National Chung Cheng University

Operating System Final Report

Research and Implementation of Deadlock

Student: 610415136 陳廷宇

610415140 詹京展

January 3, 2023

Table of contents

Chapter 1. Introduction

1.1. Deadlock

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. In Fig 1, Process 1 is holding resource 1, and waiting for resource 2, and process 2 is holding resource 2 and waiting for resource 1, then the deadlock will occur. Deadlock happened in our daily life, just as the traffic jam, shown in Fig 2.

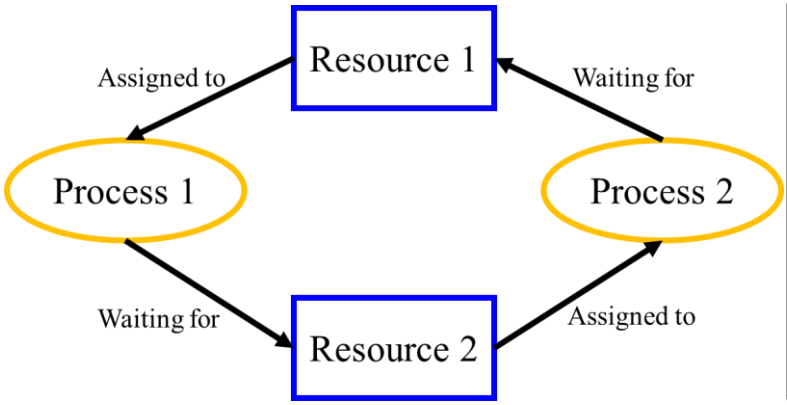

Fig 1 Deadlock condition

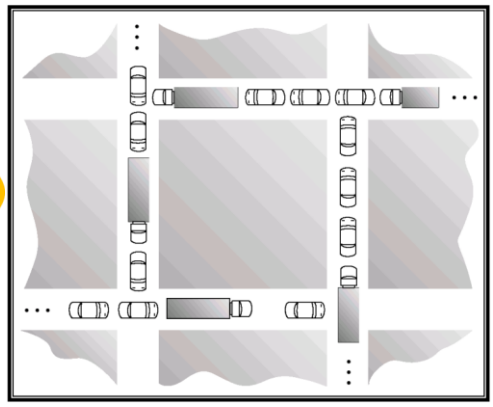

Fig 2 Deadlock situation (Traffic Jam)

Deadlock can arise if the following four conditions hold simultaneously, there're necessary conditions. First, mutual exclusion, means two or more resources are non-shareable, in the other way only one process can use at a time; hold and wait, means a process is holding at least one resource and waiting for resources; no preemption, means resource cannot be taken from a process unless the process releases the resource; circular wait, means a set of processes are waiting for each other in circular form.

1.2. Solve deadlock problem

There are three methods to solve deadlock, deadlock prevention, deadlock avoidance (Banker's algorithm) and deadlock detection & recovery. The advantage of deadlock prevention and deadlock avoidance is will make sure system won't happen deadlock, and the disadvantage of both methods are low resource utilization and low system throughput. And the advantage of deadlock detection & recovery is having better resource utilization, and the disadvantage of deadlock detection & recovery is system may happen deadlock, if deadlock happen, need to detection and do recovery, it takes much more cost. And deadlock prevention usually breaks one of the four necessary

condition to make sure not occur deadlock. Banker's algorithm is the common way to avoid deadlock, it can make sure system is in safety state, that won't happen deadlock.

Chapter 2. Deadlock Avoidance

2.1. Banker's algorithm

When process makes an application for a resource, OS will do Banker's algorithm, to make sure system is in a safe state, and there exists a safe sequence to satisfy requests by all processes. Safe state won't happen deadlock, unsafe state possibly will happen deadlock, shown in Fig 3. And the Banker's algorithm can derive into two algorithm, Banker's and Safety algorithm. The follow will be data structures and execute steps of two algorithm.

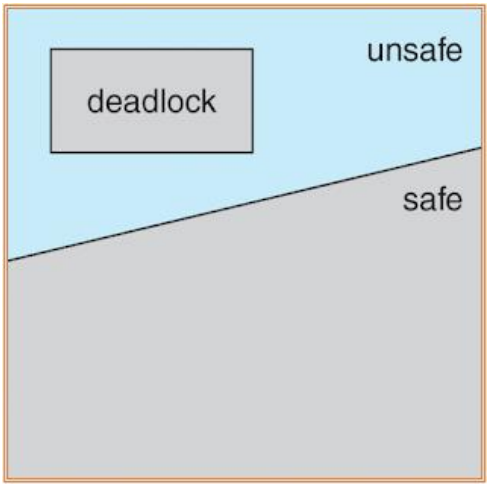

Fig 3. Deadlock avoidance: Safe & Unsafe

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[m]: indicates how many resources are currently available of each type.
- Max[n][m]: indicates the maximum demand of each process of each resource.
- Allocation[n][m]: indicates the number of each resource category allocated to each process, dynamically released resources at runtime does not affect its value.
- Need[n][m]: indicates the remaining resources needed of each type for each process.

Next, it's the step of the Banker's algorithm.

Step 1. If Request <= Need, go to step 2; otherwise, raise an error condition, since the process has exceeded its maximum claim.

Step 2. If Request <= Available, go to step 3; otherwise, Process must wait, since the resources are not available.

Step 3. Have the system pretend to have allocated the requested resources to process

by modifying the state as follows:

- Available = Available – Request
- Allocation = Allocation + Request
- Need = Need– Request

Step 4. Execution Safety algorithm, finding out whether or not a system is in a safe state, then allow application.

Following Data structures are used to implement the Safety Algorithm:

- Work[m]: Assume after allocated resource, the number of resource that system can work.
- Finish[n] of Boolean: Finish[i] means Process i finish or not.
  - True: finish works; False: not finish yet.
  - Initial: Finish[i] = False, i = 1~n.

Next, it's the step of the Safety Algorithm:

Step 1. Work = Available; Finish[i] = False, i = 1~n.

Step 2. Find an i such that both happen:

- Need <= Work
- Finish[i] = False
- If no such i exists, go to step 4.

Step 3. Work = Work + Allocation; Finish[i] = true; go to step 2.

Step 4. If Finish[i] == true for all i, then the system is in safe state, and have a Safe Sequence.

2.2. Limitations of Banker's algorithm

Although Banker's algorithm is a great way to avoid deadlock, but still have some limitation need to be improved. Follows are the limitations of Banker's algorithm, it needs to know how much of each resource a process could possibly request; in most systems, the information about resource and processes is unavailable, which makes it impossible to implement the Banker's algorithm; it is unrealistic to assume that the number of processes is static since in most systems the number of processes varies dynamically.

2.3. History of improvement of the Banker's algorithm

There are three way that scholar mostly do to improve the Banker's algorithm, Correction, Efficiency and Concurrency. The history of improvement of Banker's algorithm is summarized in Fig 4.

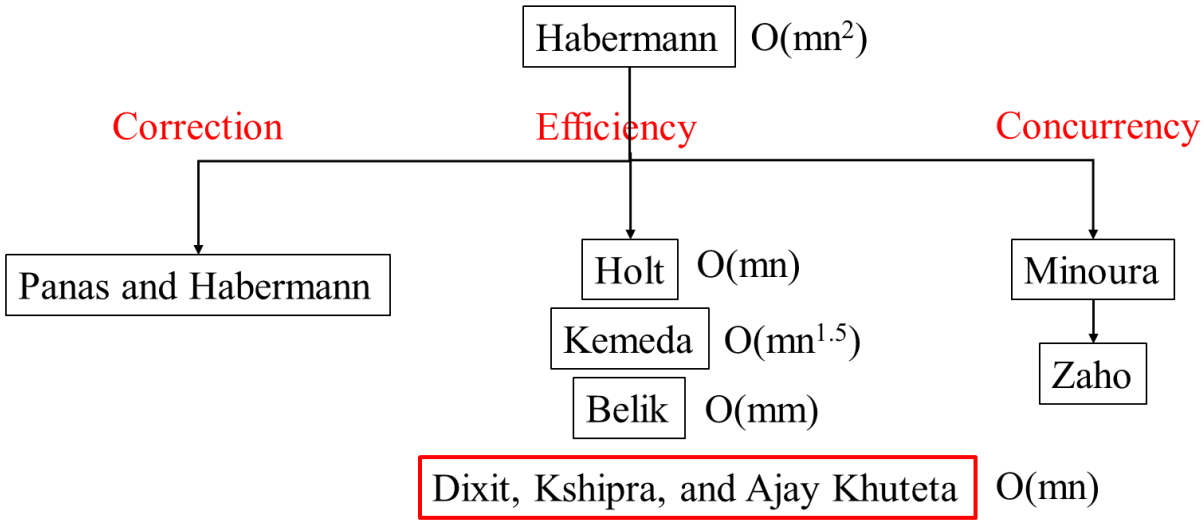

Fig 4. History of Banker's algorithm

2.4. Modified Banker's algorithm

The above table contains the available resource, and the schematic diagram of the stack on the right. The table in the lower left corner also contains items such as process, max demand, allocation, and need, and we also consider three different types of resources to allocate to these five processes. As shown in Fig 5.

AVAILABLE RESOURCES:

R1	R2	R3
0	2	3

Process	Max Demand			Allocation			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P0	11	3	4	3	0	2	8	3	2
P1	8	3	4	3	0	0	5	3	4
P2	1	3	5	0	1	3	1	2	2
P3	2	3	4	2	1	1	0	2	3
P4	3	3	7	3	1	1	0	2	6
Allocated resources	11			3			7		

Process STACK Values:

P2
P4
P1
P0

Fig 5. All resource usage

Next, introduce the improved algorithm, the following is its pseudo code.

- Modified banker's algorithm:

- 1. FOR Process(i)
  WHERE i=0 to n-1
  CHECK Need (i)
- 2. IF Need (i)<= Available
  THEN
  a. Available = Available –Need (i);
  b. execute (i);
  c. Available = Available + Allocation;
  d. write process execute
  e. go to next process
- 3. ELSE
  INSERT Need INTO STACK
- 4. END

This is the algorithm in our reference literature. You can see that the first point is to check the resources required by each of our processes, and then use the judgment formula to judge whether the resources we have at hand are sufficient to supply the process. The second part is to judge the demand of the process first. If there are not enough resources available, put the process in the stack, and use parameters to record how many layers of the stack there are. If possible, execute the steps from a to c. That is, if it can be supplied to this process, it will be executed directly. The above is the general flow of the algorithm we implemented this time.

Chapter 3. Implementation

3.1. Set up tools and environment

The following is the environment and tools of the implementation.
CPU: 11th Gen Intel® Core™ i5-11400 @ 2.60GHz × 12.
Memory: 32 Gigabyte.
Operating system: Ubuntu 22.04.1 LTS.
Programming language: C.
gcc version: 11.3.0.

3.2. Implementation of deadlock

The POSIX thread libraries are a standards-based thread API for C language. The follow will be code introduction. First announce **threadA** and **threadB**, then announce the data structure **attr**, include the schedule policy, inherit schedule and stack set. When finish the thread need to release the resource, so we need **pthread_mutex_destroy** in the end. The **pthread_mutex_init** is to create the two threads, and get the address from **threadA** and **threadB**. Pthread join is to wait for the **threadA** and **threadB** finish their work, and go to next step, otherwise program will be stuck here.

```
11  int main(int argc, char *argv[]) {
12
13      pthread_t threadA, threadB;
14      pthread_attr_t attr;
15
16      pthread_attr_init(&attr);
17      pthread_mutex_init(&x, NULL);
18      pthread_mutex_init(&y, NULL);
19
20      pthread_create(&threadA, &attr, A, NULL);
21      pthread_create(&threadB, &attr, B, NULL);
22
23      pthread_join(threadA, NULL);
24      pthread_join(threadB, NULL);
25
26      pthread_mutex_destroy(&x);
27      pthread_mutex_destroy(&y);
28  }
```

Fig 6. Implementation of deadlock code 1

The follow is the function of thread A and thread B, shown in Fig 7. Pthread_mutex was initial in main function. When pthread_mutex_lock happen, it means the thread was locked, no one can access this thread, only if it was unlocked.

```
30  void *A() {
31
32      pthread_mutex_lock(&x);
33      printf("A lock x\n");
34      pthread_mutex_lock(&y);
35      printf("A lock y\n");
36
37      pthread_mutex_unlock(&y);
38      pthread_mutex_unlock(&x);
39
40      pthread_exit(0);
41  }
```

```
43  void *B() {
44
45      pthread_mutex_lock(&y);
46      printf("B lock y\n");
47      pthread_mutex_lock(&x);
48      printf("B lock x\n");
49
50      pthread_mutex_unlock(&x);
51      pthread_mutex_unlock(&y);
52
53      pthread_exit(0);
54  }
```

Fig 7. Implementation of deadlock code 2

Next, used gcc to compile and run the code deadlock.c in terminal, shown in Fig 8. The code will randomly lock, in the left A lock x first happen,

```
eparc3080@eparc:~/jhan$ gcc deadlock.c -o deadlock
eparc3080@eparc:~/jhan$ ./deadlock
A lock x
B lock y
```

```
eparc3080@eparc:~/jhan$ ./deadlock
B lock y
A lock x
```

Fig 8. Deadlock execute 1

Use terminal command **ps** to find out the deadlock process. The execution is stuck now, shown in Fig 9.

```
eparc3080@eparc:~/jhan$ ps -ef|grep deadlock
eparc30+ 2683031 2681382  0 14:51 pts/2    00:00:00 ./deadlock
eparc30+ 2684846 2683636  0 14:53 pts/3    00:00:00 grep --color=auto deadlock
```

Fig 9. Deadlock execute 2

Use gdb to check the thread of deadlock.c.

```
eparc3080@eparc:~/jhan$ sudo gdb attach 2683031
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
Attaching to process 2683031
[New LWP 2683032]
[New LWP 2683033]
[Thread debugging using libthread_db enabled]
```

Fig 10. Deadlock execute 3

Use "**info thread**", check all the thread. It stuck at the first thread. There is a star in front of the number 1 thread, it means it's stop at number 1 thread right now, and the thread 2,3 is waiting for thread 1, show as futex_wait in Fig 11.

```
(gdb) info thread
  Id   Target Id                                          Frame
* 1    Thread 0x7f9259907740 (LWP 2683031) "deadlock" __futex_abstimed_wait_common64 (
    private=128, cancel=true, abstime=0x0, op=265, expected=2683032, futex_word=0x7f9259906910)
    at ./nptl/futex-internal.c:57
  2    Thread 0x7f9259906640 (LWP 2683032) "deadlock" futex_wait (private=0, expected=2,
    futex_word=0x55a7675f6080 <y>) at ../sysdeps/nptl/futex-internal.h:146
  3    Thread 0x7f9259105640 (LWP 2683033) "deadlock" futex_wait (private=0, expected=2,
    futex_word=0x55a7675f6040 <x>) at ../sysdeps/nptl/futex-internal.h:146
```

Fig 11. Deadlock execute 4

3.3. Implementation of Banker's algorithm

Original Banker's algorithm

After introducing the original banker algorithm, the follow is the implementation of the code. First, initialize all parameters and calculate the resources required by the process, that is, the need matrix, and then execute the safety algorithm. As shown in Fig 12.

```
230 int main()
231 {
232     clock_t t1, t2;
233
234     init(); // initial all the parameter, and calculate the Need matrix
235     t1 = clock();
236     showdata(); // print out the input
237
238     if(!safe()) exit(0); //Safaty algorithm
239     t2 = clock();
240     printf("%lf\n", (t2-t1)/(double)(CLOCKS_PER_SEC));
241
242 }
```

Fig 12. Implementation of original Banker's algorithm code 1

This is the function of Safe, which is used to find the safe sequence, and corresponds to the previous algorithm to execute the above program. At the beginning, the finish matrix is all false, and simply judges whether the resource is sufficient and releases the resource after the end. If a sequence is found that satisfies all processes Then the finish matrix is all true. As shown in Fig 13.

```
int safe()
{
    // initial
    int i,j,k=0,m,apply;
    for(j=0;j<N;j++)
        Work[j] = Available[j];
    for(i=0;i<M;i++)
        Finish[i] = False;
```

```
    //Safety sequence
    for(i=0;i<M;i++){
        apply=0;
        for(j=0;j<N;j++){
            if(Finish[i]==False && Need[i][j]<=Work[j])
            {
                apply++;
                if(apply==N)
                {
                    for(m=0;m<N;m++)
                        Work[m]=Work[m]+Allocation[i][m];
                    Finish[i]=True;
                    Security[k++]=i;
                    i=-1;
                }
            }
        }
    }
```

Fig 13. Implementation of original Banker's algorithm code 2

Modified Banker's algorithm

This is our modified Banker's algorithm. First, we declare the max, allocation, available, and need matrices. **tp** and **r** are the number of our processes and resources, respectively. Then, there are more stacks, maxstack matrices, and some matrices for stack operations. As shown in Fig 14.

```
    int tp;  //number of processes
    int r;   //number of resources
    int max[tp][r];  //max matrix
    int avail[r];  //available matrix
    int alloc[tp][r];  //allocation matrix
    int need[tp][r];  //max matrix - allocation matrix
    int stack[tp][r];  //stack matrix
    int maxstack[tp][r]; // duplicate this thread's max matrix
    int Ps[tp];  //record processes id
    int dup;  //record number of stacks
    int swap[tp][r];  //change processes's priority
    int add[dup];  //used in stack operaiton
    int diff[dup];  //used in stack operaiton
```

Fig 14. Implementation of modified Banker's algorithm code 1

The figure on the left is to judge the demand of the process. If there are not enough resources available, put the process into the stack, and use the dup parameter to record how many layers of the stack there are. The picture on the right shows the steps corresponding to a to c, that is, if it can be supplied to this process, it will be executed directly. As shown in Fig 15.

```
75      for(i=0;i<tp;i++){
76          for(j=0;j<r;j++)
77          {
78              if(need[i][j]>avail[j])
79              break;
80          }
81          if(j!=3)
82          {
83              for(k=0;k<r;k++)
84              {
85                  stack[dup][k]=need[i][k];
86                  maxstack[dup][k]=max[i][k];
87              }
88              Ps[dup]=i;
89              dup++;
90          }
```

```
91          else
92          {
93              safe[count++]=i;
94              printf("\n");
95              printf("P[%d] is executed successfully!!!",i);
96              printf("\n");
97              if(j==3){
98                  for(k=0;k<r;k++){
99                      avail[k]=avail[k]-need[i][k];
100                     avail[k]=avail[k]+alloc[i][k];
101                 }
102             }
103         }
104     }
```

Fig 15. Implementation of modified Banker's algorithm code 2

The next step is to process the processes that have just been put into the Stack. The figure on the left is to calculate the sum of the resources of each process to determine which resources have less demand and can be moved to the upper layer to increase the priority of execution. The picture on the right is the operation of the Stack to realize the movement in the stack. As shown in Fig 16.

```
107     for(i=0;i<dup;i++)
108     {
109         add[i]=0;
110         for(j=0;j<r;j++)
111         {
112             diff[i]+= stack[i][j];
113             add[i]=add[i]+diff[i];
114         }
115     }
```

```
116     int temp=add[0],swap;
117     for(i=0;i<dup;i++) {
118         for(j=i+1;j<dup;j++)  {
119
120             if(add[i]<add[j]) {
121                 temp=j;
122             }
123             for(k=0;k<r;k++)
124             {
125                 swap=stack[i][k];
126                 stack[i][k]=stack[temp][k];
127                 stack[temp][k]=swap;
128                 swap=maxstack[i][k];
129                 maxstack[i][k]=maxstack[temp][k];
130                 maxstack[i][k]=swap;
131             }
132             temp=Ps[i];
133             Ps[i]=Ps[j];
134             Ps[j]=temp;
135         }
136     }
```

Fig 16. Implementation of modified Banker's algorithm code 3

Execute in order after sorting, which is equivalent to finding out a safety sequence. As shown in Fig 17.

```
138     for(i=0;i<dup;i++)
139     {
140         for(k=0;k<r;k++){
141             avail[k]=avail[k]-stack[i][k];
142             avail[k]=avail[k]+alloc[i][k];
143         }
144         printf("\nP[%d] is executed successfully!!!\n",Ps[i]);
145     }
```

Fig 17. Modified Banker's algorithm execute 4

Finally, the results are visualized, and you can see that the safety sequence is found and the calculation time is obtained. As shown in Fig 18.

```
 a b c
P[0] 7 4 3
P[1] 1 2 2
P[2] 6 0 0
P[3] 0 1 1
P[4] 4 3 1

P[1] is executed successfully!!!

P[3] is executed successfully!!!

P[4] is executed successfully!!!

P[2] is executed successfully!!!

P[0] is executed successfully!!!
Times: 0.000233
```

Fig 18. Result of Modified Banker's algorithm

3.4. Results

After refer some paper about Banker's algorithm, most of them compared execution time and number of process, so we choose those to compare the efficiency of each algorithm. Shown in Fig 19, the x axis is number of process, the y axis is execution time, blue line is original Banker's algorithm, orange line is modified Banker's algorithm, both of them are linear growth. As the number of processes increases, the average execution time of both algorithms will increase, because more processes will lead to more resource contention and more frequent deadlocks. However, the modified Banker's algorithm can be better in all ranges, because the overhead involved in resource allocation is much lower than the original algorithm.

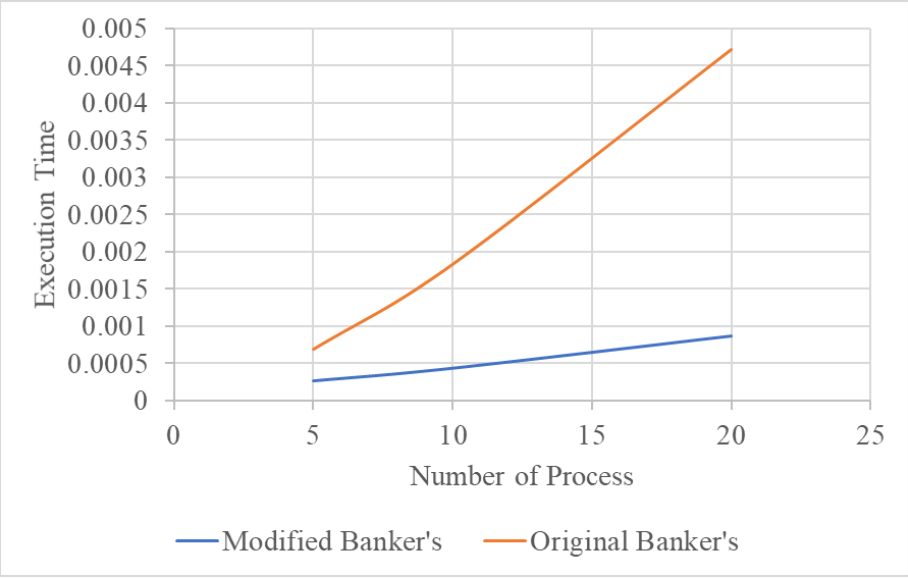

Fig 19. Experimental results

3.5. Merge Deadlock and Banker's algorithm

To make above two implementations more complete, merging Deadlock and Banker's algorithm. Shown in Fig 20, let the user choose which process need to be allocate.

```
void bank()
{
    int flag = True;
    int i,j;

    printf("Input which process need to be allocate(0-%d):",M-1);
    scanf("%d",&i);

    printf("Input number of resource that process %d need:\n",i);
    for(j=0;j<N;j++)
    {
        printf("%c:",NAME[j]);
        scanf("%d",&Request[j]);
    }

    for (j=0;j<N;j++)
    {
        if(Request[j]>Need[i][j])//Check allocate is lower than need
        {
            printf("Allocate not allow, it's oversize than need!\n");
            flag = False;
            break;
        }else{
            if(Request[j]>Available[j])//Check allocate is lower than system can use
            {
                printf("Allocate not allow, it's oversize than system can use!\n");
                flag = False;
                break;
            }
        }
    }
    if(flag) {
        test(i);
        showdata();
        if(!safe()) //find safety sequence
        {
            Retest(i);
            showdata();
        }
    }
}
```

Fig 20. Merge Deadlock and Banker's algorithm code 1

Shown in Fig 21, the system will ask user to input which process need to be allocate. When accept the requirement, system will allocate the resources to process, if not, won't allocate the resources.

```
        -------------------Banker's algorithm------------------
R(r):request to allocate resource
E(e):Exit
plz chooses:r
Input which process need to be allocate(0-4):0
Input number of resource that process 0 need:
:0
:1
:0
------------------------------------------------------------

Available Resources:
3  2  2
The current resource configuration of the system is as follows:
            Max          Allocation     Need
NAME
 P0         7  5  3       0  2  0       7  3  3
 P1         3  2  2       2  0  0       1  2  2
 P2         9  0  2       3  0  2       6  0  0
 P3         2  2  2       2  1  1       0  1  1
 P4         4  3  3       0  0  2       4  3  1
System is safe!
there is a safety sequence:P1->P3->P0->P2->P4
```

Fig 21. Merge Deadlock and Banker's algorithm execute 1

Chapter 4. Conclusion

The original Banker’s algorithm does not consider the part of dynamic process. When using the modified Banker’s algorithm, the time of execution decrease 50% more. The modified Banker’s algorithm has decreased the time complexity. By combining the deadlock algorithm and the banker's algorithm, the algorithm can be applied to more fields, such as multi-threaded programming to prevent deadlocks, the system will first request the resources required by the thread to determine whether the resource allocation can be completed this time. In future, it can be used for the auto added process and killing the undesired process.

Description of member contribution (%)

We both get 50% of this project.

Reference

[1] Dixit, Kshipra, and Ajay Khuteta. “A dynamic and improved implementation of banker’s algorithm.” *International Journal on Recent and Innovation Trends in Computing and Communication* 5.8 (2017): 45-49.