat)

# Display processes along with all details

print( "Processes Burst time " +" Waiting time " +" Turn around time") # Calculate total waiting time and total turn around time

for i in range(n):

total\_wt = total\_wt + wt[i] total\_tat = total\_tat + tat[i] print(" " + str(i + 1) + "\t\t" + str(bt[i]) + "\t " +

str(wt[i]) + "\t\t " +

str(tat[i]))

print( "Average waiting time = "+ str(total\_wt / n))

print("Average turn around time = "+ str(total\_tat / n))

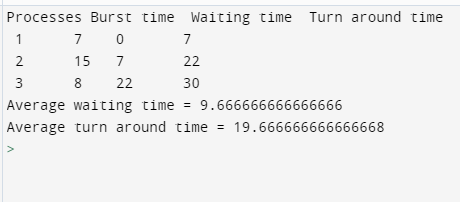
# Driver code

if name ==" main ": # process id's

processes = [ 1, 2, 3] n = len(processes)

# Burst time of all processes burst\_time = [7, 15, 8] findavgTime(processes, n, burst\_time)

**Output:**



**Discussion:**

1. Understood the CPU Scheduling and the FCFS (First come, First served) algorithm with the working of code. Given n processes with their burst times, the task was to find average waiting time and average turn around time using FCFS scheduling algorithm.
2. We understood that it is the simplest scheduling algorithm wherein it simply queues processes in the order that they arrive in the ready queue.
3. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

**Lab Outcome:-**

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The successful implementation of FCFS algorithm helps to understand that Jobs are always executed on a first-come, first-serve basis.

**Conclusion:-**

* + FCFS is a Non-Preemptive CPU scheduling algorithm, so after the process has been allocated to the CPU, it will never release the CPU until it finishes executing.
  + The Average Waiting Time is high.
  + Short processes that are at the back of the queue have to wait for the long process at the front to finish.

## EXPERIMENT NO-3

**Aim: -** To implement Shortest Job First - SJF process scheduling algorithm

**Lab objective:** Describe Process & process management using CPU scheduling Algorithm

## Theory:-

In this algorithm the process with the shorted burst time is allocated to CPU first. If the first process arrives then the CPU is allocated to it. Meanwhile if the 2nd process arrives, the burst time of this process is compared with the remaining burst time of the 1st process. If the burst time of just arrived process is less than that existing process the execution of current process is stopped and the execution of 2nd process begins. On the reverse cane, the execution of current process is continued and the 2nd process goes to waiting queue. This algorithm is appropriate for batch jobs for which run times are known in advanced. SJF scheduling algorithm gives minimum average waiting time for given set of process. It requires precise knowledge of how long a job or process will run.

## Code:-

# Python3 program to implement Shortest Remaining Time First # Shortest Remaining Time First (SRTF)

# Preemptive Shortest Job First

# Function to find the waiting time # for all processes

def findWaitingTime(processes, n, wt): rt = [0] \* n

# Copy the burst time into rt[] for i in range(n):

rt[i] = processes[i][1]

complete = 0

t = 0

minm = 999999999

short = 0 check = False

# Process until all processes gets # completed

while (complete != n):

# Find process with minimum remaining # time among the processes that

# arrives till the current time` 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

# Reduce remaining time by one rt[short] -= 1

# Update minimum minm = rt[short]

if (minm == 0):

minm = 999999999

# If a process gets completely # executed

if (rt[short] == 0):

# Increment complete complete += 1

check = False

# Find finish time of current # process

fint = t + 1

# Calculate waiting time wt[short] = (fint - proc[short][1] -

proc[short][2])

if (wt[short] < 0):

wt[short] = 0

# Increment time t += 1

# Function to calculate turn around time

def findTurnAroundTime(processes, n, wt, tat):

# Calculating turnaround time for i in range(n):

tat[i] = processes[i][1] + wt[i]

# Function to calculate average waiting # and turn-around times.

def findavgTime(processes, n): wt = [0] \* n

tat = [0] \* n

# Function to find waiting time # of all processes

findWaitingTime(processes, n, wt)

# Function to find turn around time # for all processes

findTurnAroundTime(processes, n, wt, tat)

# Display processes along with all details

print("Processes 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][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)

# Driver code

if name ==" main ":

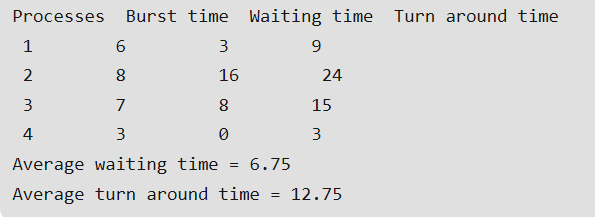
# Process id's

proc = [[1, 6, 1], [2, 8, 1],

[3, 7, 2], [4, 3, 3]]

n = 4 findavgTime(proc, n)

## Output:



**Discussion:** Shortest Job First is a CPU Scheduling Algorithm which focuses on the burst time of the processes. The process having the shortest burst time is executed first. In the above example there are four processes namely P1,P2,P3 and P4. Their Burst Time is 6,8,7,3 respectively. The code finds the shortest execution time of each process at the current time and that process is given priority while the other process is in the ready queue until the first process completes its execution. Likewise the burst time is checked at every point and all the processes are completed. This how the Preemptive Shortest Job First Algorithm works.

## Lab Outcome:-

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The successful implementation of SJF algorithm helps to understand that the process with the shorted burst time is allocated to CPU first**.**

## Conclusion:-

* + - Shortest job first is a scheduling algorithm in which the process with the shortest execution time should be selected for execution next.
    - This is the best approach to minimize waiting time.

# EXPERIMENT NO-4

**Aim: -** To implement Round Robin - RR process scheduling algorithm

**Lab objective:** Describe Process & process management using CPU scheduling Algorithm

**Theory:-**

The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turn. This is the preemptive version of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a cyclic way. A certain time slice is defined in the system which is called time quantum. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will terminate else the process will go back to the ready queue and waits for the next turn to complete the execution. All the jobs get a fare allocation of CPU. The higher the time quantum, the higher the response time in the system. The lower the time quantum, the higher the context switching overhead in the system

**Result:-**

**Code:**

def findWaitingTime(processes, n, bt, wt, quantum): rem\_bt = [0] \* n

for i in range(n):

rem\_bt[i] = bt[i] t = 0 # Current time while(1):

done = True

for i in range(n):

if (rem\_bt[i] > 0) :

done = False

if (rem\_bt[i] > quantum) : t += quantum

rem\_bt[i] -= quantum

else:

if (done == True): break

t = t + rem\_bt[i] wt[i] = t - bt[i] rem\_bt[i] = 0

# Function to calculate turn around time

def findTurnAroundTime(processes, n, bt, wt, tat):

# Calculating turnaround time for i in range(n):

tat[i] = bt[i] + wt[i]

# Function to calculate average waiting

# and turn-around times.

def findavgTime(processes, n, bt, quantum): wt = [0] \* n

tat = [0] \* n

# Function to find waiting time # of all processes

findWaitingTime(processes, n, bt, wt, quantum)

# Function to find turn around time # for all processes

findTurnAroundTime(processes, n, bt, wt, tat) # Display processes along with all details print("Processes 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(" ", i + 1, "\t\t", bt[i],

"\t\t", wt[i], "\t\t", tat[i])

print("\nAverage waiting time = %.5f "%(total\_wt /n) ) print("Average turn around time = %.5f "% (total\_tat / n))

# Driver code

if name ==" main ":

# Process id's proc = [1, 2, 3]

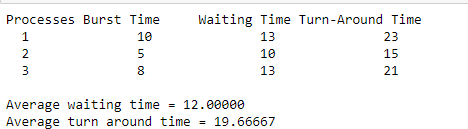
n = 3

# Burst time of all processes burst\_time = [10, 5, 8]

# Time quantum quantum = 2;

findavgTime(proc, n, burst\_time, quantum)

**Output:**



**Discussion:**

**Lab Outcome:-**

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The successful implementation of RR scheduling algorithm helps to understand that every process gets executed in a cyclic way.

**Conclusion:-**

* It can be actually implementable in the system because it is not depending on the burst time.
* A certain time slice is defined in the system which is called time quantum.
* if the execution of the process is completed during that time then the process will terminate else the process will go back to the ready queue

**EXPERIMENT-5**

**Aim: -** To study implement Peterson’s Algorithm

**Learning Objective:** Understand the concept of Peterson’s Algorithm

**Theory: -**

The producer consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producer produce an item and put it into buffer. If buffer is already full then producer will have to wait for an empty block in buffer. Consumer consume an item from buffer. If buffer is already empty then consumer will have to wait for an item in buffer. Implement Peterson’s Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.

**Peterson’s algorithm –**

|  |
| --- |
| // code for producer (j)    // producer j is ready  // to produce an item  flag[j] = true;    // but consumer (i) can consume an item  turn = i;    // if consumer is ready to consume an item  // and if its consumer's turn  while (flag[i] == true && turn == i)        { // then producer will wait }        // otherwise producer will produce      // an item and put it into buffer (critical Section)        // Now, producer is out of critical section      flag[j] = false;      // end of code for producer        //--------------------------------------------------------      // code for consumer i        // consumer i is ready      // to consume an item      flag[i] = true;        // but producer (j) can produce an item      turn = j;        // if producer is ready to produce an item      // and if its producer's turn      while (flag[j] == true && turn == j)            { // then consumer will wait }            // otherwise consumer will consume          // an item from buffer (critical Section)            // Now, consumer is out of critical section          flag[i] = false;  // end of code for consumer |

**Explanation of Peterson’s algorithm –**

Peterson’s Algorithm is used to synchronize two processes. It uses two variables, a bool array **flag** of size 2 and an int variable **turn** to accomplish it.  
In the solution i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute it’s critical section, it sets it’s flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it’s own critical section.  
After this the current process enters it’s critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it’s own flag to false, indication it does not wish to execute anymore.

The program runs for a fixed amount of time before exiting. This time can be changed by changing value of the macro RT.

|  |
| --- |
| // C program to implement Peterson’s Algorithm  // for producer-consumer problem.  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdbool.h>  #define \_BSD\_SOURCE  #include <sys/time.h>  #include <stdio.h>    #define BSIZE 8 // Buffer size  #define PWT 2 // Producer wait time limit  #define CWT 10 // Consumer wait time limit  #define RT 10 // Program run-time in seconds    int shmid1, shmid2, shmid3, shmid4;  key\_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;  bool\* SHM1;  int\* SHM2;  int\* SHM3; |
| int myrand(int n) // Returns a random number between 1 and n  {      time\_t t;      srand((unsigned)time(&t));      return (rand() % n + 1);  }    int main()  {      shmid1 = shmget(k1, sizeof(bool) \* 2, IPC\_CREAT | 0660); // flag      shmid2 = shmget(k2, sizeof(int) \* 1, IPC\_CREAT | 0660); // turn      shmid3 = shmget(k3, sizeof(int) \* BSIZE, IPC\_CREAT | 0660); // buffer      shmid4 = shmget(k4, sizeof(int) \* 1, IPC\_CREAT | 0660); // time stamp        if (shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {          perror("Main shmget error: ");          exit(1);      }      SHM3 = (int\*)shmat(shmid3, NULL, 0);      int ix = 0;      while (ix < BSIZE) // Initializing buffer          SHM3[ix++] = 0;        struct timeval t;      time\_t t1, t2;      gettimeofday(&t, NULL);      t1 = t.tv\_sec;        int\* state = (int\*)shmat(shmid4, NULL, 0);      \*state = 1;      int wait\_time;        int i = 0; // Consumer      int j = 1; // Producer        if (fork() == 0) // Producer code      {          SHM1 = (bool\*)shmat(shmid1, NULL, 0);          SHM2 = (int\*)shmat(shmid2, NULL, 0);          SHM3 = (int\*)shmat(shmid3, NULL, 0);          if (SHM1 == (bool\*)-1 || SHM2 == (int\*)-1 || SHM3 == (int\*)-1) {              perror("Producer shmat error: ");              exit(1);          }            bool\* flag = SHM1;          int\* turn = SHM2;          int\* buf = SHM3;          int index = 0;            while (\*state == 1) {              flag[j] = true;              printf("Producer is ready now.\n\n");              \*turn = i;              while (flag[i] == true && \*turn == i)                  ;                // Critical Section Begin              index = 0;              while (index < BSIZE) {                  if (buf[index] == 0) {                      int tempo = myrand(BSIZE \* 3);                      printf("Job %d has been produced\n", tempo);                      buf[index] = tempo;                      break;                  }                  index++;              }              if (index == BSIZE)                  printf("Buffer is full, nothing can be produced!!!\n");              printf("Buffer: ");              index = 0;              while (index < BSIZE)                  printf("%d ", buf[index++]);              printf("\n");              // Critical Section End                flag[j] = false;              if (\*state == 0)                  break;              wait\_time = myrand(PWT);              printf("Producer will wait for %d seconds\n\n", wait\_time);              sleep(wait\_time);          }          exit(0);      }        if (fork() == 0) // Consumer code      {          SHM1 = (bool\*)shmat(shmid1, NULL, 0);          SHM2 = (int\*)shmat(shmid2, NULL, 0);          SHM3 = (int\*)shmat(shmid3, NULL, 0);          if (SHM1 == (bool\*)-1 || SHM2 == (int\*)-1 || SHM3 == (int\*)-1) {              perror("Consumer shmat error:");              exit(1);          }            bool\* flag = SHM1;          int\* turn = SHM2;          int\* buf = SHM3;          int index = 0;          flag[i] = false;          sleep(5);          while (\*state == 1) {              flag[i] = true;              printf("Consumer is ready now.\n\n");              \*turn = j;              while (flag[j] == true && \*turn == j)                  ;                // Critical Section Begin              if (buf[0] != 0) {                  printf("Job %d has been consumed\n", buf[0]);                  buf[0] = 0;                  index = 1;                  while (index < BSIZE) // Shifting remaining jobs forward                  {                      buf[index - 1] = buf[index];                      index++;                  }                  buf[index - 1] = 0;              } else                  printf("Buffer is empty, nothing can be consumed!!!\n");              printf("Buffer: ");              index = 0;              while (index < BSIZE)                  printf("%d ", buf[index++]);              printf("\n");              // Critical Section End                flag[i] = false;              if (\*state == 0)                  break;              wait\_time = myrand(CWT);              printf("Consumer will sleep for %d seconds\n\n", wait\_time);              sleep(wait\_time);          }          exit(0);      }      // Parent process will now for RT seconds before causing child to terminate      while (1) {          gettimeofday(&t, NULL);          t2 = t.tv\_sec;          if (t2 - t1 > RT) // Program will exit after RT seconds          {              \*state = 0;              break;          }      }      // Waiting for both processes to exit      wait();      wait();      printf("The clock ran out.\n");      return 0;  } |

Output:

Producer is ready now.

Job 9 has been produced

Buffer: 9 0 0 0 0 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 8 has been produced

Buffer: 9 8 0 0 0 0 0 0

Producer will wait for 2 seconds

Producer is ready now.

Job 13 has been produced

Buffer: 9 8 13 0 0 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 23 has been produced

Buffer: 9 8 13 23 0 0 0 0

Producer will wait for 1 seconds

Consumer is ready now.

Job 9 has been consumed

Buffer: 8 13 23 0 0 0 0 0

Consumer will sleep for 9 seconds

Producer is ready now.

Job 15 has been produced

Buffer: 8 13 23 15 0 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 13 has been produced

Buffer: 8 13 23 15 13 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 11 has been produced

Buffer: 8 13 23 15 13 11 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 22 has been produced

Buffer: 8 13 23 15 13 11 22 0

Producer will wait for 2 seconds

Producer is ready now.

Job 23 has been produced

Buffer: 8 13 23 15 13 11 22 23

Producer will wait for 1 seconds

**Lab Outcome**: Students are able to understand Peterson’s algorithm.

**Conclusion**: Peterson’s Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.

**Experiment No. 6**

**Aim:-**To implement Binary Semaphore for Process Synchronization.

**Lab objective**:-Describe Semaphore & use of semaphore for Process Synchronization.

**Theory:-**

Semaphore is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.

It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.

A semaphore either allows or disallows access to the resource, which depends on how it is set up.

Semaphores are of two types:

1. **Binary Semaphore –**   
   This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore –**   
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

**Implementing Semaphores**

Semaphores are implemented in the system kernel.

– The semaphore values are kept in a table stored in kernel memory. A semaphore is identified by a number corresponding to a position in this table.

– There are system calls for creating or freeing semaphores, as well as for executing the wait and signal operations. These operations are executed in supervisor mode and hence atomically (interrupts are disabled in supervisor mode).

**Result:-**

**Code:**

import threading

import time

parkRequests = 0

removeRequests = 0

parked = 0

removed = 0

parkedLock = threading.Lock()

removedLock = threading.Lock()

availbleParkings = threading.Semaphore(10)

def ParkCar():

availbleParkings.acquire()

global parkedLock

parkedLock.acquire()

global parked

parked = parked+1

parkedLock.release()

print("Parked: %d"%(parked))

def RemoveCar():

availbleParkings.release()

global removedLock

removedLock.acquire()

global removed

removed = removed+1

removedLock.release()

print("Removed: %d"%(removed))

# Thread that simulates the entry of cars into the parking lot

def parkingEntry():

# Creates multiple threads inside to simulate cars that are parked

while(True):

time.sleep(1)

incomingCar = threading.Thread(target=ParkCar)

incomingCar.start()

global parkRequests

parkRequests = parkRequests+1

print("Parking Requests: %d"%(parkRequests))

# Thread that simulates the exit of cars from the parking lot

def parkingExit():

# Creates multiple threads inside to simulate cars taken out from the parking lot

while(True):

time.sleep(3)

outgoingCar = threading.Thread(target=RemoveCar)

outgoingCar.start()

global removeRequests

removeRequests = removeRequests+1

print("Remove Requests: %d"%(removeRequests))

# Start the parking eco-system

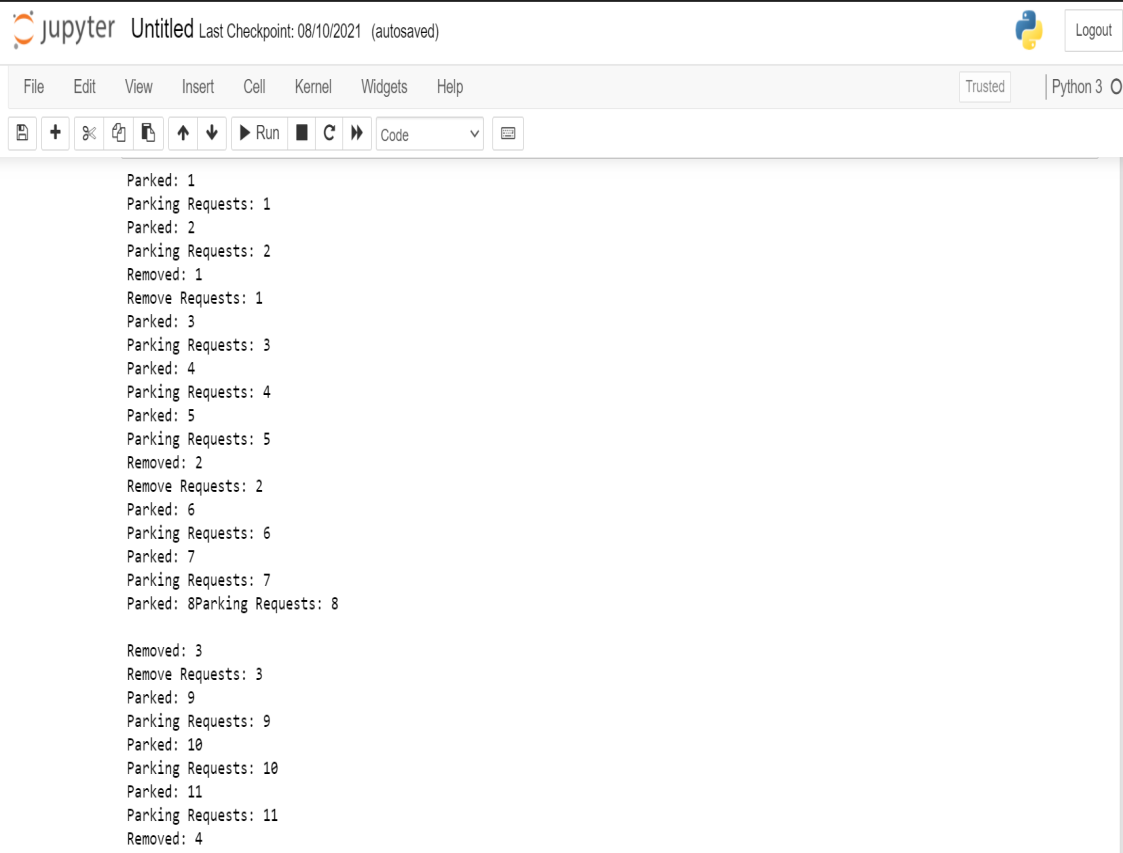
parkingEntryThread = threading.Thread(target=parkingEntry)

parkingExitThread = threading.Thread(target=parkingExit)

parkingEntryThread.start()

parkingExitThread.start()

**Output:**



**Lab Outcome:-**

Semaphore is a [variable](https://en.wikipedia.org/wiki/Variable_(programming)) or [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type) used to control access to a common resource by multiple [processes](https://en.wikipedia.org/wiki/Process_(computing)) and avoid [critical section](https://en.wikipedia.org/wiki/Critical_section) problems in a [concurrent](https://en.wikipedia.org/wiki/Concurrent_computing) system such as a [multitasking](https://en.wikipedia.org/wiki/Computer_multitasking) operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

**Conclusion:-**

* Mutual exclusion is achieved for n number of processes using Binary semaphores.
* Progress is also achieved using binary semaphores

EXPERIMENT-7

**Aim**: - To implement Banker’s algorithm in JAVA.

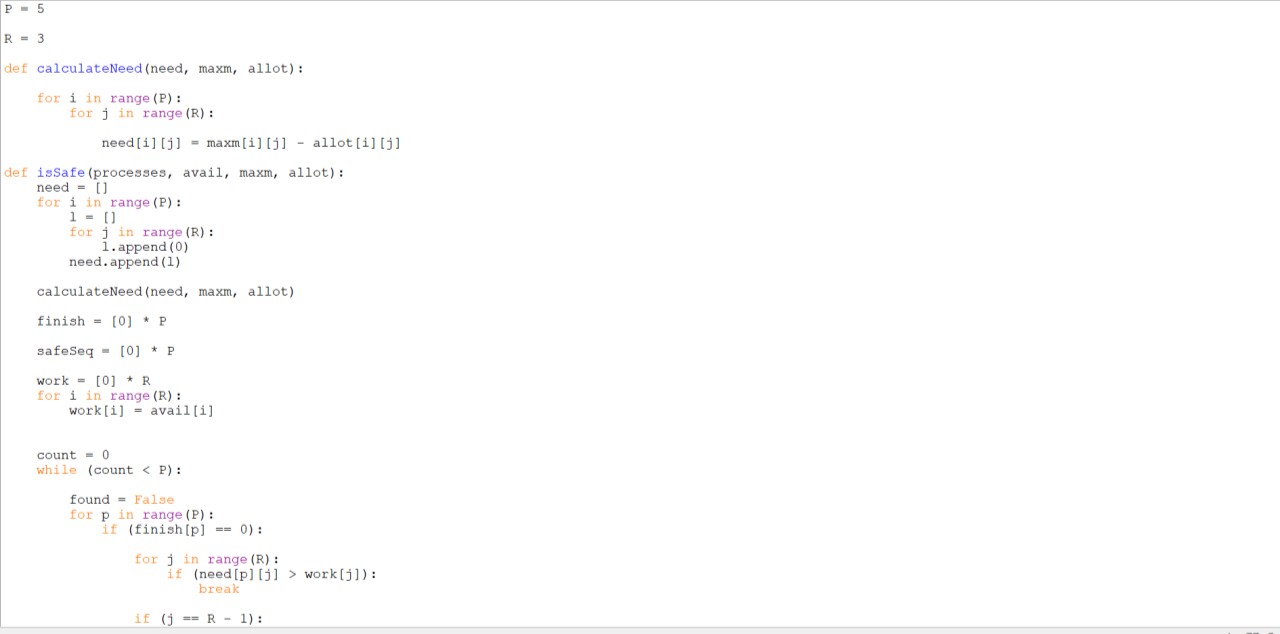
**Learning objective**: To study deadlock avoidance algorithm

**Theory**:-

Consider a system with fixed number of process and fixed number of resources. At any time a process may have zero or more resources allocated to it. The state of system reflects the current allocation to processes. A safe state is one in which there is a process that does not result in deadlock.

Banker’s Algorithm is a resource allocation and deadlock avoidance algorithm. This algorithm test for safety simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue. When a process make a request for set of processes and resources assume that the request is granted, update the system state accordingly and then determine if result is in safe state. If so grant request and if not block process until it is safe to grant.

**Result**:- Calculate Safety Sequence of given example to avoid deadlock.

**Code**:



**Output**:



**Discussion**:-

Banker’s Algorithm is basically used to find whether the system can execute without deadlock or not. In this we first try to calculate the number of resources more a process needs to complete it’s execution. Later, we try to allocate resources one after other and see to it that the same resource is not being asked by other resource at the same time. Finally, we suggest a safe sequence in which the system can execute without any dealock.

**Lab Outcome**:-

Banker’s algorithm is used to find if a safe sequence exist or not, thus avoiding a deadlock in a system.

**Conclusion**: -

The algorithm checks if allocation of any resource will lead to deadlock or not, OR is it safe to allocate a resource to a process and if not then resource is not allocated to that process.

Determining a safe sequence (even if there is only 1) will assure that system will not go into deadlock.

**EXPERIMENT NO – 8**

**Aim:-**To implement Producer Consumer Problem Solution for Process Synchronization.

**Learning objective**:-Understand Producer Consumer Problem & implement solution for the same in r Process Synchronization.

**Theory:-**

Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

**What is the Problem Statement?**

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

**Implementing Semaphores**

* One solution of this problem is to use semaphores. The semaphores which will be used here are:
* A binary semaphore which is used to acquire and release the lock.
* Empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
* Full, a counting semaphores whose initial value is 0.
* At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

**Result:-**

**Implemented python code using thread and semaphore packages.**

**Learned how to solve producer-consumer problem.**

**Used binary semaphore to solve the problem.**

**Code:**

from threading import Thread, Semaphore

import time

import random

queue = []

MAX\_NUM = 10

sem = Semaphore()

class ProducerThread(Thread):

    def run(self):

        nums = range(5)

        global queue

        while True:

            sem.acquire()

            if len(queue) == MAX\_NUM:

                print ("List is full, producer will wait")

                sem.release()

                print ("Space in queue, Consumer notified the producer")

            num = random.choice(nums)

            queue.append(num)

            print ("Produced", num)

            sem.release()

            time.sleep(random.random())

class ConsumerThread(Thread):

    def run(self):

        global queue

        while True:

            sem.acquire()

            if not queue:

                print ("List is empty, consumer waiting")

                sem.release()

                print ("Producer added something to queue and notified the consumer")

            num = queue.pop(0)

            print ("Consumed", num)

            sem.release()

            time.sleep(random.random())

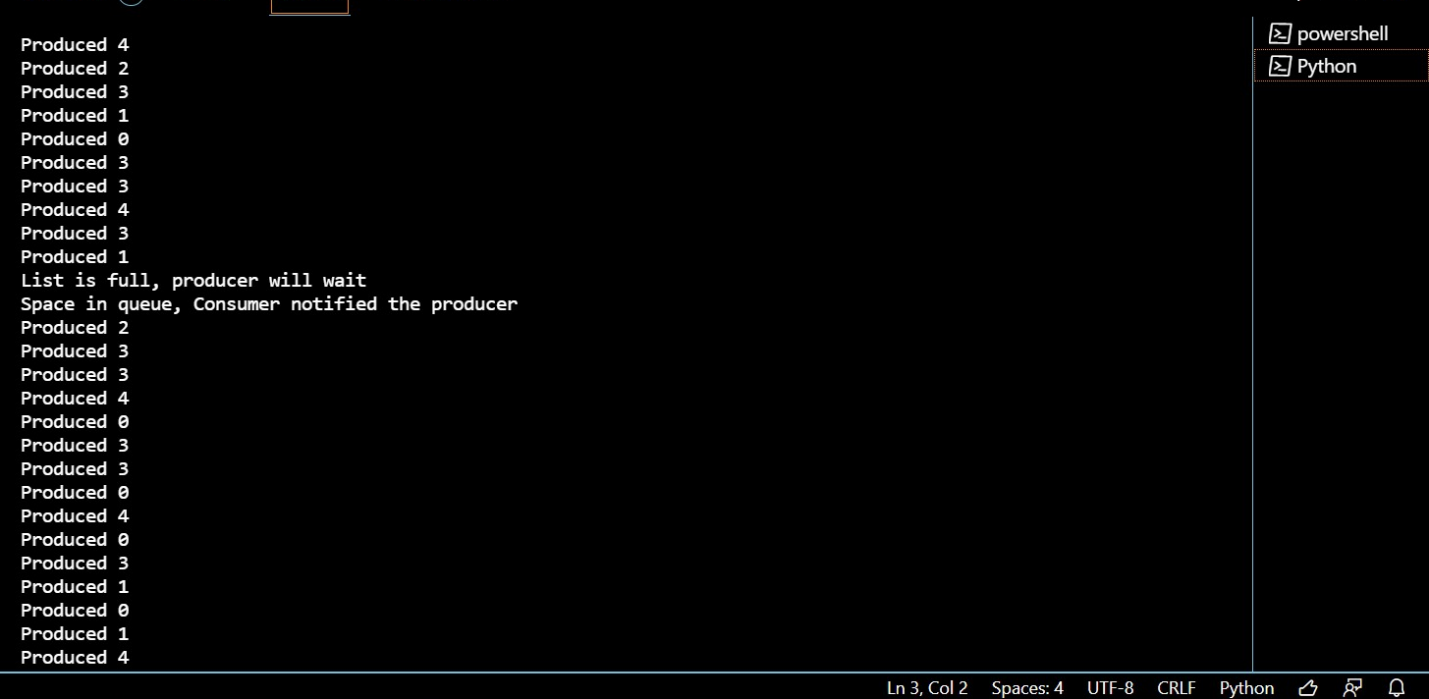
def main():

    ProducerThread().start()

    ConsumerThread().start()

if \_\_name\_\_ == '\_\_main\_\_':

 main()

**Output:** 

(this continues till there is a keyboard interrupt)

**Discussion:** In this experiment we understood the producer consumer problem . We also figured out the ways to over come this problem . we also implemented this in a python code using thread and semaphore packages . We implemented this on the basis of semaphore logic where there is a wait and signal functions. We have used two semaphores for implementing this code. The out of the program is shown above.

**Lab Outcome:-**

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes. The producer consumer problem can be resolved using semaphores.

**Conclusion:-**

Two counting Semaphores Empty & Full are used for solving this problem of producer Consumer.

## EXPERIMENT-9

**Aim: -** To implement FIFO & LRU page replacement algorithm

Lab objective: To study memory management techniques.

## Theory:-

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault). The page replacement is all about determining the page number which needs to be replaced in order to make space for the requested page.

To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference.

LRU algorithm replaces the page which has not been referred for a long time.

## Result:

**Code:**

capacity = 3 processList = [ 4, 7, 6, 1, 7,

6, 1, 2, 7, 2]

s = []

pageFaults = 0 # pageHits = 0

for i in processList: if i not in s:

# Check if the list can hold equal pages if(len(s) == capacity):

s.remove(s[0]) s.append(i)

else:

s.append(i)

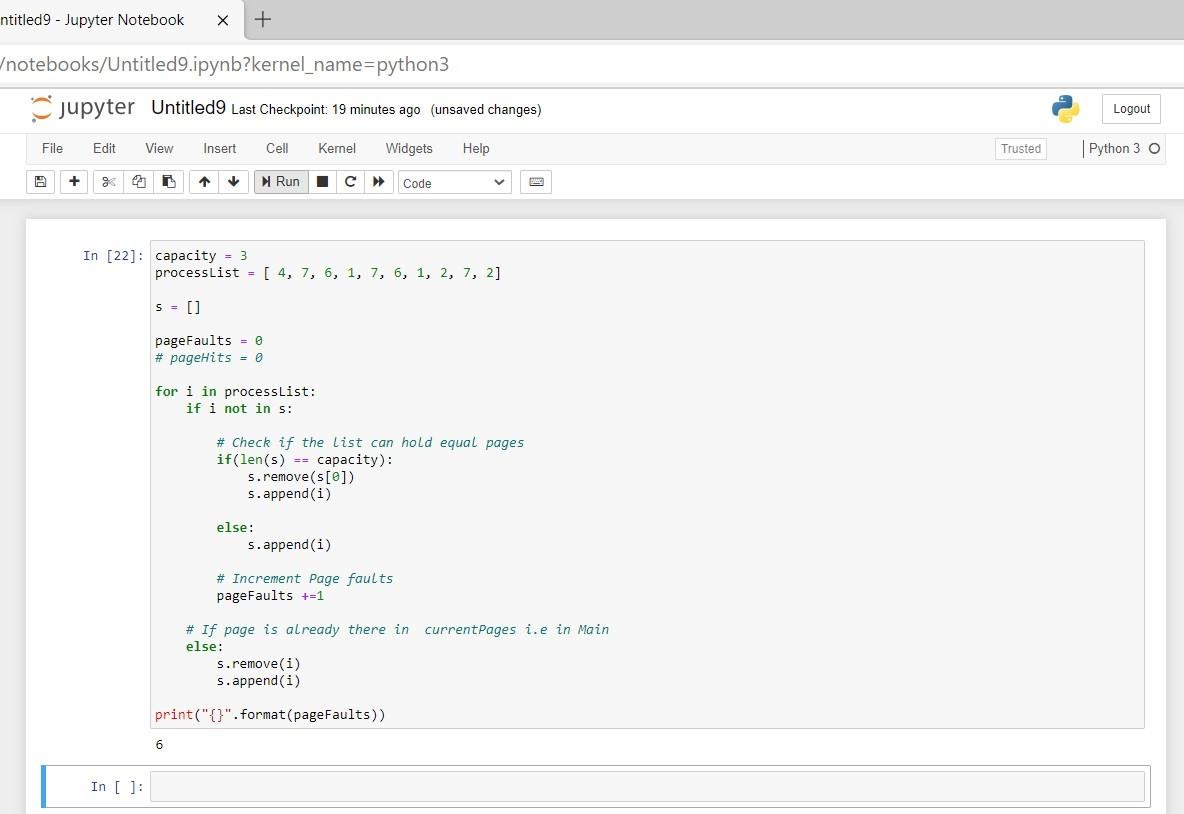
# Increment Page faults pageFaults +=1

# If page is already there in currentPages i.e in Main else:

s.remove(i) s.append(i)

print("{}".format(pageFaults))

## Output:



## Result:

**Code:** from queue import Queue

# Function to find page faults using FIFO def pageFaults(pages, n, capacity):

s = set()

# To store the pages in FIFO manner indexes = Queue()

# Start from initial page page\_faults = 0

for i in range(n):

# Check if the set can hold more pages if (len(s) < capacity):

if (pages[i] not in s):

s.add(pages[i]) page\_faults += 1 indexes.put(pages[i]) else: if (pages[i] not in s):

val = indexes.queue[0] indexes.get()

s.remove(val) s.add(pages[i])

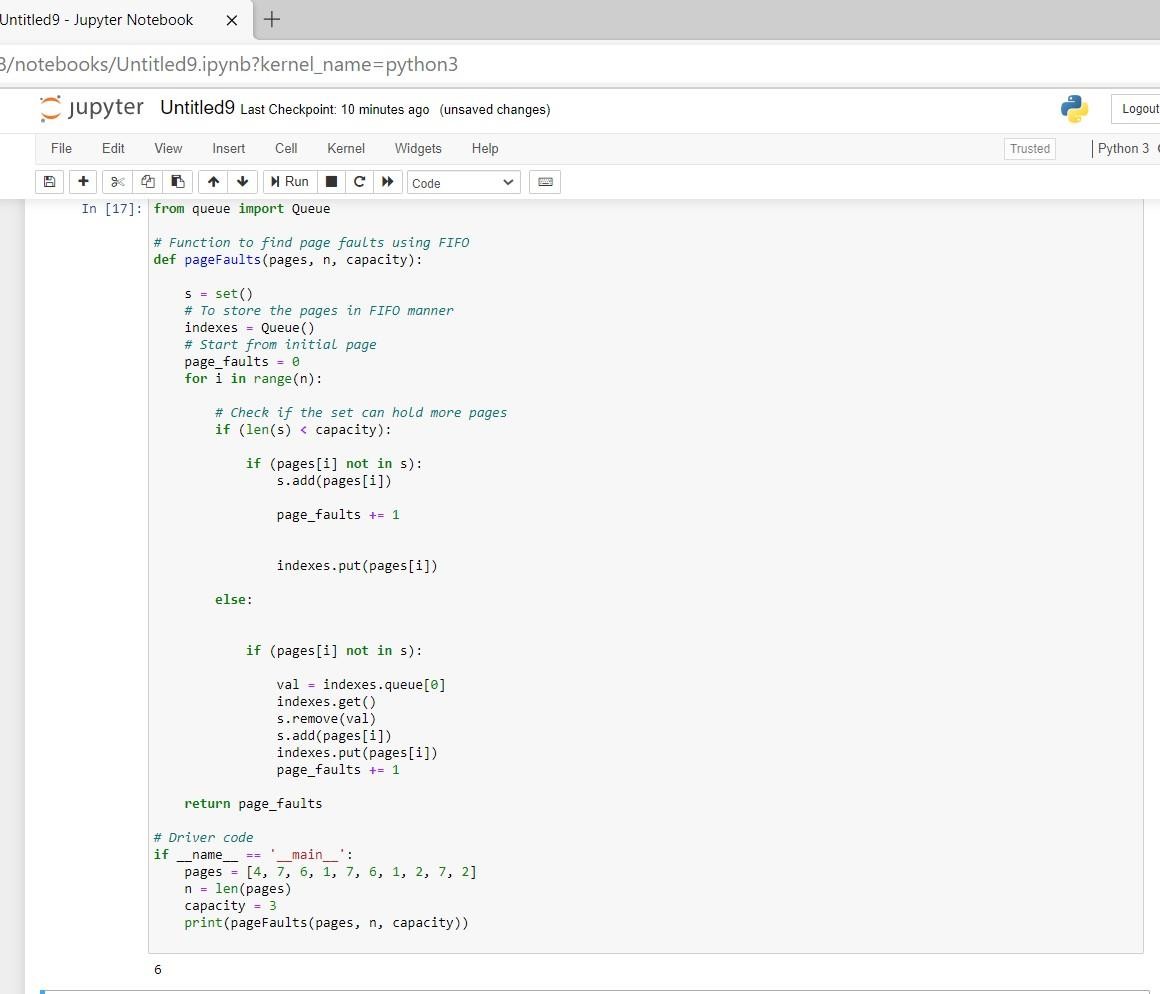
indexes.put(pages[i]) page\_faults += 1 return page\_faults # Driver code if

name == ' main ':

pages = [4, 7, 6, 1, 7, 6, 1, 2, 7, 2] n

= len(pages) capacity = 3 print(pageFaults(pages, n, capacity))

**Output:**



**Discussion:** FIFO stands for First In First Out. This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal. Initially, all the slots are empty and the slots are made depending upon the number of frames given in the question. Accordingly, by following the principle of first in first out, the pages are filled in the memory. At the end, numbers of page faults and number of page hits are found. LRU stands for Least Recently Used Page Replacement algorithm. LRU is a Greedy algorithm where the page to be replaced is least recently used. Here the number of frames is three. So we start filling the empty slots one by one. So, once we fill 4,7 and 6; we look for 1. So, we look for the page that was least recently used and replace it with 1, now there occurs a page fault. This way we calculate the page fault and page hits.

## Lab outcome:

In this algorithm, a queue is maintained. The page which is assigned the frame first will be replaced first.

## Conclusion:-

Page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.