BANKER’S ALGORITHM

<u>Group members</u>

Sudanshi Sehgal (2021A1R156)

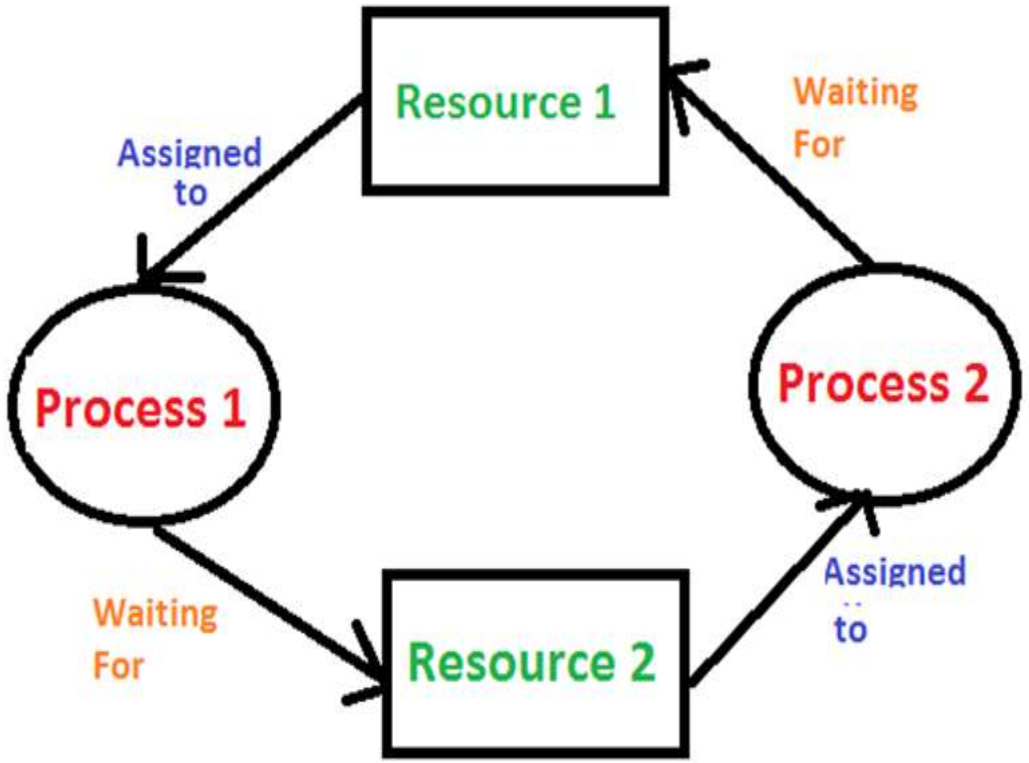
Souhani Gandotra (2021A1R167)

Simrat Singh (2021A1R170)

Anjali Salaria (2021A1R181)

Problem Statement:-

To implement the most representative banker’s algorithm for deadlock avoidance

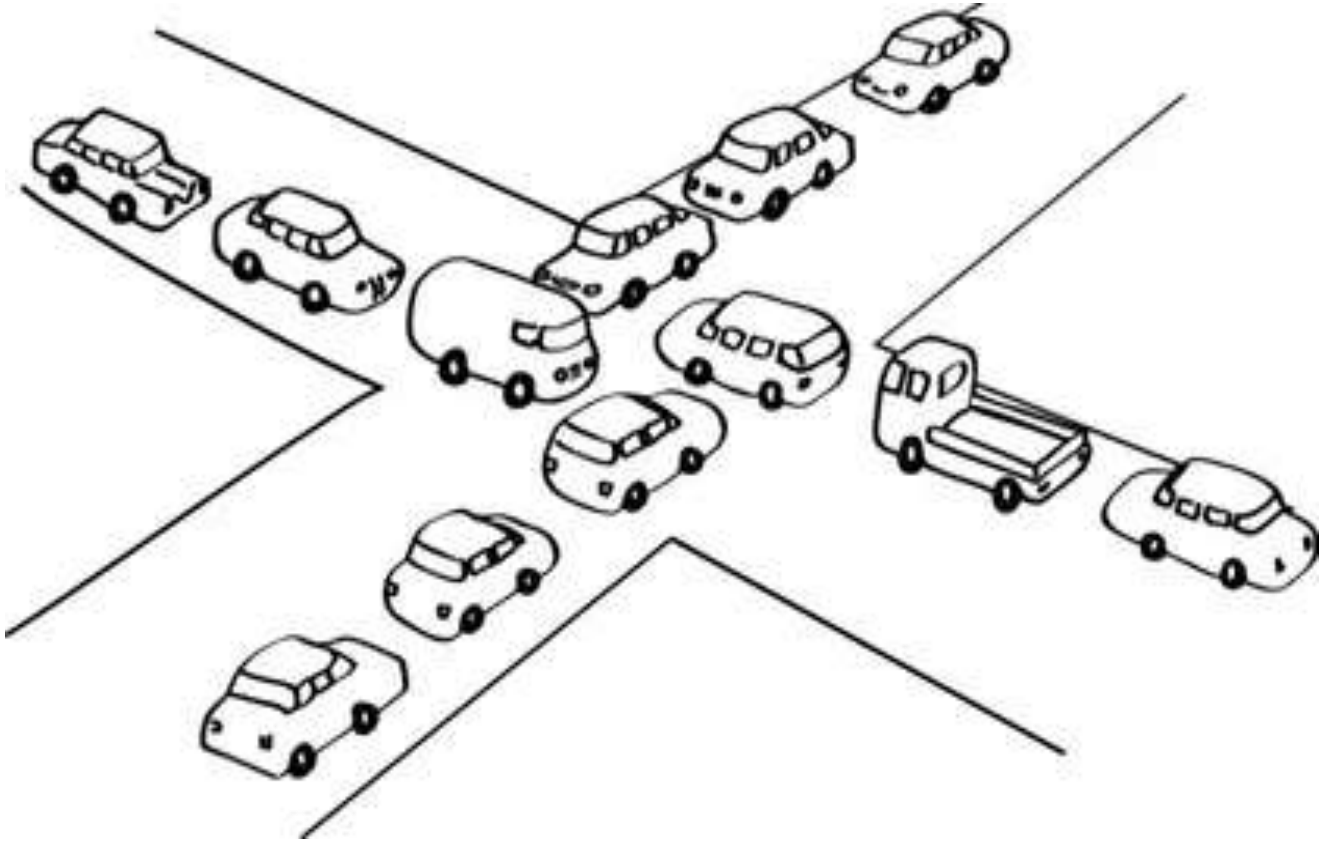


What is Deadlock?

- Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

A Real-World Example

- A real world example would be traffic, which is going only in one direction.
- Here, a bridge is considered a resource.
- So, when Deadlock happens, it can be easily resolved if one car backs up.
- Several cars may have to be backed up if a deadlock situation occurs.
- So starvation is possible.



The Conditions for Deadlock

Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)

1. Mutual Exclusion:

Only one process at a time may use a shared resource (i.e. critical section).

2. Hold and wait:

A process may hold allocated resources while awaiting assignment of others.

3. No pre-emption:

A resource can be released only voluntarily by the process holding it, after that process holding it, after that process has completed its task

4. Circular Wait:

A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

The Conditions for Deadlock

- Deadlock occurs if and only if the circular wait condition is unresolvable
- The circular wait condition is unresolvable if the first 3 policy condition hold
- So the 4 conditions taken together constitute necessary and sufficient conditions for deadlock

Requirements

Banker’s Algorithm

Linux Operating System

Coding Platforms:

Nano editor / vi editor

Banker’s Algorithm

- Banker’s Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resource, it checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it does not allow the request made by the process.
- **Inputs to Banker’s Algorithm:**

1. Max need of resources by each process.
2. Currently, allocated resources by each process.
3. Max free available resources in the system.

- **The request will only be granted under the below condition:**

1. If the request made by the process is less than equal to max need to that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

Example of Banker's Algorithm

Example:
Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances.

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (contd.)

Need = Max- Allocation

	Need		
	A	**B**	**C**
P0	**7**	**4**	**3**
P1	**1**	**2**	**2**
P2	**6**	**0**	**0**
P3	**0**	**1**	**1**
P4	**4**	**3**	**1**

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies criteria.

Check that Request <= Available

(that is, (1,2,2)<=(3,3,2)) is true

Now P1 requests (1,2,2)

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Pseudo Code:-

A safe sequence of processes, or an empty sequence if no such sequence exists.

1.while there exists a process P in N that has not been terminated do

2.if there exists a resource R in M that P can request and be granted without causing a deadlock then

3.request R from P

4.else if P can release a resource R without causing a deadlock then

5.release R from P

6.else

7.wait

8.end if

9.end while

10.return the safe sequence

Algorithm:

Safety Algorithm

1) Let Work and Finish be Arrays of length ‘m’ and ‘n’ respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) Need i <= 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

Algorithm:

- **Request Algorithm**

1) If Request i <= Need i

Go to step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If Request i <= Available

Go to step (3); otherwise, Pi must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as

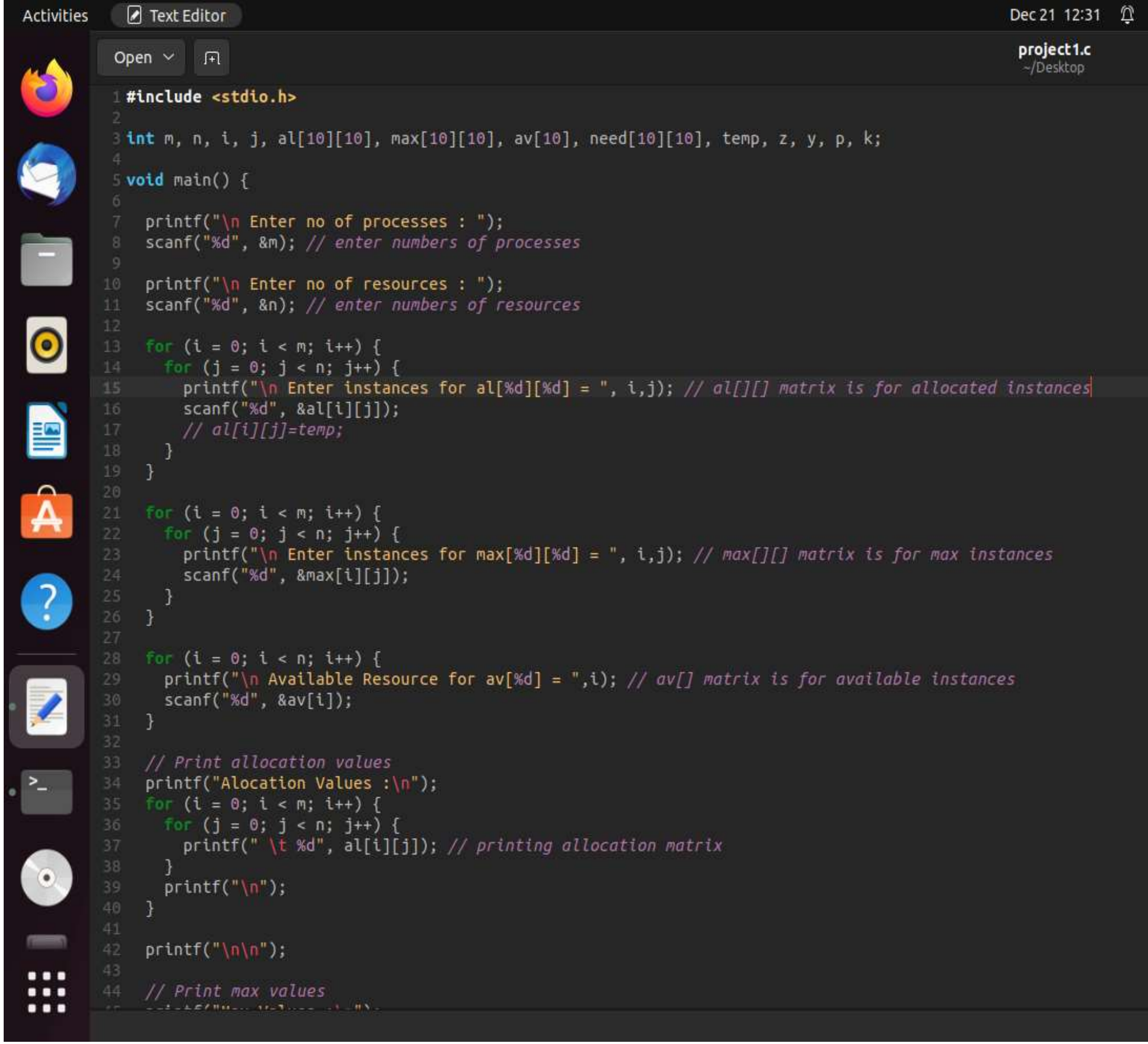
follows:

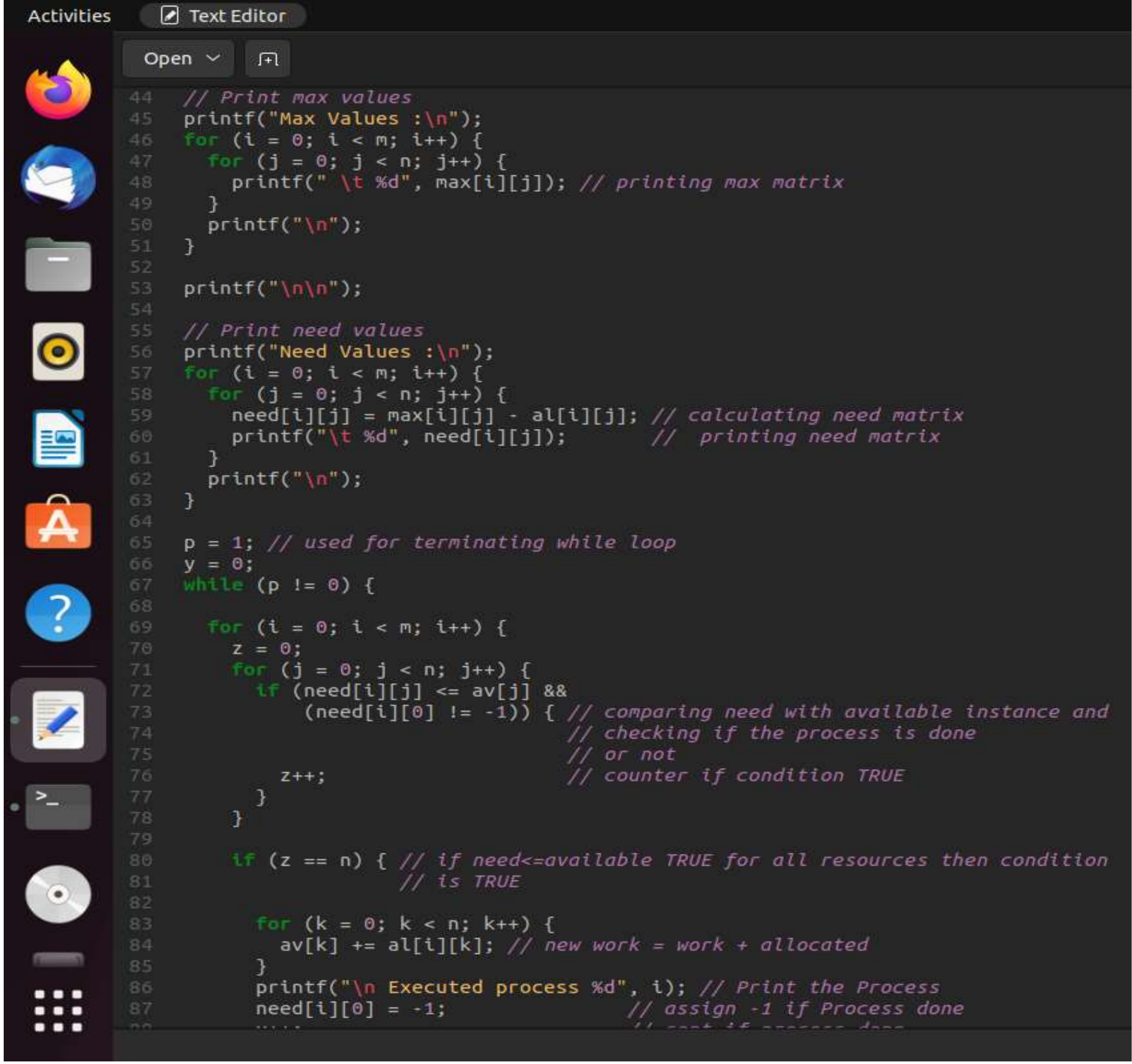
Available = Available – Request i

Allocation i = Allocation i + Request i

Need i = Need i– Request i

TECHNICAL DETAILS (CODING)



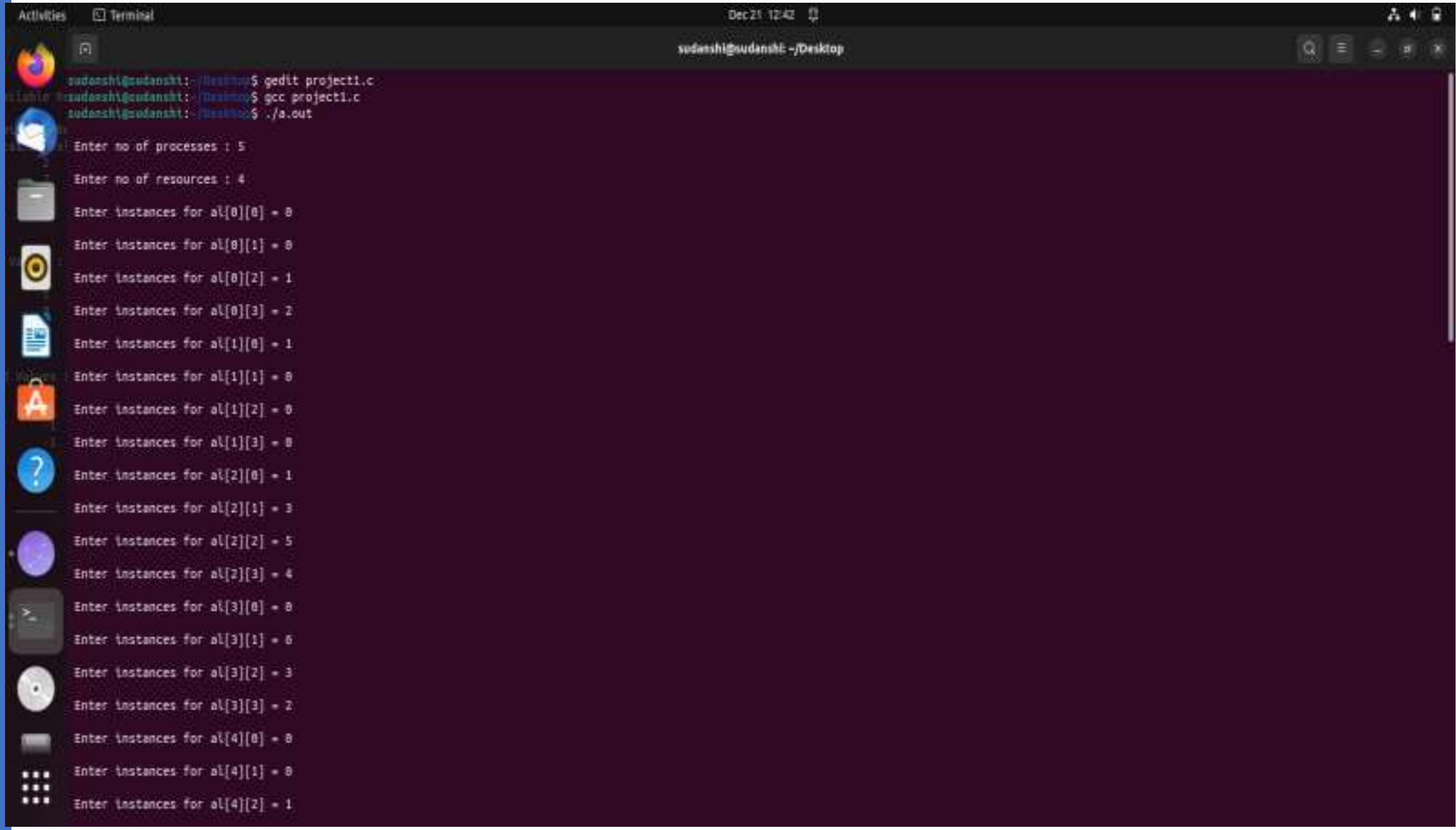


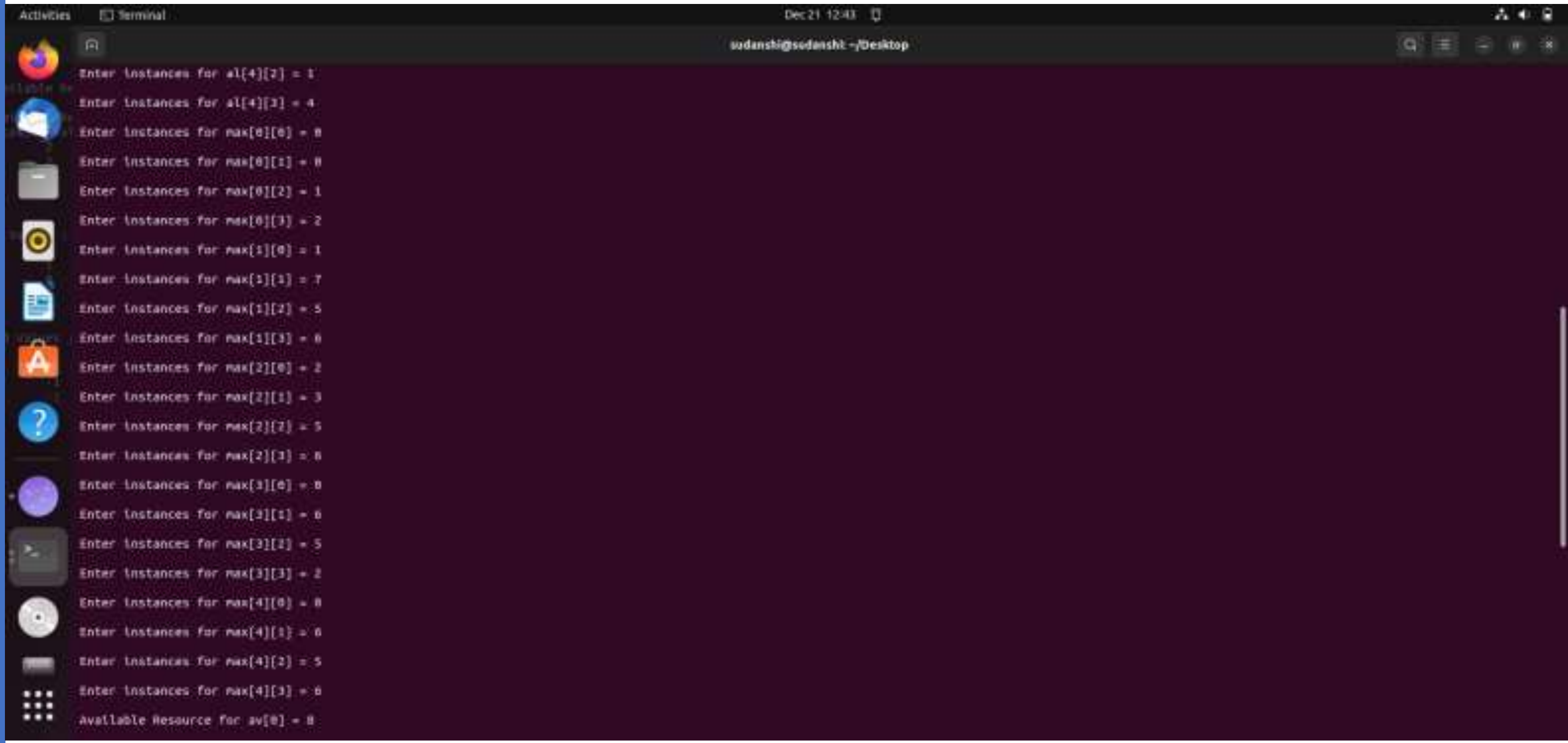

```
// Print max values
printf("Max Values :\n");
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
    printf(" \t %d", max[i][j]); // printing max matrix
  }
  printf("\n");
}

printf("\n\n");

// Print need values
printf("Need Values :\n");
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
    need[i][j] = max[i][j] - al[i][j]; // calculating need matrix
    printf("\t %d", need[i][j]);        //  printing need matrix
  }
  printf("\n");
}

p = 1; // used for terminating while loop
y = 0;
while (p != 0) {

  for (i = 0; i < m; i++) {
    z = 0;
    for (j = 0; j < n; j++) {
      if (need[i][j] <= av[j] &&
          (need[i][0] != -1)) { // comparing need with available instance and
                                // checking if the process is done
                                // or not
        z++;                    // counter if condition TRUE
      }
    }

    if (z == n) { // if need<=available TRUE for all resources then condition
                  // is TRUE

      for (k = 0; k < n; k++) {
        av[k] += al[i][k]; // new work = work + allocated
      }
      printf("\n Executed process %d", i); // Print the Process
      need[i][0] = -1;                     // assign -1 if Process done
```

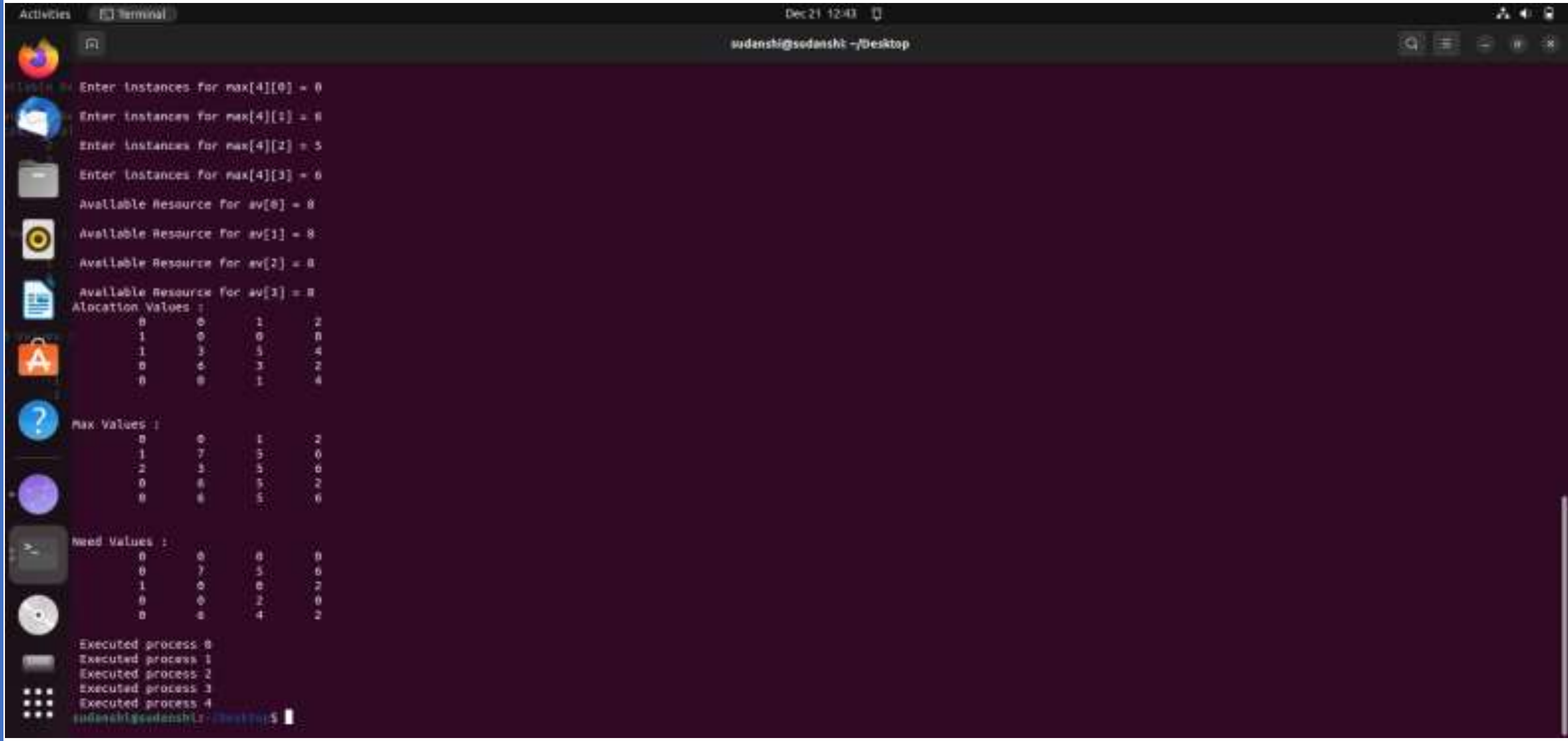
```c
            }
            printf("\n Executed process %d", i); // Print the Process
            need[i][0] = -1;                        // assign -1 if Process done
            y++;                                    // cont if process done
            }

    } // end for loop

    if (y == m) { // if all done then
      p = 0;      // exit while loop
    }

  } // end while

  printf("\n");

}
```

OUTPUT

NO DEADLOCK

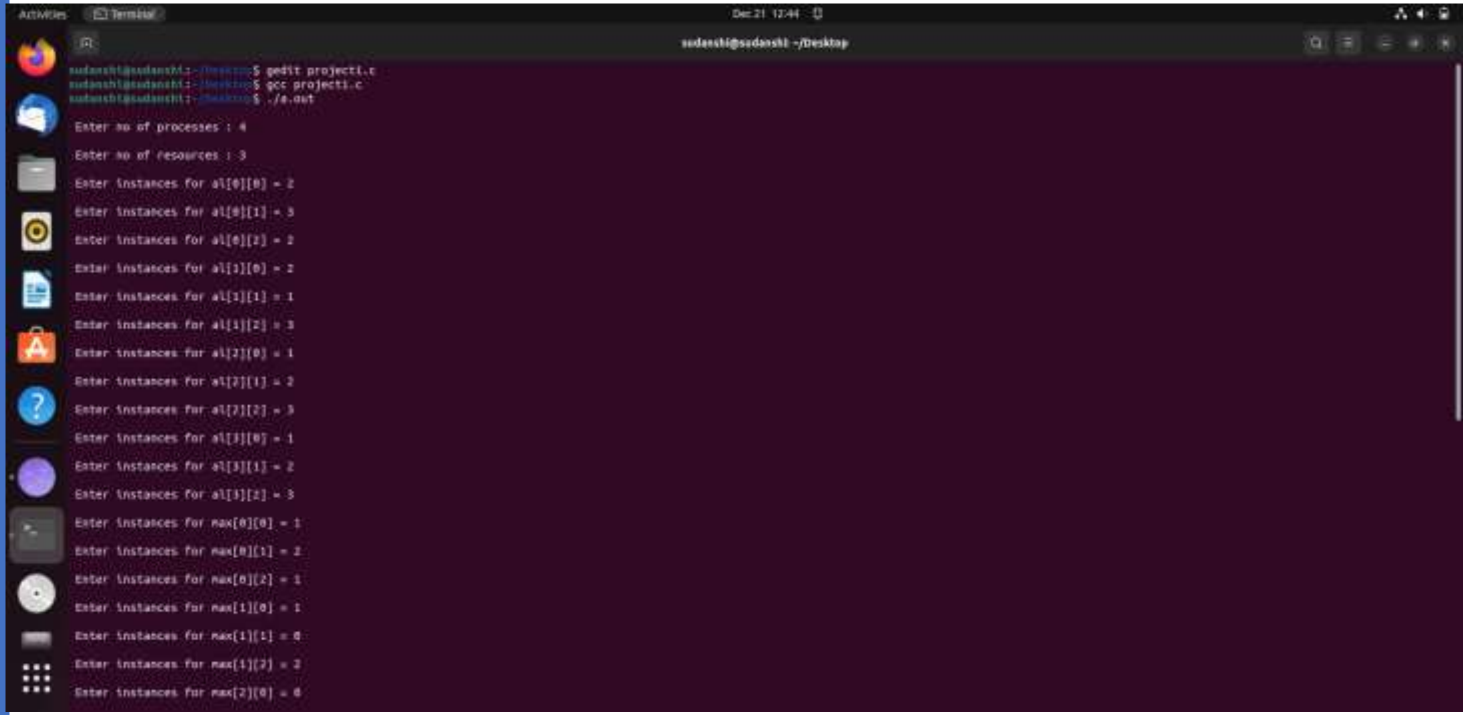


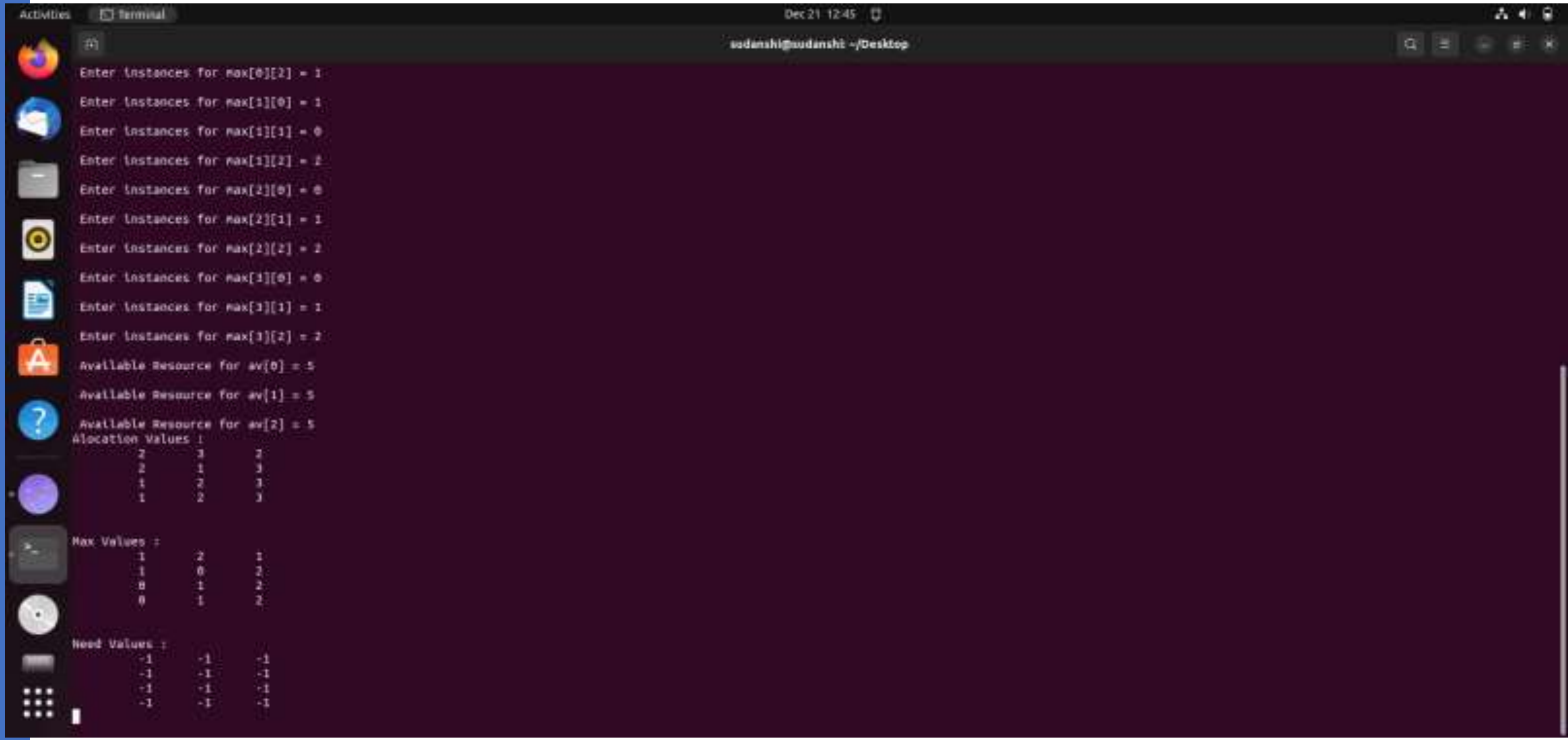

Activities
Terminal
Dec 21 12:43
sudanshi@sudanshi: ~/Desktop
Enter instances for al[4][2] = 1
Enter instances for al[4][3] = 4
Enter instances for max[0][0] = 0
Enter instances for max[0][1] = 0
Enter instances for max[0][2] = 1
Enter instances for max[0][3] = 2
Enter instances for max[1][0] = 1
Enter instances for max[1][1] = 7
Enter instances for max[1][2] = 5
Enter instances for max[1][3] = 0
Enter instances for max[2][0] = 2
Enter instances for max[2][1] = 3
Enter instances for max[2][2] = 5
Enter instances for max[2][3] = 6
Enter instances for max[3][0] = 0
Enter instances for max[3][1] = 6
Enter instances for max[3][2] = 5
Enter instances for max[3][3] = 2
Enter instances for max[4][0] = 0
Enter instances for max[4][1] = 6
Enter instances for max[4][2] = 5
Enter instances for max[4][3] = 6
Available Resource for av[0] = 0


Activities
Terminal
Dec 21 12:43
sudanshi@sudanshi: ~/Desktop
Enter instances for max[4][0] = 0
Enter instances for max[4][1] = 6
Enter instances for max[4][2] = 5
Enter instances for max[4][3] = 6
Available Resource for av[0] = 0
Available Resource for av[1] = 0
Available Resource for av[2] = 0
Available Resource for av[3] = 0
Alocation Values :
0 0 1 2
1 0 0 0
1 3 5 4
0 6 3 2
0 0 1 4
Max Values :
0 0 1 2
1 7 5 0
2 3 5 6
0 6 5 2
0 6 5 6
Need Values :
0 0 0 0
0 7 5 0
1 0 0 2
0 0 2 0
0 6 4 2
Executed process 0
Executed process 1
Executed process 2
Executed process 3
Executed process 4

OUTPUT

DEADLOCK





Activities
Terminal
Dec 21 12:45
sudanshi@sudanshi: ~/Desktop
Enter instances for max[0][2] = 1
Enter instances for max[1][0] = 1
Enter instances for max[1][1] = 0
Enter instances for max[1][2] = 2
Enter instances for max[2][0] = 0
Enter instances for max[2][1] = 1
Enter instances for max[2][2] = 2
Enter instances for max[3][0] = 0
Enter instances for max[3][1] = 1
Enter instances for max[3][2] = 2
Available Resource for av[0] = 5
Available Resource for av[1] = 5
Available Resource for av[2] = 5
Alocation Values :
2 3 2
2 1 3
1 2 3
1 2 3
Max Values :
1 2 1
1 0 2
0 1 2
0 1 2
Need Values :
-1 -1 -1
-1 -1 -1
-1 -1 -1
-1 -1 -1

THANK YOU