Name-Abhi Mehta
Reg No.-221080001

Experiment-06

Aim: Study and Implement Bankers Algorithm for Deadlock and implement algorithm for deadlock Detection in C programming. Implement Programming Projects: Banker's Algorithm on page 345 in Galvin,9th edition.

Theory: The Banker's Algorithm, originally devised for deadlock avoidance, serves as a crucial tool not only in preventing deadlocks but also indirectly in detecting potential deadlock situations. By meticulously managing resource allocation, the algorithm ensures that the system remains in a state that is immune to deadlocks. It achieves this by calculating a safe sequence of processes that can execute without triggering a deadlock. If such a sequence can be determined, the system is deemed safe; however, if such a sequence cannot be found, it signals an unsafe state, hinting at a possible deadlock scenario. Therefore, while the primary objective of the Banker's Algorithm is to prevent deadlocks by proactively avoiding unsafe states, its inability to establish a safe sequence for resource allocation serves as an early warning sign for potential deadlock occurrences. This dual functionality of the Banker's Algorithm not only enhances its significance in maintaining system stability but also underscores its importance in detecting conditions that could potentially lead to deadlocks, thereby elevating the overall reliability and robustness of concurrent systems.

Implementation:

C os6.c > request_resources(int, int [])

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<time.h>
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 3

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];

/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the remaining need of each customer */
// need = maximum - allocation
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

// we use mutex to prevent
pthread_mutex_t mutex;

//set maximum demand/
void setMaximum(int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]);

//set allocation/
void setAllocation(int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]);

//calculate Need/
void calculateNeed();

//calculate available/
void calculateAvailable(int temp[NUMBER_OF_RESOURCES]);

//print out the result/
void print();

//check if the request is safe/
int safetyCheck();


int request_resources(int customer_num, int request[]);

int release_resources(int customer_num, int release[]);
```

os6.c > ...

```c
//prevent race conditions/
void* thread_control(void* arg);

int main(int argc, char const *argv[]) {
    int i;

    // the available resources
    int temp[NUMBER_OF_RESOURCES];

    // the thread id
    pthread_t tid[NUMBER_OF_CUSTOMERS];

    // initialize mutex
    pthread_mutex_init(&mutex,NULL);

    // initialize the available resources
    for(i=0;i<argc-1;i++)
    {
        // because arg 0 is the name of the program
        // we start from arg 1
        temp[i] = atoi(argv[i+1]);
    }

    // set the maximum demand
    setAllocation(allocation);
    setMaximum(maximum);
    calculateNeed();
    calculateAvailable(temp);
    print();
    //create thread/
    for (i=0; i<NUMBER_OF_CUSTOMERS; i++)
    {
        // init the thread
        // syntax: pthread_create(&thread_id, NULL, function_name, argument)
        pthread_create(&tid[i],NULL,thread_control,(void*)(size_t)i);
    }
    /*join thread */
    for (i=0; i<NUMBER_OF_CUSTOMERS; i++)
    {
        // join the thread to the main thread
        // syntax: pthread_join(thread_id, NULL)
        pthread_join(tid[i],NULL);
    }
    return 0;
}
```

os6.c > ...

```c
void setMaximum(int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]) {
    printf("Please enter the maximum:\n");
    int i, j;
    for (i = 0; i < NUMBER_OF_CUSTOMERS; i++) {
        for (j = 0; j < NUMBER_OF_RESOURCES; j++) {
            scanf("%d", &maximum[i][j]);
        }
        if (i < NUMBER_OF_CUSTOMERS - 1) {
            printf("Enter the maximum for the next process:\n"); // for the next process
        }
    }
}

void setAllocation(int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]) {
    printf("Please enter the allocation:\n");
    int i, j;
    for (i = 0; i < NUMBER_OF_CUSTOMERS; i++) {
        for (j = 0; j < NUMBER_OF_RESOURCES; j++) {
            scanf("%d", &allocation[i][j]);
        }
        if (i < NUMBER_OF_CUSTOMERS - 1) {
            printf("Enter the allocation for the next process:\n"); // for the next process
        }
    }
}

void calculateNeed()
{
    int i,j;
    for (i=0;i<NUMBER_OF_CUSTOMERS;i++) {
        for (j=0; j<NUMBER_OF_RESOURCES;j++) {
            // we know that need = maximum - allocation
            need[i][j] = maximum[i][j]-allocation[i][j];
        }
    }
}

void calculateAvailable(int temp[NUMBER_OF_RESOURCES])
{
    int i,j;
    // calculate the sum of each column //
    // declare an array to store the sum of each column //
    int sum[NUMBER_OF_RESOURCES]={0};

    // calculate the sum of each column //
    for (i=0;i<NUMBER_OF_RESOURCES;i++) {
        for (j=0; j<NUMBER_OF_CUSTOMERS;j++) {
```

os6.c > ...

```c
            sum[i]+=allocation[j][i];
        }

    }

    // AVAILABLE = INITIAL - ALLOCATED //
    for (i=0;i<NUMBER_OF_RESOURCES; i++) {
        available[i]=temp[i]-sum[i];
    }
}

void print()
{
    int i,j;
    printf("-----Allocation-----Maximum-----\n");

    for (i=0;i<NUMBER_OF_CUSTOMERS;i++) {
        for (j=0;j<NUMBER_OF_RESOURCES;j++) {
            printf(" %2d",allocation[i][j]);
        }
        printf("     ");
        for (j=0;j<NUMBER_OF_RESOURCES;j++) {
            printf("%2d ",maximum[i][j]);
        }
    printf("\n");
    }
    printf("-----Available-----\n");
    for (i=0; i<NUMBER_OF_RESOURCES; i++) {
        printf(" %2d",available[i]);
    }
    printf("\n");
    printf("-----Need-----\n");
    for (i=0; i<NUMBER_OF_CUSTOMERS; i++) {
        for (j=0; j<NUMBER_OF_RESOURCES; j++) {
            printf(" %2d",need[i][j]);
        }
        printf("\n");
    }
}

int safetyCheck()
{
    int i,j,t;
    int result=1;
    // work = available
    int work[NUMBER_OF_RESOURCES];

```

C os6.c > ...

```c

    // array to count the number of resources that can be allocated
    int flag[NUMBER_OF_CUSTOMERS]={0};

    // finish is an array to store the finish status of each process
    int finish[NUMBER_OF_CUSTOMERS] ={0};

    // initialize work
    for (i=0; i<NUMBER_OF_RESOURCES;i++) {
        work[i]=available[i];
    }
    for (t=0; t<NUMBER_OF_CUSTOMERS; t++) {
        for (i=0;i<NUMBER_OF_CUSTOMERS;i++) {
            if(finish[i]!=1)
            {
                for (j=0;(j<NUMBER_OF_RESOURCES);j++) {
                    if(need[i][j]>work[j])
                    {
                        break;
                    }
                    else
                        flag[i]++;


                }
                if(flag[i]==NUMBER_OF_RESOURCES)
                {
                    printf("work is: ");
                    for (j=0; j<NUMBER_OF_RESOURCES; j++) {
                        work[j]+=allocation[i][j];
                        printf(" %d ",work[j]);
                    }
                    printf("\n");
                    finish[i]=1;
                    printf("Process %d is finished!\n",i);
                }
            }
        }
    }
    for (i=0;i<NUMBER_OF_CUSTOMERS;i++) {
        if(finish[i]==0)
        {
            result=0;
            printf("It is unsafe!\n");
            break;
        }
    }
```

os6.c > ...

```c
    return result;
}

int request_resources(int customer_num, int request[])
{
    int j,flag;
    flag=0;
    for (j=0; j<NUMBER_OF_RESOURCES; j++) {
        if(request[j]>need[customer_num][j])
        {
            printf("Error! Process has exceeded its maximum claim! Request Denied.\n");
            flag = -1;
            break;
        }
    }
    if (flag == 0) {
        for (j=0;j<NUMBER_OF_RESOURCES;j++) {
            if(request[j]>available[j])
            {
                printf("Resources are not available now,P(%d)must wait.\n",customer_num);
                flag = -1;
            }
        }
        if(flag==0){
            for (j=0; j<NUMBER_OF_RESOURCES; j++) {
                available[j]-=request[j];
                allocation[customer_num][j]+=request[j];
                need[customer_num][j]-=request[j];
            }
            if(safetyCheck()==1)
                printf("It is safe!Resources have been allocated.\n");
            else
            {
                printf("It is unsafe!P(%d) has to wait.\n",customer_num);
                for (j=0; j<NUMBER_OF_RESOURCES; j++) {
                    available[j]+=request[j];
                    allocation[customer_num][j]-=request[j];
                    need[customer_num][j]+=request[j];
                }
                flag=-1;
            }

        }
    }
    return flag;
}

```

os6.c > ...

```c
int release_resources(int customer_num, int release[])
{
    int j,flag;
    flag=0;
    for (j=0; j<NUMBER_OF_RESOURCES; j++) {
        if(allocation[customer_num][j]<release[j])
        {
            printf("Error!P(%d)don't have that many resources to release.\n",customer_num);
            flag = -1;
            break;
        }
    }
    for (j=0;j<NUMBER_OF_RESOURCES;j++) {
        if (flag==0) {
            available[j]+=release[j];
            allocation[customer_num][j]-=release[j];
            need[customer_num][j]+=release[j];
        }
    }
    if (flag==0) {
        printf("Yes! Resources have been released.\n");
    }
    else
    {
        printf("Error! Resources cannot be released.\n");
    }
    return flag;

}

void* thread_control(void* arg)
{
    int i = (int)(size_t)arg;
    int j;
    srand((unsigned)time(NULL));
    int lock_ret = 1;
    int request[NUMBER_OF_RESOURCES]={0, 1, 0};
    int release[NUMBER_OF_RESOURCES]={0, 1, 0};
    lock_ret = pthread_mutex_lock(&mutex);
    do{
        if (lock_ret) {
            printf("lock process %d failed...\n",i);
        }
        else
        {
            printf("lock process %d success!\n",i);
        }
    }while (lock_ret);

    printf("process %d request:\n",i);
    for (j=0; j<NUMBER_OF_RESOURCES; j++) {
        printf("%d ",request[j]);
    }
    printf("\n");
    request_resources(i,request);
    print();
    printf("process %d release:\n",i);
    for (j=0; j<NUMBER_OF_RESOURCES; j++) {
        printf("%d ",release[j]);
    }
    printf("\n");
    release_resources(i,release);
    lock_ret = pthread_mutex_unlock(&mutex);
    printf("Unlock process %d success!\n\n",i);
    return NULL;
}
```

Output:

```
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~$ cd coc
abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~/coc$ ./os6 10 6 9
Please enter the allocation:
0 1 0
Enter the allocation for the next process:
2 0 0
Enter the allocation for the next process:
3 0 2
Enter the allocation for the next process:
2 1 1
Enter the allocation for the next process:
0 0 2
Please enter the maximum:
7 5 3
Enter the maximum for the next process:
3 2 2
Enter the maximum for the next process:
9 0 2
Enter the maximum for the next process:
4 2 2
Enter the maximum for the next process:
5 3 3
-----Allocation-----Maximum-----
  0  1  0      7  5  3
  2  0  0      3  2  2
  3  0  2      9  0  2
  2  1  1      4  2  2
  0  0  2      5  3  3
-----Available-----
  3  4  4
-----Need-----
  7  4  3
  1  2  2
  6  0  0
  2  1  1
  5  3  1
```

```
lock process 0 success!
process 0 request:
0 1 0
work is:  5  3  4
Process 1 is finished!
work is:  7  4  5
Process 3 is finished!
work is:  7  4  7
Process 4 is finished!
work is:  7  6  7
Process 0 is finished!
work is:  10  6  9
Process 2 is finished!
It is safe!Resources have been allocated.
-----Allocation-----Maximum-----
  0  2  0      7  5  3
  2  0  0      3  2  2
  3  0  2      9  0  2
  2  1  1      4  2  2
  0  0  2      5  3  3
-----Available-----
  3  3  4
-----Need-----
  7  3  3
  1  2  2
  6  0  0
  2  1  1
  5  3  1
process 0 release:
0 1 0
Yes! Resources have been released.
Unlock process 0 success!

lock process 4 success!
process 4 request:
0 1 0
work is:  5  3  4
Process 1 is finished!
work is:  7  4  5
Process 3 is finished!
work is:  7  5  7
Process 4 is finished!
work is:  7  6  7
Process 0 is finished!
work is:  10  6  9
Process 2 is finished!
```

```
It is safe!Resources have been allocated.
-----Allocation-----Maximum-----
  0  1  0      7  5  3
  2  0  0      3  2  2
  3  0  2      9  0  2
  2  1  1      4  2  2
  0  1  2      5  3  3
-----Available-----
  3  3  4
-----Need-----
  7  4  3
  1  2  2
  6  0  0
  2  1  1
  5  2  1
process 4 release:
0 1 0
Yes! Resources have been released.
Unlock process 4 success!

lock process 2 success!
process 2 request:
0 1 0
Error! Process has exceeded its maximum claim! Request Denied.
-----Allocation-----Maximum-----
  0  1  0      7  5  3
  2  0  0      3  2  2
  3  0  2      9  0  2
  2  1  1      4  2  2
  0  0  2      5  3  3
-----Available-----
  3  4  4
-----Need-----
  7  4  3
  1  2  2
  6  0  0
  2  1  1
  5  3  1
process 2 release:
0 1 0
Error!P(2)don't have that many resources to release.
Error! Resources cannot be released.
Unlock process 2 success!
```

```
lock process 3 success!
process 3 request:
0 1 0
work is:  5  3  4
Process 1 is finished!
work is:  7  5  5
Process 3 is finished!
work is:  7  5  7
Process 4 is finished!
work is:  7  6  7
Process 0 is finished!
work is:  10  6  9
Process 2 is finished!
It is safe!Resources have been allocated.
-----Allocation-----Maximum-----
  0  1  0      7  5  3
  2  0  0      3  2  2
  3  0  2      9  0  2
  2  2  1      4  2  2
  0  0  2      5  3  3
-----Available-----
  3  3  4
-----Need-----
  7  4  3
  1  2  2
  6  0  0
  2  0  1
  5  3  1
process 3 release:
0 1 0
Yes! Resources have been released.
Unlock process 3 success!

lock process 1 success!
process 1 request:
0 1 0
work is:  5  4  4
Process 1 is finished!
work is:  7  5  5
Process 3 is finished!
work is:  7  5  7
Process 4 is finished!
work is:  7  6  7
Process 0 is finished!
```

```
work is:  10  6  9
Process 2 is finished!
It is safe!Resources have been allocated.
-----Allocation-----Maximum-----
  0  1  0      7  5  3
  2  1  0      3  2  2
  3  0  2      9  0  2
  2  1  1      4  2  2
  0  0  2      5  3  3
-----Available-----
  3  3  4
-----Need-----
  7  4  3
  1  1  2
  6  0  0
  2  1  1
  5  3  1
process 1 release:
0 1 0
Yes! Resources have been released.
Unlock process 1 success!

abhi@abhi-VivoBook-ASUSLaptop-M3400QA-M3400QA:~/coc$
```

- **Conclusion:**

  To summarize, although the Banker's Algorithm primarily functions as a deadlock avoidance mechanism by ensuring the system remains in a safe state, its fundamental principles indirectly aid in detecting potential deadlocks. Through meticulous management of resource allocation and validation of a safe execution sequence, the algorithm offers a mechanism to recognize scenarios where the system might be vulnerable to deadlock occurrence. Through its proactive resource management and safety verification, the Banker's Algorithm not only prevents deadlocks but also identifies conditions that could potentially lead to deadlock situations, thereby bolstering the overall resilience and dependability of concurrent systems.