Implementation of Banker's Algorithm

Student Details:

Name	Reg. No.	Roll No.	Section
Abhishek Aggarwal	200905246	43	A
Prajit Sivakumar	200905247	44	A

Abstract:

This project deals with the implementation of Banker's Algorithm.

Banker's Algorithm is a deadlock-avoidance algorithm used in a resource-allocation system with multiple instances of each resource type. This algorithm is used to avoid deadlock and allocate resources safely to different processes in an operating system. It simulates the allocation for predetermined maximum amounts of all resources to be allocated, checks for possible activities via an "S-State", and confirms if it is safe to continue allocation. Algorithms such as this are very crucial to the proper functioning of an operating system as a deadlock would mean that several processes may end up non-functional leading to an underwhelming experience for the user.

Hardware/Programming Languages Used:

- C Programming Language
- Ubuntu 20.04

Additional Information:

What is a Deadlock?

Processes in Operating Systems use resources in different ways:

- They request for a resource
- They use this requested resource
- They release the resource

Sometimes, we may end up in a situation where one process (Process A) needs access to a resource, but it is being used by another process (Process B). In this case, process A has to wait until process B is finished with its execution. However, what happens when process A has a resource process B wants AND process B has a resource process A wants to use? This leads to each process waiting for the other one indefinitely and leads to a situation called deadlock.

A deadlock can arise if the following conditions are met:

1. Mutual Exclusion: At least one resource is non shareable. (only one process can use it at a time)

2. Hold and Wait: A process is holding a resource AND is waiting for resources.

3. No Preemption: Resources cannot be obtained until another process releases them.

4. Circular Wait: Processes are waiting for each other in a circular form to release resources.

There are many ways in which deadlocks are handled across different systems:

1) Ignore the problem altogether: In some systems such as Windows and UNIX,

deadlocks are so rare that they can be ignored. If it still happens, the system reboots.

2) Deadlock detection and recovery: Once a deadlock has occurred, preemption is used to handle it.

3) Deadlock prevention or avoidance: Do not let the system into a deadlock state. Here, a deadlock is avoided by removing one of the above conditions.

One such way of handling this is **banker's algorithm**.

Banker's Algorithm:

It is a deadlock avoidance algorithm. This algorithm is named so because it is used in banking systems to determine whether a loan can be granted.

Let us assume that there are N bankers and the sum of their money stored in the bank is S. Every time a loan is requested, the money to be given is subtracted from the total money that the bank has. This is the only way the bank would be able to grant loans while still being able to let all account holders withdraw ALL their money at once (worst case).

In computers, banker's algorithm works in a similar way. Each process must declare the maximum instance of each resource it needs.

Some features of banker's algorithm:

1. If a process wants to request for a resource and it is not available, it has to wait.

2. In every system, the number of resources is limited.

3. Banker's algorithm goes for maximum resource allocation.

4. Whenever a process gets all its resources, it needs to return them in a restricted period.

5. Multiple resources are maintained that can fulfill the needs of at least one client.

Some data structures used:

1) Resources array: It represents the number of available resources of each type.

2) maxRequired matrix: Maximum number of instances of each resource that a process can request. If maxRequired[i][j] = k, then a process i can request atmost k instances of resource type j.

3) Allocation matrix: The number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process i is currently allocated k instances of resource type j.

4) Need matrix: Remaining resource needs of each process. If Need[i][j] = k, then process i needs k MORE instances of resource type j.

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

Code Implementation in C:

```c
// A Multithreaded Program that implements the banker's algorithm.
#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 *safe_sequence;
int nProcessRan = 0;

pthread_mutex_t lockResources;
pthread_cond_t condition;

// get safe sequence is there is one else return false
bool get_safe_sequence();

// final output after decision
void* display_result(void*);

int main(int argc, char** argv) {
    // iterators
    int i, j;

    // Get number of processes
    do {
        printf("\nEnter the number of processes: ");
        scanf("%d", &nProcesses);
        // can't have 0 or negative processes
        // More than 5 becomes too many
        if (nProcesses < 1 || nProcesses > 5) {
            printf("Range for number of processes: 1 - 5.\n");
        }
    } while (nProcesses < 1 || nProcesses > 5); // stay realistic


    // Get number of resources
    do {
        printf("\nEnter the number of resources: ");
        scanf("%d", &nResources);
        // can't have 0 or negative resources
        // More than 5 becomes too many
        if (nResources < 1 || nResources > 5) {
            printf("Range for number of resources: 1 - 5.\n");
        }
```

os.c

```
        }
    } while (nResources < 1 || nResources > 5); // stay realistic


    // The next bits of code deal with memory assignment

    resources = (int*)malloc(nResources * sizeof(*resources));
    if (!resources) {
        printf("Failed to allocate memory to resources, exiting program...\n");
        exit(-1);
    }
    printf("\nEnter currently Available resources: ");
    for(i = 0; i < nResources; ++i) {
        scanf("%d", &resources[i]);
    }
    allocated = (int**)malloc(nProcesses * sizeof(*allocated));
    if (!allocated) {
        printf("Failed to allocate memory to allocated, exiting program...\n");
        exit(-1);
    }
    for(i = 0; i < nProcesses; ++i) {
        allocated[i] = (int*)malloc(nResources * sizeof(**allocated));
        if (!allocated[i]) {
        printf("Failed to allocate memory to allocated[%d], exiting program...\n", i);
        exit(-1);
        }
    }
    maxRequired = (int**)malloc(nProcesses * sizeof(*maxRequired));
    if (!maxRequired) {
        printf("Failed to allocate memory to maxRequired, exiting program...\n");
        exit(-1);
    }
    for(i = 0; i < nProcesses; ++i) {
        maxRequired[i] = (int*)malloc(nResources * sizeof(**maxRequired));
        if (!maxRequired[i]) {
            printf("Failed to allocate memory to maxRequired[%d], exiting program...\n", i);
        exit(-1);
        }
    }

    // allocated

    printf("\n");
    for(i = 0; i < nProcesses; ++i) {
        printf("\nHow much resource has to be allocated to process %d: ", i + 1);
        for(j = 0; j < nResources; ++j) {
            scanf("%d", &allocated[i][j]);
        }
```

os.c

```c
    }
    printf("\n");

    // maximum required resources

    for(i = 0; i < nProcesses; ++i) {
        printf("\nWhat is the maximum resource required by process %d: ", i + 1);
            for(j = 0; j < nResources; ++j) {
                scanf("%d", &maxRequired[i][j]);
            }
    }
    printf("\n");

    // calculate need matrix

    need = (int**)malloc(nProcesses * sizeof(*need));
    if (!need) {
        printf("Failed to allocate memory to need, exiting program...\n");
        exit(-1);
    }
    for(i = 0; i < nProcesses; ++i) {
        need[i] = (int*)malloc(nResources * sizeof(**need));
        if (!need[i]) {
            printf("Failed to allocate memory to need[%d], exiting program...\n",i);
            exit(-1);
        }
    }
    for(i = 0; i < nProcesses; ++i) {
        for(j = 0; j < nResources; ++j) {
            need[i][j] = maxRequired[i][j] - allocated[i][j];
        }
    }
    safe_sequence = (int*)malloc(nProcesses * sizeof(*safe_sequence));
    if (!safe_sequence) {
        printf("Failed to allocate memory to safe_sequence, exiting program...\n");
        exit(-1);
    }

    // set default values

    for(i = 0; i < nProcesses; ++i) {
        safe_sequence[i] = -1;
    }

    // attempt to get a safe sequence

    if(!get_safe_sequence()) {
        printf("\nUnsafe! The processes leads the system to a unsafe state.\n\n");
```

os.c

```c
        exit(-1);
    }
    printf("\n\nSafe Sequence Found: ");
    for(i = 0; i < nProcesses; ++i) {
        printf("%-3d", safe_sequence[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(i = 0; i < nProcesses; ++i) {
        processNumber[i] = i;
    }
    for(i = 0; i < nProcesses; ++i) {
        pthread_create(&processes[i], &attr, display_result,(void*)(&processNumber[i]));
    }
    for(i = 0; i < nProcesses; ++i) {
        pthread_join(processes[i], NULL);
    }
    printf("\nAll processes have finished executing.\n");// free resources
    printf("\nFreeing assigned memory.\n");
    free(resources);
    for(i = 0; i < nProcesses; ++i) {
        free(allocated[i]);
        free(maxRequired[i]);
        free(need[i]);
    }
    free(allocated);
    free(maxRequired);
    free(need);
    free(safe_sequence);
    // end
}
bool get_safe_sequence() {

    // iterators

    int i, j, k;

    // get safe sequence

    int tempRes[nResources];
    for(i = 0; i < nResources; ++i) {
```

os.c

```
        tempRes[i] = resources[i];
    }
    bool finished[nProcesses];
    for(i = 0; i < nProcesses; ++i) {
        finished[i] = false;
    }
    int nfinished = 0;
    while(nfinished < nProcesses) {
        bool safe = false;
        for(i = 0; i < nProcesses; ++i) {
            if(!finished[i]) {
                bool possible = true;
                for(j = 0; j < nResources; ++j) {

                    // Required resources not available

                        if(need[i][j] > tempRes[j]) {
                            possible = false;
                            break;
                        }
                }
                if(possible) {
                    for(j = 0; j < nResources; ++j) {
                    tempRes[j] += allocated[i][j];
                }
                safe_sequence[nfinished] = i;
                finished[i] = true;
                ++nfinished;

                // All must be safe to get a safe sequence

                safe = true;
                }
            }
        }
        if(!safe) {
            for(k= 0; k < nProcesses; ++k) {
                safe_sequence[k] = -1;
            }

            // no safe sequence found

            return false;
        }
    }

    // safe sequence found

```

```c
    return true;
}

// process code
void* display_result(void *arg) {

    // iterator
    int i;
    int p = *((int*)arg);

    // lock resources
    // we should not allow access to prevent mismatch

    pthread_mutex_lock(&lockResources);

    // condition check

    while(p != safe_sequence[nProcessRan])
        pthread_cond_wait(&condition, &lockResources);

        // process

    printf("\nNow running Process %d....", p + 1);
    printf("\n\tAllocated: ");
    for(i = 0; i < nResources; ++i){
        printf("%3d", allocated[p][i]);
    }
    printf("\n\tNeeded: ");
    for(i = 0; i < nResources; ++i) {
        printf("%3d", need[p][i]);
    }
    printf("\n\tAvailable: ");
    for(i = 0; i < nResources; ++i) {
        printf("%3d", resources[i]);
    }

    // Sleep for aesthetics

    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);
```

```c
    printf("\tResource Released!");
    for(i = 0; i < nResources; ++i) {
        resources[i] += allocated[p][i];
    }
    printf("\n\tNow Available: ");
    for(i = 0; i < nResources; ++i) {
        printf("%3d", resources[i]);
    }
    printf("\n\n");

    // Sleep for aesthetics

    sleep(1);
    nProcessRan++;
    pthread_cond_broadcast(&condition);
    pthread_mutex_unlock(&lockResources);
    pthread_exit(NULL);
}
```

Testcases:

Pass 1:

```
acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$ ./t1

Enter the number of processes: 5

Enter the number of resources: 3

Enter currently Available resources: 3 3 2

How much resource has to be allocated to process 1: 0 1 0

How much resource has to be allocated to process 2: 2 0 0

How much resource has to be allocated to process 3: 3 0 2

How much resource has to be allocated to process 4: 2 1 1

How much resource has to be allocated to process 5: 0 0 2

What is the maximum resource required by process 1: 7 5 3

What is the maximum resource required by process 2: 3 2 2

What is the maximum resource required by process 3: 9 0 2

What is the maximum resource required by process 4: 4 2 2

What is the maximum resource required by process 5: 5 3 3


Safe Sequence Found: 2  4  5  1  3
Executing Processes...
```

```
Safe Sequence Found: 2  4  5  1  3
Executing Processes...


Now running Process 2....
        Allocated:   2  0  0
        Needed:   1  2  2
        Available:   3  3  2
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   5  3  2


Now running Process 4....
        Allocated:   2  1  1
        Needed:   2  1  1
        Available:   5  3  2
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   7  4  3


Now running Process 5....
        Allocated:   0  0  2
        Needed:   5  3  1
        Available:   7  4  3
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   7  4  5
```

```
Now running Process 1....
        Allocated:   0  1  0
        Needed:   7  4  3
        Available:   7  4  5
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   7  5  5


Now running Process 3....
        Allocated:   3  0  2
        Needed:   6  0  0
        Available:   7  5  5
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:  10  5  7


All processes have finished executing.

Freeing assigned memory.
acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$
```

Pass 2 :

```
acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$ ./t1

Enter the number of processes: 5

Enter the number of resources: 4

Enter currently Available resources: 1 5 2 0


How much resource has to be allocated to process 1: 0 0 1 2

How much resource has to be allocated to process 2: 1 0 0 0

How much resource has to be allocated to process 3: 1 3 5 4

How much resource has to be allocated to process 4: 0 6 3 2

How much resource has to be allocated to process 5: 0 0 1 4


What is the maximum resource required by process 1: 0 0 1 2

What is the maximum resource required by process 2: 1 7 5 0

What is the maximum resource required by process 3: 2 3 5 6

What is the maximum resource required by process 4: 0 6 5 2

What is the maximum resource required by process 5: 0 6 5 6



Safe Sequence Found: 1  3  4  5  2
Executing Processes...
```

```
Safe Sequence Found: 1  3  4  5  2
Executing Processes...


Now running Process 1....
        Allocated:   0  0  1  2
        Needed:   0  0  0  0
        Available:   1  5  2  0
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   1  5  3  2


Now running Process 3....
        Allocated:   1  3  5  4
        Needed:   1  0  0  2
        Available:   1  5  3  2
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   2  8  8  6


Now running Process 4....
        Allocated:   0  6  3  2
        Needed:   0  0  2  0
        Available:   2  8  8  6
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   2 14 11  8
```

```
Now running Process 5....
        Allocated:   0  0  1  4
        Needed:   0  6  4  2
        Available:   2 14 11  8
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   2 14 12 12


Now running Process 2....
        Allocated:   1  0  0  0
        Needed:   0  7  5  0
        Available:   2 14 12 12
        Resource Allocated!
        Process Code Running...
        Process Code Completed...
        Process Releasing Resource...
        Resource Released!
        Now Available:   3 14 12 12


All processes have finished executing.

Freeing assigned memory.
acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$
```

For the above cases, we can see that resource requirements are sufficient and can be provided for by the OS.

When the needed maximum resource allocation cannot be provided by the OS because the resource doesn't exist, then we cannot let the process continue as the system would be in an unsafe state.

Fail 1:

```
acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$ ./t1

Enter the number of processes: 5

Enter the number of resources: 4

Enter currently Available resources: 3 3 0 1

How much resource has to be allocated to process 1: 2 0 0 1

How much resource has to be allocated to process 2: 3 1 2 1

How much resource has to be allocated to process 3: 2 1 0 3

How much resource has to be allocated to process 4: 1 3 1 2

How much resource has to be allocated to process 5: 1 4 5 2

What is the maximum resource required by process 1: 4 2 1 2

What is the maximum resource required by process 2: 5 2 5 2

What is the maximum resource required by process 3: 2 3 1 6

What is the maximum resource required by process 4: 1 4 2 4

What is the maximum resource required by process 5: 3 6 6 5

Unsafe! The processes leads the system to a unsafe state.

acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$
```

In this case, we don't have any availability of the 3rd resource. Thus, after constructing the Need Matrix it is found that no process can be completed. Hence it is unsafe state.

Fail 2:

```
acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$ ./t1

Enter the number of processes: 5

Enter the number of resources: 3

Enter currently Available resources: 2 2 3

How much resource has to be allocated to process 1: 0 0 1

How much resource has to be allocated to process 2: 2 0 3

How much resource has to be allocated to process 3: 0 2 0

How much resource has to be allocated to process 4: 1 2 1

How much resource has to be allocated to process 5: 0 0 2

What is the maximum resource required by process 1: 4 8 2

What is the maximum resource required by process 2: 9 2 3

What is the maximum resource required by process 3: 0 2 0

What is the maximum resource required by process 4: 2 2 2

What is the maximum resource required by process 5: 0 0 2

Unsafe! The processes leads the system to a unsafe state.

acer@acer-Aspire-A715-75G:~/200905247/OSL/MiniProject$
```

In this case, the sum of available and allocated resources for resources 1 and 2 are not enough to satisfy need of Process 2 and 1 in each case, ie. available[i] + allocated[i] (sum of all resources) < Max[i][j] (for specific process). This situation is further explained below:

For resource 1, [2] + [0 + 2 + 0 + 1 + 0] = 5 < 9, so Process 2 cannot be satisfied.

For resource 2, [2] + [0 + 0 + 2 + 2 + 0] = 6 < 8, so Process 1 cannot be satisfied.

For resource 3, [3] + [1 + 3 + 0 + 1 + 2] = 10 > all the process requirements, and can be satisfied by the OS.

Contribution:

Prajit Sivakumar: Code Implementation, Additional Information, Test Cases.

Abhishek Aggarwal: Code Implementation, Future Scope, Limitations, Test Cases.

Learning Outcome:

Some limitations of this algorithm observed are as follows:

1. While processing is done, the algorithm does not permit change in maximum need of a process.
2. All processes must know the maximum resource needs of every process in advance.
3. All requests may be granted in a finite time, but this period is fixed at one year.
4. The number of processes must be fixed, no additional processes may start while it is executing.
5. All processes guarantee that resources loaned will be repaid in some finite time. This prevents starvation but resource hungry processes may still develop.

Future Scope:

Banker's algorithm can be applied in the universities, by designing a reasonable course scheduling system, thus avoiding the situation of deadlock and rationally utilizes the resources of the school. The algorithm can ensure that all customers are satisfied within a limited time.

Banker's algorithm can also be used for conflict resolution in multivehicle traffic systems, where a number of mobile agents move freely in a finite area, with each agent following a prespecified-motion profile. Using real-time management of sequential resource allocation systems, a protocol can be developed that can formally guarantee the safe and live operation of the underlying traffic system, while they remain scalable with respect to the number of moving agents.

Banker's algorithm is also applicable to buffer space allocation in flexible manufacturing. Banker's approach can provide very good operational flexibility when properly applied to the manufacturing environment.

References:

The websites and resources referred are given below:

1. https://www.wikipedia.org/

2. https://www.geeksforgeeks.org/

3. Operating System Concepts, 9th Edition (Abraham Silberschatz, Peter B. Galvin, Greg Gagne)