# ASSINGMENT

## OPERATING SYSTEM

LOVELY PROFESSIONAL UNIVERSITY

Transforming Education Transforming India

## Submitted To

**Oshin , Assistant Professor**

**Lovely Professional University**

**Jalandhar, Punjab, India.**

## Submitted By

| Sr.no | Registration Number | Student Name | Roll Number | Obtained Mark | Signature |
|---|---|---|---|---|---|
| 01. | 11815813 | Shykat Roy | RK18PTB42 | | Shykat |

## >> What is banker’s algorithm ?

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

## >>Why Banker’s algorithm is named so?

Ans:- Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Following **Data structures** are used to implement the Banker’s Algorithm:
Let **‘n’** be the number of processes in the system
and **‘m’** be the number of resources types.

**Available :**

- It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
- Available[ j ] = k means there are **‘k’** instances of resource type $\mathbf{R_j}$

**Max :**

- It is a 2-d array of size ‘**n*m’** that defines the maximum demand of each process in a system.
- Max[ i, j ] = k means process $\mathbf{P_i}$ may request at most **‘k’** instances of resource type $\mathbf{R_{j.}}$

**Allocation :**

- It is a 2-d array of size **‘n*m’** that defines the number of resources of each type currently allocated to each process.
- Allocation[ i, j ] = k means process $\mathbf{P_i}$ is currently allocated **‘k’** instances of resource type $\mathbf{R_j}$

**Need :**

- It is a 2-d array of size **‘n*m’** that indicates the remaining resource need of each process.
- Need [ i, j ] = k means process $\mathbf{P_i}$ currently need **‘k’** instances of resource type $\mathbf{R_j}$

for its execution.

- Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]

## Safety Algorithm:-

*1) Let Work and Finish be vectors 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 goto step (4)*

*3) Work = Work + Allocation[i]*

*Finish[i] = true*

*goto step (2)*

*4) if Finish [i] = true for all i*

*then the system is in a safe state*

## Resource-Request Algorithm:-

*1) If* $Request_i$ *<=* $Need_i$

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

*2) If* $Request_i$ *<= Available*

*Goto step (3); otherwise,* $P_i$ *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 – Requesti*
*$Allocation_i$ = $Allocation_i$ + $Request_i$*
*$Need_i$ = $Need_i$– $Request_i$*

## Safety Algorithm:-

```
1.  Let Work and Finish be vectors of
length m and n, respectively. Initialize
Work := Available and Finisk[i] :=false for i = 1,2…, n.
2.  Find an i such that both a. Finisk[i] =false
3.  $Need_i \leq$ Work.
  If no such i exists, go to step 4.
4.  Work := Work + $Allocation_i$
      Finisk[i] := true
        go to step 2.

5.  If Finish[i] = true for all i, then the system is in a safe
state.
```

## Characteristics of Banker's Algorithm

Here are important characteristics of banker's algorithm:

- Keep many resources that satisfy the requirement of at least one client

- Whenever a process gets all its resources, it needs to return them in a restricted period.
- When a process requests a resource, it needs to wait
- The system has a limited number of resources
- Advance feature for max resource allocation

## Disadvantage of Banker's algorithm

Here, are cons/drawbacks of using banker's algorithm

- Does not allow the process to change its Maximum need while processing
- It allows all requests to be granted in restricted time, but one year is a fixed period for that.
- All processes must know and state their maximum resource needs in advance.

## *Code:-*

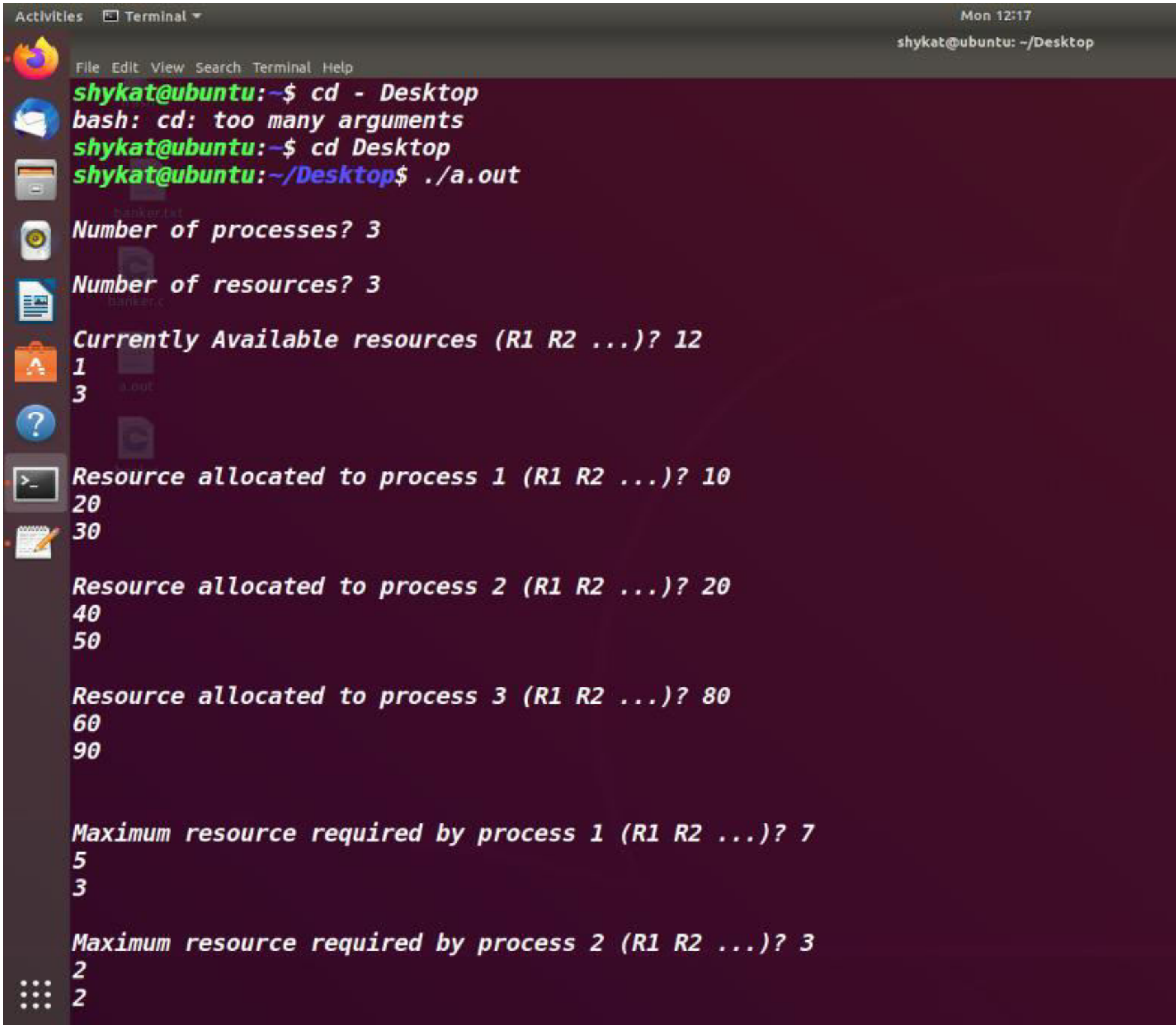
```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <stdbool.h>

#include <time.h>


int nResources,

        nProcesses;

int *resources;

int **allocated;

int **maxRequired;

int **need;

int *safeSeq;

int nProcessRan = 0;


pthread_mutex_t lockResources;
```

```
pthread_cond_t condition;

// get safe sequence is there is one else return false
bool getSafeSeq();
// process function
void* processCode(void* );

int main(int argc, char** argv) {
    srand(time(NULL));

        printf("\nNumber of processes? ");
        scanf("%d", &nProcesses);

        printf("\nNumber of resources? ");
        scanf("%d", &nResources);

        resources = (int *)malloc(nResources * sizeof(*resources));
        printf("\nCurrently Available resources (R1 R2 ...)? ");
        for(int i=0; i<nResources; i++)
                scanf("%d", &resources[i]);

        allocated = (int **)malloc(nProcesses * sizeof(*allocated));
        for(int i=0; i<nProcesses; i++)
                allocated[i] = (int )malloc(nResources *
sizeof(*allocated));


        maxRequired = (int **)malloc(nProcesses *
sizeof(*maxRequired));
        for(int i=0; i<nProcesses; i++)
                maxRequired[i] = (int )malloc(nResources *
sizeof(*maxRequired));


        // allocated
        printf("\n");
        for(int i=0; i<nProcesses; i++) {
                printf("\nResource allocated to process %d (R1 R2
...)? ", i+1);
                for(int j=0; j<nResources; j++)
                        scanf("%d", &allocated[i][j]);
        }
        printf("\n");


    // maximum required resources
        for(int i=0; i<nProcesses; i++) {
                printf("\nMaximum resource required by process
%d (R1 R2 ...)? ", i+1);
                for(int j=0; j<nResources; j++)
                        scanf("%d", &maxRequired[i][j]);
        }
        printf("\n");

    // calculate need matrix
        need = (int **)malloc(nProcesses * sizeof(*need));
        for(int i=0; i<nProcesses; i++)
                need[i] = (int )malloc(nResources * sizeof(*need));

        for(int i=0; i<nProcesses; i++)
                for(int j=0; j<nResources; j++)
                        need[i][j] = maxRequired[i][j] -
allocated[i][j];

    // get safe sequence
    safeSeq = (int *)malloc(nProcesses * sizeof(*safeSeq));
        for(int i=0; i<nProcesses; i++) safeSeq[i] = -1;

        if(!getSafeSeq()) {
                printf("\nUnsafe State! The processes leads the
system to a unsafe state.\n\n");
                exit(-1);
        }

        printf("\n\nSafe Sequence Found : ");
        for(int i=0; i<nProcesses; i++) {
                printf("%-3d", safeSeq[i]+1);
        }

        printf("\nExecuting Processes...\n\n");
        sleep(1);

    // run threads
    pthread_t processes[nProcesses];
        pthread_attr_t attr;
        pthread_attr_init(&attr);

    int processNumber[nProcesses];
    for(int i=0; i<nProcesses; i++) processNumber[i] = i;

        for(int i=0; i<nProcesses; i++)
```

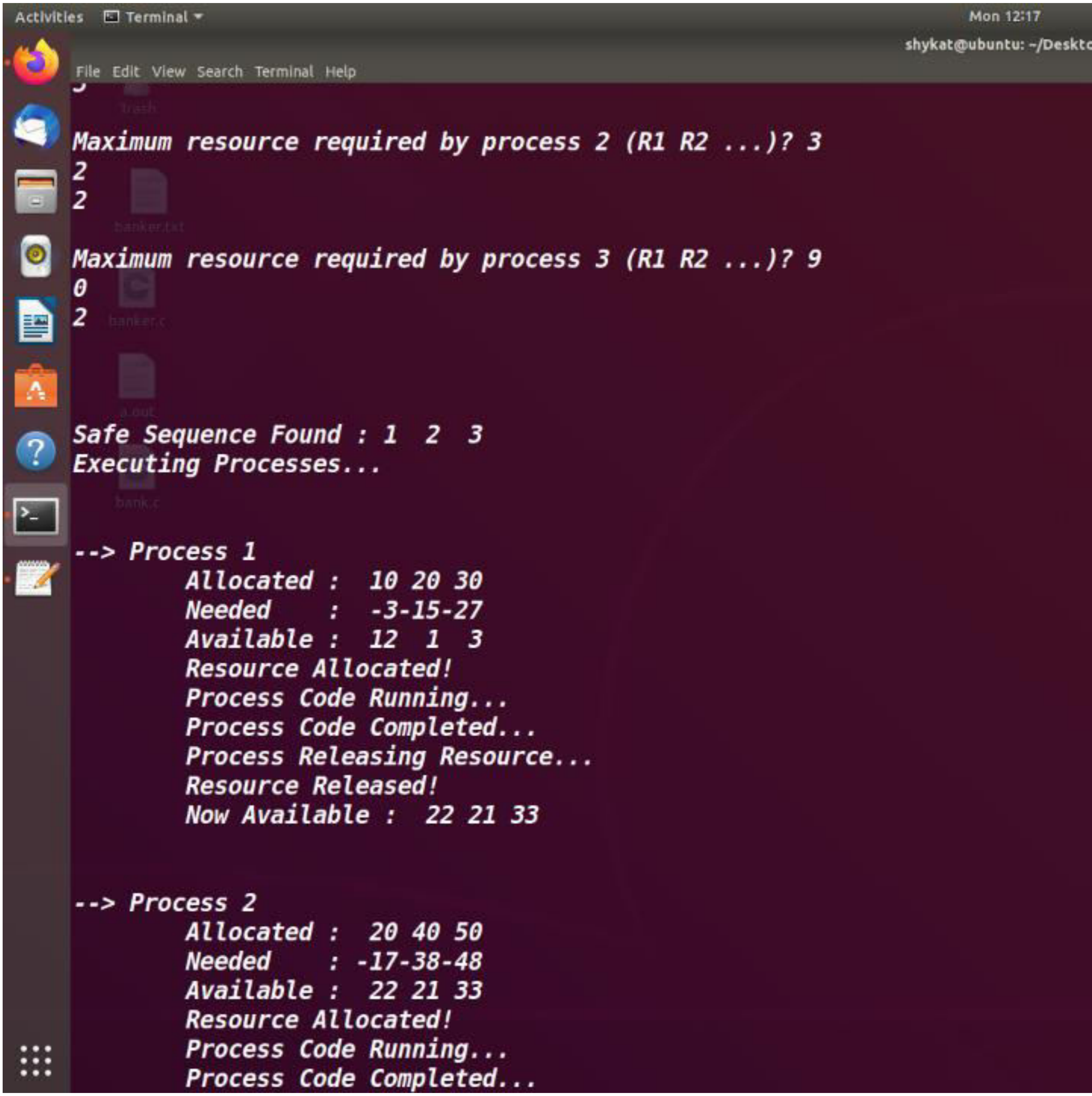
```
                pthread_create(&processes[i], &attr, processCode,
(void *)(&processNumber[i]));


        for(int i=0; i<nProcesses; i++)
                pthread_join(processes[i], NULL);


        printf("\nAll Processes Finished\n");


    // free resources
        free(resources);
        for(int i=0; i<nProcesses; i++) {
                free(allocated[i]);
                free(maxRequired[i]);
         free(need[i]);
        }
        free(allocated);
        free(maxRequired);
    free(need);
        free(safeSeq);
}
```

```
bool getSafeSeq() {
    // get safe sequence
        int tempRes[nResources];
        for(int i=0; i<nResources; i++) tempRes[i] = resources[i];

        bool finished[nProcesses];
        for(int i=0; i<nProcesses; i++) finished[i] = false;
        int nfinished=0;
        while(nfinished < nProcesses) {
                bool safe = false;

                for(int i=0; i<nProcesses; i++) {
                        if(!finished[i]) {
                                bool possible = true;

                                for(int j=0; j<nResources; j++)
                                        if(need[i][j] >
tempRes[j]) {
                                                possible =
false;
                                                break;
                                        }
                if(possible) {
                    for(int j=0;
j<nResources; j++)

tempRes[j] += allocated[i][j];
                    safeSeq[nfinished] =
i;
                    finished[i] = true;
                    ++nfinished;
                    safe = true;
                }
            }
        }

        if(!safe) {
            for(int k=0; k<nProcesses; k++)
safeSeq[k] = -1;
            return false; // no safe sequence found
        }
    }
    return true; // safe sequence found
}
```

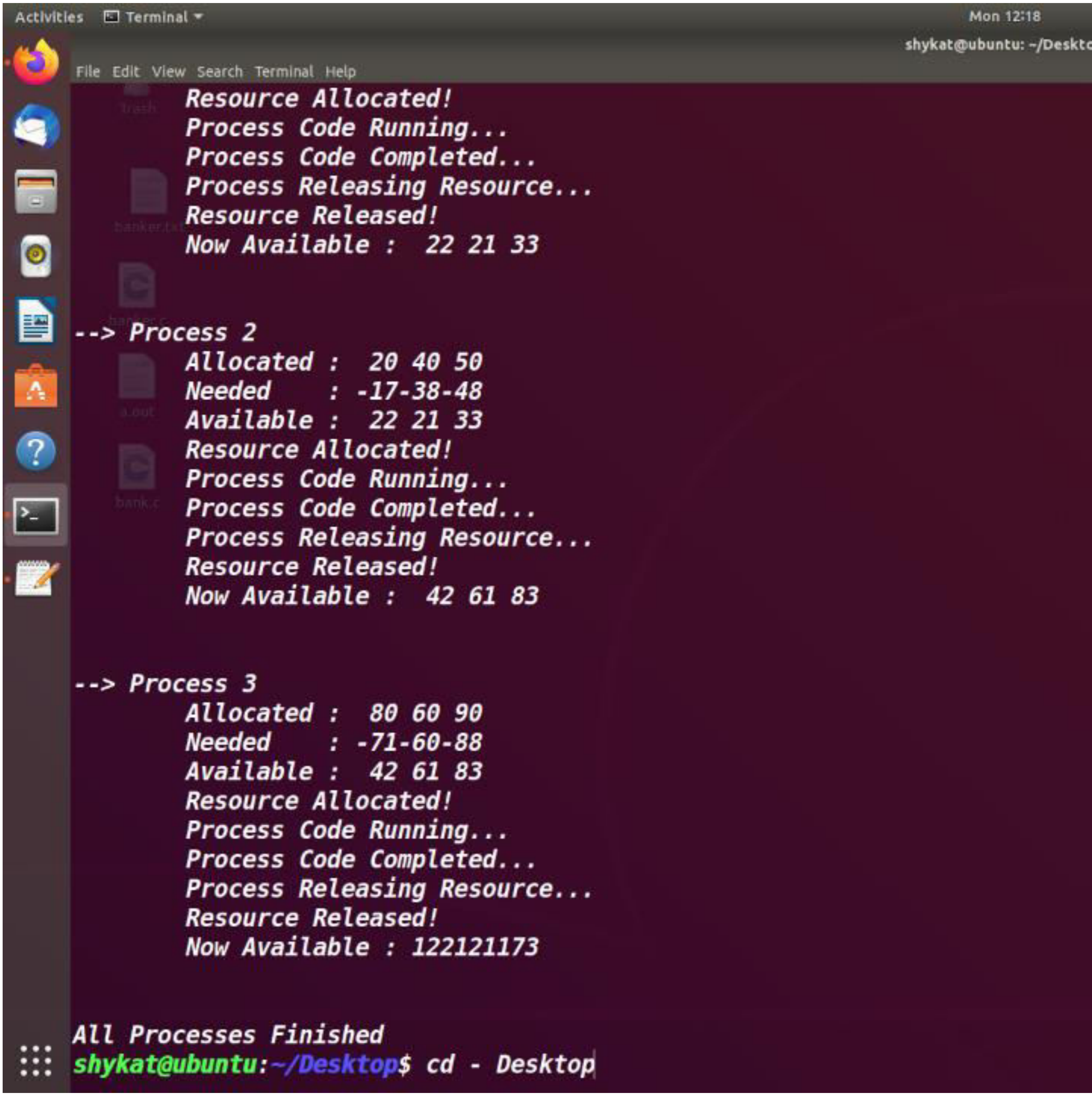
```
// process code
void* processCode(void *arg) {
        int p = *((int *) arg);

    // lock resources
        pthread_mutex_lock(&lockResources);

        // condition check
        while(p != safeSeq[nProcessRan])
                pthread_cond_wait(&condition, &lockResources);

    // process
        printf("\n--> Process %d", p+1);
        printf("\n\tAllocated : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", allocated[p][i]);

        printf("\n\tNeeded        : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", need[p][i]);

        printf("\n\tAvailable : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", resources[i]);

        printf("\n"); sleep(1);

        printf("\tResource Allocated!");
        printf("\n"); sleep(1);
        printf("\tProcess Code Running...");
        printf("\n"); sleep(rand()%3 + 2); // process code
        printf("\tProcess Code Completed...");
        printf("\n"); sleep(1);
        printf("\tProcess Releasing Resource...");
        printf("\n"); sleep(1);
        printf("\tResource Released!");

    for(int i=0; i<nResources; i++)
                resources[i] += allocated[p][i];

        printf("\n\tNow Available : ");
        for(int i=0; i<nResources; i++)
                printf("%3d", resources[i]);
        printf("\n\n");
        sleep(1);

    // condition broadcast
        nProcessRan++;
        pthread_cond_broadcast(&condition);
        pthread_mutex_unlock(&lockResources);
    pthread_exit(NULL);
}
```



```
shykat@ubuntu:~$ cd - Desktop
bash: cd: too many arguments
shykat@ubuntu:~$ cd Desktop
shykat@ubuntu:~/Desktop$ ./a.out

Number of processes? 3

Number of resources? 3

Currently Available resources (R1 R2 ...)? 12
1
3


Resource allocated to process 1 (R1 R2 ...)? 10
20
30

Resource allocated to process 2 (R1 R2 ...)? 20
40
50

Resource allocated to process 3 (R1 R2 ...)? 80
60
90


Maximum resource required by process 1 (R1 R2 ...)? 7
5
3

Maximum resource required by process 2 (R1 R2 ...)? 3
2
2
```



```
3

Maximum resource required by process 2 (R1 R2 ...)? 3
2
2

Maximum resource required by process 3 (R1 R2 ...)? 9
0
2



Safe Sequence Found : 1  2  3
Executing Processes...


--> Process 1
        Allocated :  10 20 30
        Needed    :  -3-15-27
        Available :  12  1  3
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available :  22 21 33


--> Process 2
        Allocated :  20 40 50
        Needed    : -17-38-48
        Available :  22 21 33
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
```



```
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available :  22 21 33


--> Process 2
        Allocated :  20 40 50
        Needed     : -17-38-48
        Available :  22 21 33
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available :  42 61 83


--> Process 3
        Allocated :  80 60 90
        Needed     : -71-60-88
        Available :  42 61 83
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available : 122121173


All Processes Finished
shykat@ubuntu:~/Desktop$ cd - Desktop
```