For Course

Operating Systems

(CSE 316)

Registration No	11810826
Name of Student	Abhilasha Sharma
Roll No	34
Section	K18SP
Email ID:	Silkabhilasha4@gmail.com
GitHub Link:	https://github.com/AbhilashaSharma14/OS_11_Question

Contents

Serial No Name

Problem Description

Problem states that consider three allocation resources A,B,C to three processes P0,P1,and P2 and instance of each process will be given P0 has 0,0,1instances P1 has

3, 2, 0 instances and P2 has 2, 1, 1 instances also the maximum number of instances required is given and there total number of each allocation resources are given. Now we have to it is in safe state or unsafe means it deadlock or not.

So for that we use banker's algorithm also we call deadlock avoidance.

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]

$Allocation_i$ specifies the resources currently allocated to process P_i and
$Need_i$ specifies the additional resources that process P_i may still request to complete its task.

Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

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.

Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_i. When a request for resources is made by process P_i, the following actions are taken:

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
   $Allocation_i$ = $Allocation_i$ + $Request_i$
   $Need_i$ = $Need_i$– $Request_i$

Purpose of Use

Banker's Algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.

This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources. It also checks for all the possible activities before determining whether allocation should be continued or not.

For example, there are X number of account holders of a specific bank, and the total amount of money of their accounts is G.

When the bank processes a car loan, the software system subtracts the amount of loan granted for purchasing a car from the total money (G+ Fixed deposit + Monthly Income Scheme + Gold, etc.) that the bank has.

It also checks that the difference is more than or not G. It only processes the car loan when the bank has sufficient money even if all account holders withdraw the money G simultaneously.

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

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.

Summary:

- Banker's algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.
- Notations used in banker's algorithms are 1) Available 2) Max 3) Allocation 4) Need
- Resource request algorithm enables you to represent the system behavior when a specific process makes a resource request.
- Banker's algorithm keeps many resources that satisfy the requirement of at least one client
- The biggest drawback of banker's algorithm this that it does not allow the process to change its Maximum need while processing.

Code Snippet

```
#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;
}
```