**Student Name:** vandana kumari

**Student ID:** 11808191

**Email Address:** [vandanagautam@gmail.com](mailto:vandanagautam@gmail.com)

**GitHub Link:** <https://github.com/vandana1602/Os-project.git>

**Problem:**

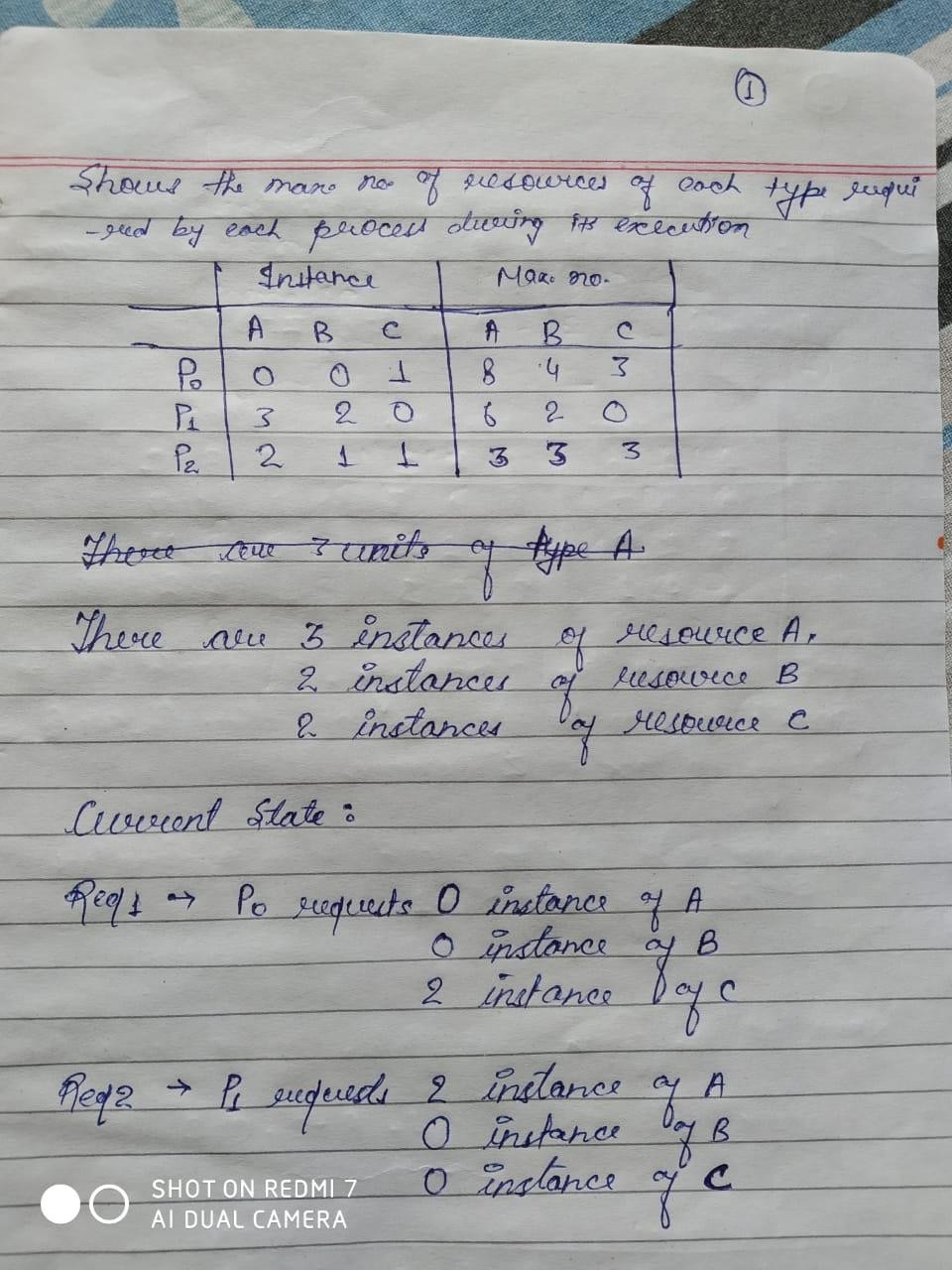
**Ques11**. Reena’s operating system uses an algorithm for deadlock avoidance to manage the allocation of resources say three namely A, B, and C to three processes P0, P1, and P2. Consider the following scenario as reference .user must enter the current state of system as given in this example :

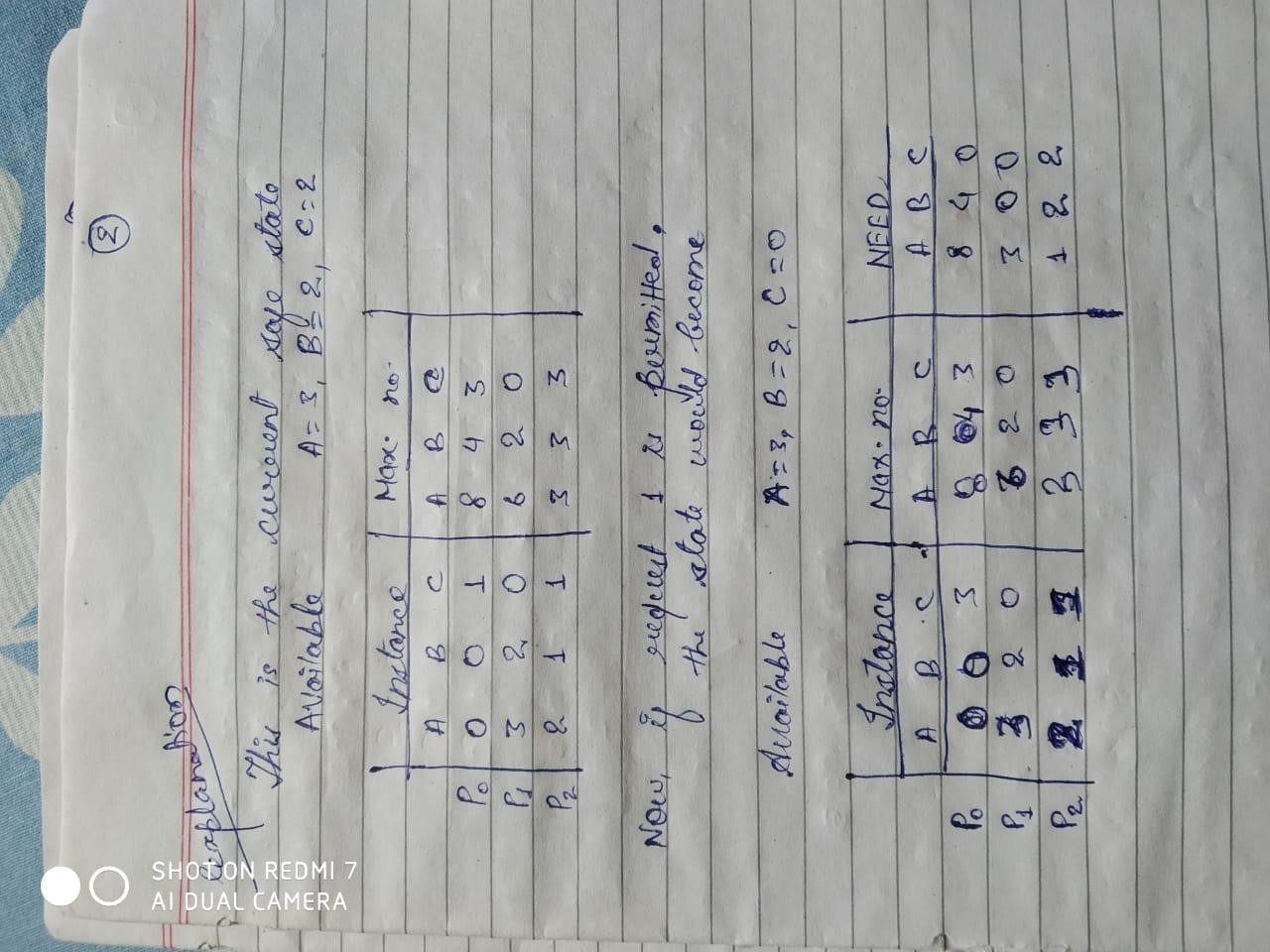
Suppose P0 has 0,0,1 instances , P1 is having 3,2,0 instances and P2 occupies 2,1,1 instances of A,B,C resource respectively.

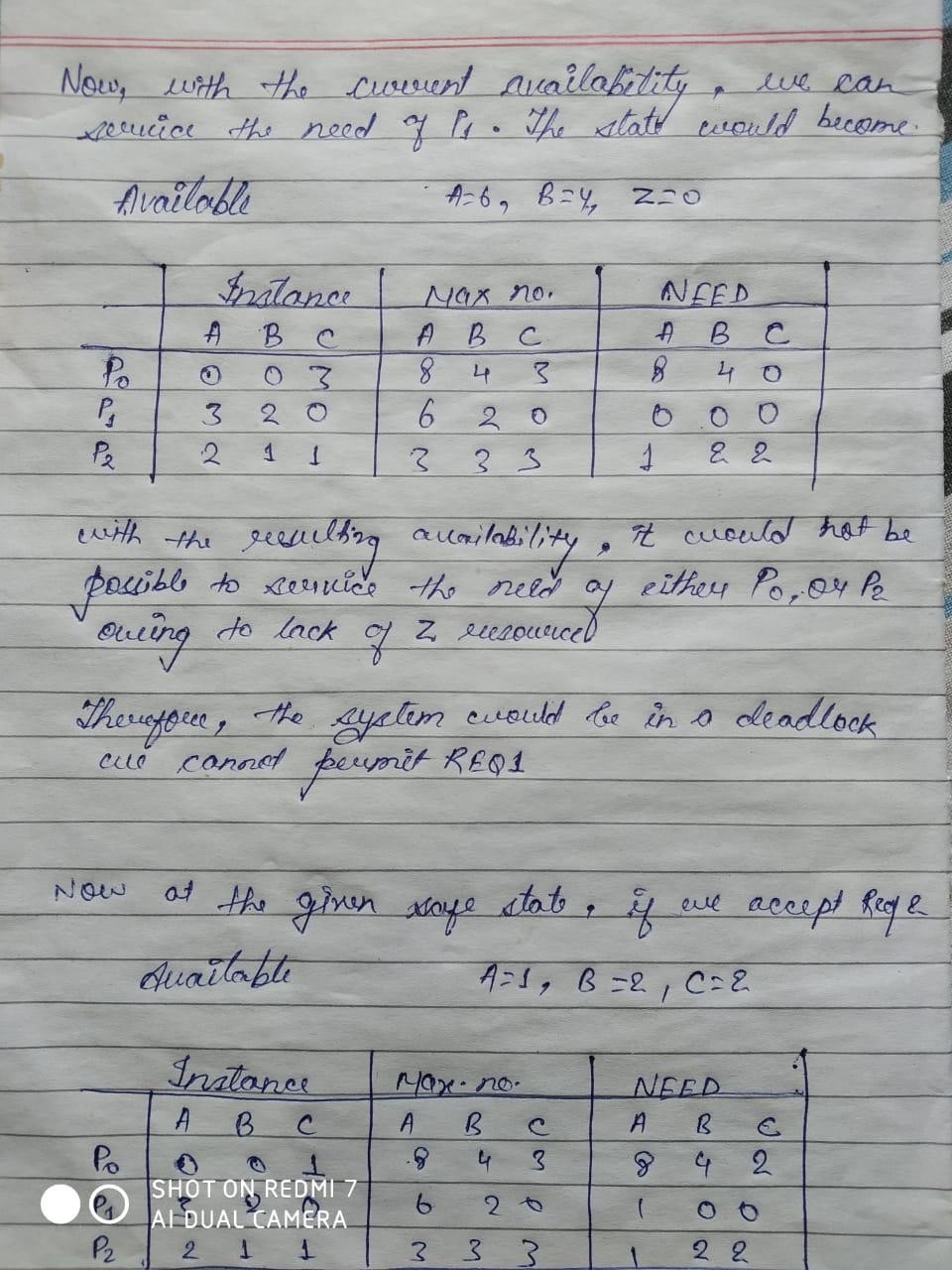
Also the maximum number of instances required for P0 is 8,4,3 and for p1 is 6,2,0 and finally for P2 there are 3,3,3 instances of resources A,B,C respectively. There are 3 instances of resource A, 2 instances of resource B and 2 instances of resource C available. Write a program to check whether Reena’s operating system is in a safe state or not in the following independent requests for additional resources in the

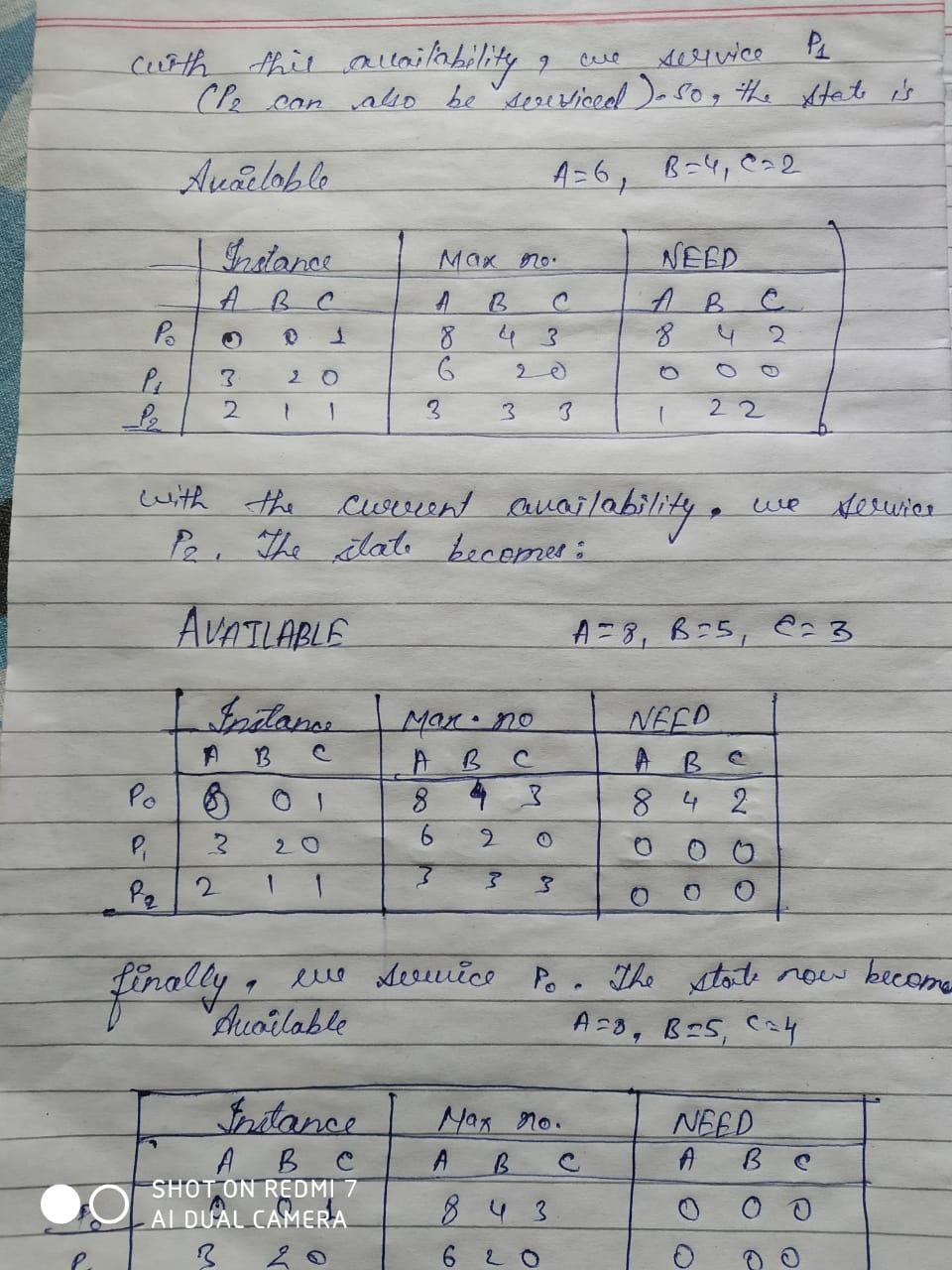
current state:

1. Request1: P0 requests 0 instances of A and 0 instances of B and 2 instances of C.
2. Request2: P1 requests for 2 instances of A, 0 instances of B and 0 instances of C.









The state so obtained is a safe state. ⇒ REQ2 can be permitted. So, only REQ2 can be permitted

**Banker's Algorithm**

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Everytime a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers.

Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

Let us assume that there are n processes and m resource types. Some data structures that are used to implement the banker's algorithm are:

### 1. Available

It is an array of length m. It represents the number of available resources of each type. If Available[j] = k, then there are k instances available, of resource type R(j).

2. Max

It is an n x m matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process P(i) can request at most k instances of resource type R(j).

### 3.Instance

It is an n x m matrix which represents the number of resources of each type currently allocated to each process. If Instance[i][j] = k, then process P(i) is currently allocated k instances of resource type R(j).

### **4. Need**

It is an n x m matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process P(i) may need k more instances of resource type R(j) to complete its task.

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

## **Safety Algorithm**

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,
2. Work = Available
3. Finish[i] =false for i = 0, 1, ... , n - 1.

This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

1. Find an index **i** such that both
2. Finish[i] ==false
3. Needi <= Work

If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.

1. Perform the following:
2. Work = Work +Instance;
3. Finish[i] = true;

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

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

That means if all processes are finished, then the system is in safe state.

$Code$

#include <assert.h> #include <memory.h> #include <stdbool.h> #include <stdio.h> #include <stdlib.h>

#ifdef VERBOSE\_ENABLED

#define LOG(...) fprintf(stderr, VA\_ARGS ) #else

#define LOG(...) \ do { \

} while (false); #endif

/\*

* This is a wrapper that will hold all the information about the current system
* state

\*

\*/

struct system\_state { int resource\_count; int process\_count; int\* avail\_resource; int\*\* allocation\_table; int\*\* max\_table;

} \* global\_system\_state, \*global\_transient\_state;

/\*

* This is a wrapper that holds the request information that is made

\*/

struct request { int process\_id;

int\* resource\_requests;

} \* request\_info;

/\*

* This function free all the dynamically allocated resources.

\*/

void free\_dynamic\_resource() {

for (int a = 0; a < global\_system\_state->process\_count; a++) { free(global\_system\_state->allocation\_table[a]); free(global\_system\_state->max\_table[a]); free(global\_transient\_state->max\_table[a]); free(global\_transient\_state->allocation\_table[a]);

}

free(global\_system\_state->allocation\_table); free(global\_transient\_state->allocation\_table); free(global\_system\_state->max\_table); free(global\_transient\_state->max\_table); free(global\_system\_state->avail\_resource); free(global\_transient\_state->avail\_resource); free(global\_system\_state); free(global\_transient\_state);

}

/\*\*

* This is function asks for the input of the user and allocates and fills the
* system state.

\* \*/

void input() {

LOG("\nStarted input function...");

global\_system\_state =

(struct system\_state\*)malloc(sizeof(struct system\_state));

global\_transient\_state =

(struct system\_state\*)malloc(sizeof(struct system\_state));

LOG("\nAllocated global\_system\_state variable...");

printf("Enter Total Process Count : ");

scanf("%d", &(global\_system\_state->process\_count));

global\_transient\_state->process\_count = global\_system\_state->process\_count; printf("\nEnter Total Resource Count : ");

scanf("%d", &(global\_system\_state->resource\_count));

global\_transient\_state->resource\_count = global\_system\_state->resource\_count; LOG("\nRead Process and resource counts...");

printf(

"\nEnter Allocated Resource Count as Table (Process in rows, and " "Resource in columns) : \n");

LOG("\nAllocating memory for allocation table...");

global\_system\_state->allocation\_table = (int\*\*)malloc(global\_system\_state->process\_count \* sizeof(int\*));

global\_transient\_state->allocation\_table = (int\*\*)malloc(global\_transient\_state->process\_count \* sizeof(int\*));

for (int a = 0; a < global\_system\_state->process\_count; a++) { global\_system\_state->allocation\_table[a] =

(int\*)malloc(global\_system\_state->resource\_count \* sizeof(int)); global\_transient\_state->allocation\_table[a] =

(int\*)malloc(global\_transient\_state->resource\_count \* sizeof(int));

}

LOG("\nAllocated allocation table memory now reading...");

// Read the allocation table

for (int a = 0; a < global\_system\_state->process\_count; a++)

for (int b = 0; b < global\_system\_state->resource\_count; b++) { scanf("%d", &(global\_system\_state->allocation\_table[a][b])); global\_transient\_state->allocation\_table[a][b] =

global\_system\_state->allocation\_table[a][b];

}

LOG("\nMax Table Allocating memory...");

// Allocate memory for Max resource table

global\_system\_state->max\_table = (int\*\*)malloc(global\_system\_state->process\_count \* sizeof(int\*));

global\_transient\_state->max\_table = (int\*\*)malloc(global\_transient\_state->process\_count \* sizeof(int\*));

for (int a = 0; a < global\_system\_state->process\_count; a++) { global\_system\_state->max\_table[a] =

(int\*)malloc(global\_system\_state->resource\_count \* sizeof(int));

global\_transient\_state->max\_table[a] = (int\*)malloc(global\_transient\_state->resource\_count \* sizeof(int));

}

LOG("\nAllocated max table memory now reading..."); printf(

"\nEnter Maximum Resource Count Limit as Table (Process in rows, and " "Resource limit in columns) : \n");

for (int a = 0; a < global\_system\_state->process\_count; a++)

for (int b = 0; b < global\_system\_state->resource\_count; b++) { scanf("%d", &(global\_system\_state->max\_table[a][b])); global\_transient\_state->max\_table[a][b] =

global\_system\_state->max\_table[a][b];

}

LOG("\nAllocating memory to avail\_resources...");

// Allocate memory for avail resource vector and list; global\_system\_state->avail\_resource =

(int\*)malloc(sizeof(int) \* global\_system\_state->resource\_count); global\_transient\_state->avail\_resource =

(int\*)malloc(sizeof(int) \* global\_transient\_state->resource\_count);

LOG("\nReading values to available\_resources...");

printf(

"\nEnter available resource count in the same order as above for each " "resource : \n");

for (int a = 0; a < global\_system\_state->resource\_count; a++) {

scanf("%d", &(global\_system\_state->avail\_resource[a])); global\_transient\_state->avail\_resource[a] =

global\_system\_state->avail\_resource[a];

}

};

/\*\*

* This is a function responsible for actually finding the stable state if found
* it sets the solution\_state with the state else sets it to NULL

\* \*/

/\*

These are some useful math functions for easing the solving task a : available;

b : allocated; c : required;

\*/

bool vec\_math\_is\_allocatable(int\* a, int\* b, int\* c, int len) { for (int iter = 0; iter < len; iter++)

if (a[iter] + b[iter] < c[iter]) return false; return true;

}

/\*

These are some useful math functions for easing the solving task a : available;

b : allocated; c : required;

\*/

void vec\_math\_allocate\_and\_free(int\* a, int\* b, int\* c, int len) {

assert(vec\_math\_is\_allocatable(a, b, c, len)); for (int iter = 0; iter < len; iter++) {

a[iter] += b[iter]; b[iter] = 0;

}

}

/\*

* a : allocated
* b : requested
* c : max\_limit

\*/

bool vec\_math\_should\_grant(int\* a, int\* b, int\* c, int len) { for (int t = 0; t < len; t++)

if (a[t] + b[t] > c[t]) return false; return true;

}

/\*

* This function restores the state to old state onces a request has been
* processed

\*/

void restore() {

LOG("\nRestoring Available Resource State to Global State...");

for (int a = 0; a < global\_system\_state->resource\_count; a++) global\_transient\_state->avail\_resource[a] =

global\_system\_state->avail\_resource[a];

LOG("\nResources Allocation Table Restored...");

for (int a = 0; a < global\_system\_state->process\_count; a++) for (int b = 0; b < global\_system\_state->resource\_count; b++) global\_transient\_state->allocation\_table[a][b] =

global\_system\_state->allocation\_table[a][b];

}

/\*

* This function solves the state after the request has been granted and returns
* true if system is in stable state or else false if deadlock is encountered

\*/

bool solve() {

LOG("\nStarted solving the system...");

int non\_executed = global\_system\_state->process\_count; bool dead\_lock = false;

bool has\_completed[global\_system\_state->process\_count];

for (int a = 0; a < global\_system\_state->process\_count; a++) has\_completed[a] = false;

while (non\_executed) { dead\_lock = true;

for (int a = 0; a < global\_system\_state->process\_count; a++) { if (has\_completed[a]) continue;

LOG("\nChecking if P%d can be allocated for execution...", a); bool res =

vec\_math\_is\_allocatable(global\_transient\_state->avail\_resource,

global\_transient\_state->allocation\_table[a], global\_transient\_state->max\_table[a], global\_transient\_state->resource\_count);

if (res) { has\_completed[a] = true; non\_executed--; dead\_lock = false;

LOG("\nAllocating and releasing resources for P%d", a);

vec\_math\_allocate\_and\_free(global\_transient\_state->avail\_resource, global\_transient\_state->allocation\_table[a], global\_transient\_state->max\_table[a], global\_transient\_state->resource\_count);

} else {

LOG("\nCannot satisfy needs for P%d. Skipping...", a);

}

}

if (dead\_lock) {

LOG("\nEncountered a deadlock..."); return false;

}

}

return true;

};

/\*

* This asks the request count from user.

\*/

void ask\_request\_count(int\* target) {

printf("\nHow many number of requests will arrive : "); scanf("%d", target);

}

/\*

* This asks the actual request and solves the state and prints if it is safe
* state or not after request has been granted

\*/

void ask\_requests(int n) { for (int t = 0; t < n; t++) {

printf("\nRequest %d : ", t + 1); int p\_c;

printf("\nEnter Process ID which request resources : "); scanf("%d", &p\_c);

p\_c--;

if (p\_c > global\_system\_state->process\_count) { printf("\nOpps !! No such PID found");

t--;

continue;

}

printf("\nEnter %d space separated integers each for each resource type : ", global\_system\_state->resource\_count);

request\_info = (struct request\*)malloc(sizeof(struct request)); request\_info->process\_id = p\_c;

request\_info->resource\_requests =

(int\*)malloc(sizeof(int) \* global\_system\_state->resource\_count);

for (int a = 0; a < global\_system\_state->resource\_count; a++) scanf("%d", &(request\_info->resource\_requests[a]));

if (vec\_math\_should\_grant(global\_system\_state->allocation\_table[p\_c], request\_info->resource\_requests, global\_system\_state->max\_table[p\_c], global\_system\_state->resource\_count)) {

bool flag = true;

for (int a = 0; a < global\_system\_state->resource\_count; a++) if (global\_transient\_state->avail\_resource[a] <

request\_info->resource\_requests[a]) { flag = false;

break;

}

if (flag)

printf("\nGranting the Resources. We have Enough resource to grant."); else

printf(

"\nPartially Granted Request. Limit increased in the allocation " "table");

for (int a = 0; a < global\_system\_state->resource\_count; a++) { if (flag)

global\_transient\_state->avail\_resource[a] -= request\_info->resource\_requests[a];

global\_transient\_state->allocation\_table[p\_c][a] += request\_info->resource\_requests[a];

}

if (!solve()) printf(

"\nDEADLOCK : After Request was Granted the system went into " "DEADLOCK");

else printf(

"\nSystem has a Stable State even after the resource requested was " "granted");

}

else { printf(

"\nRequest for the resources denied... (Requested more than limit)");

}

free(request\_info->resource\_requests); free(request\_info);

restore(); printf("\n");

}

}

//This is main driver program.

int main() { input();

if (solve())

printf("\nInitially system is in Safe State");

else {

printf("\nPanic Deadlock initially."); return 0;

}

restore(); int n;

ask\_request\_count(&n); ask\_requests(n); free\_dynamic\_resource(); return 0; }

.