Journal of Advanced Computer Science & Technology, 4 (2) (2015) 237-247
www.sciencepubco.com/index.php/JACST
©Science Publishing Corporation
doi: 10.14419/jacst.v4i2.4857
Research Paper

Design and implementing a software tool to ensure undeadlock state by perfect distribution of resources' instances among competing processes

Imad Matti Bakko [1]*, Ruwaida Mohammed Yas [2]

[1] *Lecturer in Computer Science Dep. AL Ma'Mon University College, Baghdad, Iraq*
[2] *Assist. Lecturer in Computer Science Dep. AL Ma'Mon University College, Baghdad, Iraq*
**Corresponding author E-mail: emad_matti@yahoo.com*

Copyright © 2015 Imad Matti Bakko, Ruwaida Mohammed Yas. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

In computer operating systems books, they explain and solve deadlock problems by declaring in advance the maximum needs of resources and their instances for each process, the total number of resources' instances, and the allocation of the resources' instances for each process. In this paper, we introduce an effective software tool to prove that it is not necessary to declare in advance the allocation of resources' instances for each process since we suggested and implement in the tool some equations to calculate and discover a suitable allocation of resources' instances to be distributed among competing processes in such a way that the computer system will never enter a deadlock state. In fact, the only necessary and sufficient conditions to solve deadlock problems are the total number of resources and their instances besides the maximum needs of resources and their instances for each process. The theory and details are explained with some examples in the contents.

Keywords: *Operating Systems Deadlock; Software Tool; Safety Algorithm; Processes; Resources.*

1. Introduction

A computer system consists of a finite number of Resources to be distributed among a number of competing Processes [1 - 4].

1.2. Deadlock definition

A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set [1 - 4].

1.3. Deadlock necessary conditions

A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system [1 - 12]. These are:

a) Mutual Exclusion: Only one process at a time can use the resource, if another process requests that resource, the requesting process must be delayed until the resource has been released.
b) Hold and Wait: There must exist a process that is holding at least one resource and it is waiting to acquire additional resources that are currently being held by other processes.
c) No Preemption: resources cannot be preempted.

d) Circular Wait: There must exist a set {p0, p1, …, pn} of waiting processes such as that p0 is waiting for a resource that is held by p1 p1 is waiting for a resource that is held by p2, … pn-1 is waiting for a resource that is held by pn and pn is waiting for a resource that is held by $P0 \to P1 \to P2, ..., Pn-1 \to Pn \to P0.$

1.4. Resource-allocation graph

Deadlock can be described more precisely in term of a directed graph called resource allocation graph [1].

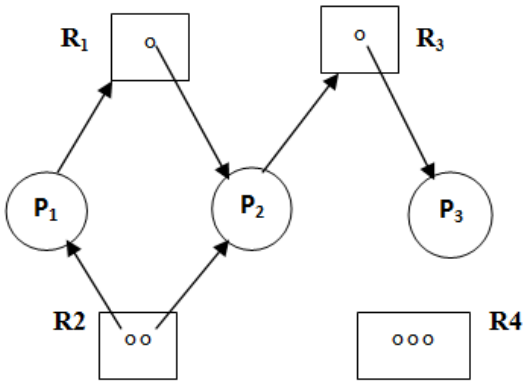


Fig. 1: Resource Allocation Graph without a Deadlock

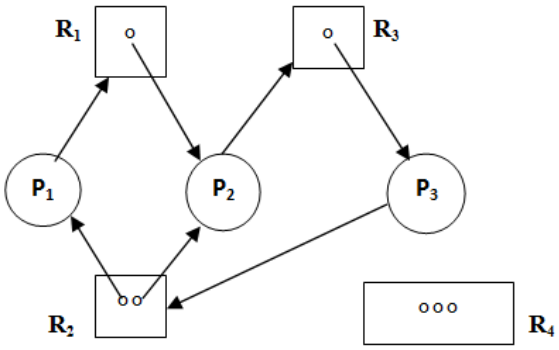


Fig. 2: Resource Allocation Graph with a Deadlock

1.5. Methods for handling deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways [1]:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter deadlock state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks [1].

2. The theory and details of our method for handling deadlock

In contrast to all deadlock problems used in operating systems books, we proved that it is not necessary to declare in advance the allocation of resource's instances for each process, instead of that we suggested an equation and inequality that calculate and discover a suitable number of allocated resource's instances to be distributed among a number of competing processes. Therefore, the only necessary and sufficient conditions to solve deadlock problems are the total number of instances in each resource besides the number of maximum needs of a resource' instances for each process. We suggested the following equation (1) and inequality (2) below:

Allocation (Pi) Rj = factor ∗ maximum needs (Pi) Rj. (1)

Where i = 1 … n and j = 1 … m, such that:

$\sum_{j=1}^{j=m} Allocation$ (Pi)Rj ≤ total instances (Rj). i=1 … n. (2)

- Pi stands for any process i in the system, where i = 1,…, n.
- Rj stands for any resource j in the system, where j = 1,…, m.
- Allocation (Pi) Rj stands for number of resource' instances of resource Rj allocated to process Pi.

- Total instances (Rj) stands for total number of instances in the resource Rj.

The element "factor" in equation (1) is an integer fraction number suggested to play a major role in discovering a safety state (undeadlock state). We start assigning the fraction factor with 0.5 and then we calculate the allocation for each process as in equation (1), after that we check the state of the system whether or not it is in a deadlock state, by applying the safety algorithm, if the system is still in a deadlock state we decrement the factor by 0.1, and again by calculating the allocation for each process we check the state of the system, and so on in an iterative manner until we reach to a safety state, and the allocation number of resource' instances assigned to each process will be the suitable choice.
Example (1): taken from operating system book [1].
Consider a system with 12 magnetic tape, and 3 processes p0, p1, p2. The maximum needs and the allocation needs for each process are given in the following table:

Table 1: Checking Deadlock State

Process	Maximum needs	Allocation needs
P0	10	5
P1	4	2
P2	9	2

In this example, the numbers of resources' instances allocated for each process are declared in advance to decide whether the system is in a deadlock state or not. Then by using one of the deadlock algorithms, such as safety algorithm, the system will be in an undeadlock state, and the safety sequence is < p1, p0, p2 >. On the other hand, we can solve this deadlock problem without the need to declare in advance the allocation needs for each process. In fact, it's sufficient for the following information:

Table 2: The Necessary and Sufficient Information to Solve Deadlock Problem.

Process	Maximum needs	Total resource' instances
P0	10	
P1	4	12
P2	9	

To prove this, we use the suggested equations 1 and the inequality 2 presented earlier. By using equation 1, we assign the factor to 0.5 for the first iteration then:
Allocation (P0) R = factor $*$ maximum needs (P0) R = $0.5 * 10 = 5$ instances.
Allocation (P1) R = factor $*$ maximum needs (P1) R = $0.5 * 4 = 2$ instances.
Allocation (P2) R = factor $*$ maximum needs (P2) R = $0.5 * 9 = 4.5 \cong 4$ instances, the fraction 0.5 is discard.
Then, we find the remaining current needs for each process are as follows:
Current need (Pi) = maximum needs (Pi) – allocation needs (Pi) [1].
We get the current needs and the allocation for each process as shown in the following table:

Table 3: Information When the Factor = 0.5

Process	Maximum needs	Allocation needs	Current needs
P0	10	5	5
P1	4	2	2
P2	9	4	5

Now, the suggested inequality (2) is true since:

$$\sum_{i=0}^{i=2} \text{allocation(pi)} \leq \text{total rsource'instances(R)}$$

$$5 + 2 + 4 = 11 \leq 12$$

Finally, we find the available needs of resources' instances remaining as follows [1]:
Available needs = total resources' instances(R) – $\sum_{i=0}^{i=2} allocation(pi)$ = 12 – 11 = 1 instance.
By using the safety algorithm , we find that the system is in a deadlock state since the available needs = 1 instance is not sufficient for P0 which needs 5 instances to complete execution, P1 needs 2 instances to complete execution, and P2 needs 5 instances to complete execution. The software tool tries a second iteration with the factor decremented by 0.1, hence factor = 0.4.By using equation (1) again we get:
Allocation (P0) R = factor $*$ maximum needs (P0) R = $0.4 * 10 = 4$ instances.
Allocation (P1) R = factor $*$ maximum needs (P1) R = $0.4 * 4 = 1.6 \cong 1$ instances, the fraction 0.6 is discard.
Allocation (P2) R = factor $*$ maximum needs (P2) R = $0.4 * 9 = 3.6 \cong 3$ instances, the fraction 0.6 is discard.
After that, we find the remaining current needs for each process as follows:
Current need (Pi) = maximum needs (Pi) – allocation needs (Pi) [1]. We get:

Table 4: Information When the Factor = 0.4

Process	Maximum needs	Allocation needs	Current needs
P0	10	4	6
P1	4	1	3
P2	9	3	6

Now, the suggested inequality (2) is true since:

$$\sum_{i=0}^{i=2} \text{allocation(pi)} \leq \text{total resource}' \text{instances(R)}$$

$4 + 1 + 3 = 8 \leq 12$

Finally, we find the available needs of resources' instances remaining as follows [1]:
Available needs = total resources' instances(R) – $\sum_{i=0}^{i=2} allocation(pi)$ = 12 – 8 = 4 instance.
And by using safety algorithm, we find that the system is in a deadlock state again, since the available = 4 is sufficient only for process P1 because it will take 3 instances from the available needs = 4, and the remaining available will become = 1. After executing, P1 will release 4 instances so the available will become 4 + 1 = 5 instances. 5 instances available are not sufficient for P0 nor it's sufficient for P2, because each one of them needs 6 instances as indicated in the table above, hence the system is in a deadlock again. The software tool tries a third iteration with the factor decremented by 0.1, hence factor = 0.3. By using equation (1) once more we get:
Allocation (P0) R = factor $*$ maximum needs (P0) R = $0.3 * 10 = 3$ instances.
Allocation (P1) R = factor $*$ maximum needs (P1) R = $0.3 * 4 = 1.2 \cong 1$ instances, the fraction 0.2 is discard.
Allocation (P2) R = factor $*$ maximum needs (P2) R = $0.3 * 9 = 2.7 \cong 2$ instances, the fraction 0.7 is discard.
After that, we find the remaining current needs for each process, and as follows:
Current need (Pi) = maximum needs (Pi) – allocation needs (Pi) [1]. We get:

Table 5: Information When the Factor = 0.3

Process	Maximum needs	Allocation needs	Current needs
P0	10	3	7
P1	4	1	3
P2	9	2	7

Now, the suggested inequality (2) is true since:

$$\sum_{i=0}^{i=2} allocation(pi) \leq total\ resource'instances(R)$$

$3 + 1 + 2 = 6 \leq 12$

Finally, we find the available needs of resources' instances remaining as follows [1]:
Available needs = total resources' instances(R) – $\sum_{i=0}^{i=2} allocation(pi)$ = 12 – 6 = 6 instances.
And by using safety algorithm, we find that the system is not in a deadlock state, since P1 will take 3 instances from the available (the remaining available will become = 3), after P1 finish executing it will release its maximum needs which are 4 instances, the available will become 3 + 4 = 7 instances which are sufficient for either P0 or P2. Therefore, we will get two safety sequences, < P1, P2, and P0 >, and < P1, P0, P2 >. From this example we conclude that our software tool can calculate and discover a suitable allocation of resource' instances for each process without the need for declaring allocation needs for each process in advance, that is the key behind our paper. The suitable allocation needs calculated and discovered are 3 resource' instances for process P0, 1 for P1, and 2 for P2. Moreover if we continue with other values for factor such as 0.2 or 0.1 we will get more undeadlock states with other safety sequences.

3. The software input and output of example (1)

We used C++ programming language to implement the software, and the following are the results after the running:
Enter number of Resources: 1
Enter Total number of instances in the Resource: 12
Enter number of Processes: 3
Enter Maximum Needs for each Process: P (0) = 10, P (1) = 4, P (2) = 9.
By using the factor = 0.5, we get the following information:
The number of instances allocated for processes are: P (0) = 5, P (1) = 2, P (2) = 4.
The sum of allocated instances = 11. The available of resources' instances = 12 – 11 = 1.
The current needs: P (0) = 5, P (1) = 2, P (2) = 5.
By using 'Safety Algorithm' with these information the system will be in a deadlock state.

Again by using the factor = 0.4, we get the following information:
The number of instances allocated for processes are: P (0) = 4, P (1) = 1 , P (2) = 3.
The sum of allocated instances = 8. The available of resource' instances = 12 – 8 = 4.
The current needs are: P (0) = 6, P (1) = 3, P (2) = 6.
And by using 'Safety Algorithm' with these information, the system will be again in a deadlock state.
Again by using the factor = 0.3, we get the following information:
The number of instances allocated for processes are: P (0) = 3, P (1) = 1, P (2) = 2.
The sum of allocated instances = 6. The available of resources' instances = 12 – 6 = 6.
The current needs: P (0) = 7, P (1) = 3, P (2) = 7.
By using 'Safety Algorithm' with this information the system will not be in a deadlock state, and the safety sequence will be : < P1, P2, P0>. This result is identical to the details of example (1) above.
Example (2): Taken from operating systems book [1].
Consider a system with five processes { P0 , P1 , P2 , P3 , P4 }, and three resource types { R0 , R1 , R2 }, resource type R0 has 10 instances, R1 has 5 instances, and R2 has 7 instances. Suppose also that at time T0 the following snapshot of the system has been taken:

Table 6: Checking Deadlock State.

	Maximum Needs			Allocation		
Process	R0	R1	R2	R0	R1	R2
P0	7	5	3	0	1	0
P1	3	2	2	2	0	0
P2	9	0	2	3	0	2
P3	2	2	2	2	1	1
P4	4	3	3	0	0	2

In this example, the writer of the book declares in advance the number of resource instances allocated for each process, and then to solve this problem, the writer uses the safety algorithm to reach to the following information:

Table 7: Information to Check Deadlock State.

	Maximum Needs			Allocation			Available			Current Needs			
Process	R0	R1	R2	R0	R1	R2	R0	R1	R2	R0	R1	R2	Finish
P0	7	5	3	0	1	0	3	3	2	7	4	3	F→T
P1	3	2	2	2	0	0				1	2	2	F→T
P2	9	0	2	3	0	2				6	0	0	F→T
P3	2	2	2	2	1	1				0	1	1	F→T
P4	4	3	3	0	0	2				4	3	1	F→T

And according to the safety algorithm, the system is not in a deadlock state, and the safety sequence is :< P1, P3, P4, P0, P2 >
On the other hand, we can solve this problem by using our software tool without the need for the allocation of resources instances to be declared for each process in advance, for this purpose we need only the following information:

Table 8: The Necessary and Sufficient Information.

	Maximum needs		
Process	R0	R1	R2
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Total Resource Instances		
R0	R1	R2
10	5	7

Now, according to the our theory presented earlier, and by using equation (1), we begin with factor = 0.5 for the first Iteration. We begin the calculations to discover the allocation of resources' instances for each process as follows:
Allocation(P0) (R0R1R2) = factor * maximum needs(P0) (R0R1R2) = 0.5 * (7, 5, 3) = (3, 2, 1) with discard fractions.
Allocation(P1) (R0R1R2) = factor * maximum needs(P1) (R0R1R2) = 0.5 * (3, 2, 2) = (1, 1, 1) with discard fractions.

Allocation (P2) (R0R1R2) = factor * maximum needs (P2) (R0R1R2) = 0.5 * (9, 0, 2) = (4, 0, 1) with discard fractions.
Allocation (P3) (R0R1R2) = factor * maximum needs (P3) (R0R1R2) = 0.5 * (2, 2, 2) = (1, 1, 1) with discard fractions.
Allocation (P4) (R0R1R2) = factor * maximum needs (P4) (R0R1R2) = 0.5 * (4, 3, 3) = (2, 1, 1) with discard fractions.
Then, we find the remaining current needs for each process as follows:
Current needs (Pi) = maximum needs (Pi) – allocation needs (Pi) [1]. We get

Table 9: Information when the Factor = 0.5

	Maximum Needs			Allocation			Current Needs		
Process	R0	R1	R2	R0	R1	R2	R0	R1	R2
P0	7	5	3	3	2	1	4	3	2
P1	3	2	2	1	1	1	2	1	1
P2	9	0	2	4	0	1	5	0	1
P3	2	2	2	1	1	1	1	1	1
P4	4	3	3	2	1	1	2	2	2

By using the inequality (2):
$\sum_{i=0}^{i=4} allocation\ (\ P\text{i}\ )$(R0R1R2) ≤ Total resources' instances (R0R1R2)
(11, 5, 5) ≤ (10, 5, 7)
This inequality is false since 11 is not less than 10. Hence no need to proceed, we try second iteration with factor = 0.4.
Allocation(P0) (R0R1R2) = factor * maximum needs (P0) (R0R1R2) = 0.4 * (7, 5, 3) = (2, 2, 1) with discard fractions.
Allocation(P1) (R0R1R2) = factor * maximum needs (P1) (R0R1R2) = 0.4 * (3, 2, 2) = (1, 0, 0) with discard fractions.
Allocation(P2) (R0R1R2) = factor * maximum needs (P2) (R0R1R2) = 0.4 * (9, 0, 2) = (3, 0, 0) with discard fractions.
Allocation(P3) (R0R1R2) = factor * maximum needs(P3) (R0R1R2) = 0.4 * (2, 2, 2) = (0, 0, 0) with discard fractions.
Allocation(P4) (R0R1R2) = factor * maximum needs(P4) (R0R1R2) = 0.4 * (4, 3, 3) = (1, 1, 1) with discard fractions.
Then, we find the remaining current needs for each process as follows:
Current needs (Pi) = maximum needs (Pi) – allocation needs (Pi) [1]. We get:

Table 10: Information When the Factor = 0.4

	Maximum Needs			Allocation			Current Needs		
Process	R0	R1	R2	R0	R1	R2	R0	R1	R2
P0	7	5	3	2	2	1	5	3	1
P1	3	2	2	1	0	0	2	2	2
P2	9	0	2	3	0	0	6	0	2
P3	2	2	2	0	0	0	2	2	2
P4	4	3	3	1	1	1	3	2	2

By using the inequality (2):
$\sum_{i=0}^{i=4} allocation\ (\ P\text{i}\ )$(R0R1R2) ≤ Total resources' instances (R0R1R2)
(7, 3, 2) ≤ (10, 5, and 7) is true.
Finally, we find the available of resource instances:
Available (R0R1R2) = Total (R0 R1 R2) - $\sum_{i=0}^{i=4} allocation\ (\ P\text{i}\ )$ (R0R1R2) = (10, 5, 7) – (7, 3, 2) = (3, 2, 5).
By using safety algorithm, we found that the system is not in a deadlock state, moreover, the safety sequence is:
< P1, P3, P4, P0, P2 >. We conclude from above, it's not necessary to declare in advance the allocation of resource instances for each process in all deadlock problems.

4. The software input and output of example (2)

Enter number of resources: 3.
Enter total instances of resource (0): 10.
Enter total instances of resource (1): 5.
Enter total instances of resource (2): 7.
Enter number of processes: 5.
Enter maximum needs of process (P0) : (7 , 5 , 3) .
Enter maximum needs of process (P1) : (3 , 2 , 2) .
Enter maximum needs of process (P2) : (9 , 0 , 2) .
Enter maximum needs of process (P3) : (2 , 2 , 2) .

Enter maximum needs of process (P4): (4, 3, 3).
By using factor = 0.5, the software tool discover the following allocation of resources' instances for the processes:

Table 11: Discovering Allocations (Factor = 0.5).

	Allocation
Process	R0 R1 R2
P1	3 2 1
P2	1 1 1
P3	4 0 1
P4	1 1 1
P5	2 1 1

$\sum_{i=0}^{i=4} allocation\ (\ Pi\ )$ (R0R1R2) = (11, 5, 5).
The system is in a deadlock state, since the sum of allocation of the resource R0 = 11 > 10, the total instances of R0.
By using 2nd iteration with factor = 0.4, the software discovers the following allocation and information:

Table 12: Information when the Factor = 0.4

	Maximum Needs			Allocation			Available			Current Needs			
Process	R0	R1	R2	R0	R1	R2	R0	R1	R2	R0	R1	R2	Finish
P0	7	5	3	2	2	1	3	2	5	5	3	2	F→T
P1	3	2	2	1	0	0				2	2	2	F→T
P2	9	0	2	3	0	0				6	0	2	F→T
P3	2	2	2	0	0	0				2	2	2	F→T
P4	4	3	3	1	1	1				3	2	2	F→T

$\sum_{i=0}^{i=4} allocation\ (\ Pi\ )$(R0R1R2) = (7, 3, 2).

By using safety algorithm, the system is not in a deadlock state, and the safety sequence is: < P1, P3, P4, P0, and P2 >.
We conclude from above that, it is not necessary to declare in advance the allocation of resource instances for each process in all deadlock problems. From this example we conclude that our software tool can calculate and explore a suitable allocation of resource' instances for each process without the need for declaring allocation needs for each process in advance, that is the key behind our paper.

5. The software tool

We implement our theory presented in paragraph 2 above by writing the following software to ensure the deadlock states will be never occur. We write it using Turbo C++ Programming Language version 4.5 and under Microsoft Office Word XP. While the paper text is written under Microsoft Office Word 2007, since C++ does not work under Office Word 2007.

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main ()
{clrscr ();
int total[10],r,i,j;
cout<<"enter number of resources: ";
cin>>r;
for (i=0;i<r;i++)
{cout<<"\n enter total number of instances in the resource ("<<i<<"):";
cin>>total[i];
cout<<"\n";
}
int process[10],p,sall[10],ava[10]={0},k,f,t,s=0,c[10],flag,cc=0;
int max[10][10],need[10][10],fin[10]={0},all[10][10];
float b[10][10];
double m;
cout<<"\n";
cout<<"enter number of processes: ";
cin>>p;
cout<<"\n enter maximum needs for each process";
```

```
for(i=0;i<p;i++)
{cout<<"\n p("<<i<<"): ";
for (j=0;j<r;j++)
cin>>max[i][j];
}cout<<"\n";
for(m=0.5;m>=0.1;m-=0.1)
{cout<<"\n by using the facter = "<<m<<",we get the following\n";
for(i=0;i<p;i++)
{for(j=0;j<r;j++)
b[i][j]=0;
all[i][j]=0;
}
 for(i=0;i<p;i++)
{for(j=0;j<r;j++)
{b[i][j]=max[i][j]*m;
all[i][j]=(int)b[i][j];
}}
int fin[10]={0,0,0,0,0,0,0,0,0,0};
cout<<"the number of instances allocated for process are: \n";
for(i=0;i<p;i++)
{cout<<"p("<<i<<")=";
{for(j=0;j<r;j++)
 cout<<all[i][j]<<" ";
 }cout<<"\n";
 }
 int sall[10]={0,0,0,0,0,0,0,0,0,0};
 for(j=0;j<r;j++)
 {int sum=0;
 for(i=0;i<p;i++)
 {sum=sum+all[i][j];
 }sall[j]=sum;
 }
 cout<<"the sum of allocated instances:\n";
for(j=0;j<r;j++)
cout<<"sall("<<j<<")="<<sall[j]<<" ";
s=0;
for(j=0;j<r;j++)
{if(sall[j]<=total[j])
s=s+1;
}
if(s==r)
{for(i=0;i<p;i++)
{ for(j=0;j<r;j++)
{need[i][j]=max[i][j]-all[i][j];
}}getch();
cout<<"\nthe current needs:\n";
 for(i=0;i<p;i++)
 { cout<<"p["<<i<<")= ";
 for(j=0;j<r;j++)
 {cout<<need[i][j]<<" ";
 }cout<<"\n";}
 for(j=0;j<r;j++)
 {ava[j]=total[j]-sall[j];
 cout<<"\nthe available of resources instances("<<j<<")="<<ava[j];
 }cout<<"\n";
 k=0;
 int cc=0;
 int c[10]={0,0,0,0,0,0,0,0,0,0};
 for(t=0;t<3;t++)
 { for(i=0;i<p;i++)
 {f=0;
            for(j=0;j<r;j++)
```

```
{{if(need[i][j]<=ava[j])
{f=f+1;
}}
if((f==r)&&(fin[i]==0))
{fin[i]=1;
for(j=0;j<r;j++)
ava[j]=ava[j]+all[i][j];
k=k+1;
c[cc]=i;
cc=cc+1;
}}}}
if(k==p)
{cout<<"\n by using safety algorithm with these information \n";
cout<<"the system will not be in a deadlock state,\n ";
cout<<"and the safety sequence is:";
cout<<"< ";
for(i=0;i<cc;i++)
{cout<<"p"<<c[i]<<" ";}
cout<<" >";
break;}
else
{cout<<"\nby using safety algorithm with these information \n";
        cout<<"the system will be in a deadlock state\n";
 continue; }
 }       } }
getch ();
```

6. Additional remarks

It is worth mentioning that increasing the total number of resources' instances will lead without any doubt to the undead lock state, but this is not the right solution to prevent deadlock, since the resources' instances are limited in computer system, in addition to its costs. In the deadlock problems, we can go further than that by reducing the total number of resources' instances to a minimum such that it will lead to an undead lock state. In example 1, the total resources' instances declared are 12 magnetic tapes, if we reduce them to 11magnetic tapes instead of 12 with giving a fraction of 0.2 to the factor(factor = 0.2),and by using our theory with implementing our software, this will reach us to the following information:

Table 13: Total Resources' Instances = 11 with Factor = 0.2

Process	Maximum needs	Allocation needs	Current needs
P0	10	2	8
P1	4	0	4
P2	9	1	8

Now, the suggested inequality (2) is true since:

$$\sum_{i=0}^{i=2} allocation(pi) \leq total\ resource'instances(R)$$

$2 + 0 + 1 = 3 \leq 12$

Finally, we find the available needs of resources' instances remaining as follows [1]:
Available needs = total resources' instances(R) – $\sum_{i=0}^{i=2} allocation(pi)$ = 11– 3 = 8 instances.
And by using safety algorithm, we find that the system is not in a deadlock state and the safety sequence is < P0, P1, and P2 >.
We can go further more by reducing the total number of resource' instances from 12 magnetic tapes to 10, and by giving the fraction 0.1 to the factor(factor = 0.1), and by using our theory with implementing our software this will be reach us to following information:

Table 14: Total Resources' Instances = 10 with Factor = 0.1

Process	Maximum needs	Allocation needs	Current needs
P0	10	1	9
P1	4	0	4
P2	9	0	9

Now, the suggested inequality (2) is true since:

$\sum_{i=0}^{i=2} allocation(pi) \leq total\ resource'instances(R)$

$1 + 0 + 0 = 1 \leq 10$

Finally, we find the available needs of resources' instances remaining as follows [1]:
Available needs = total resources' instances(R) – $\sum_{i=0}^{i=2} allocation(pi)$ = 10– 1 = 9 instances.
And by using safety algorithm, we find that the system is not in a deadlock state and the safety sequence is < P0, P1, P2 >.
In example 2, the total resources' instances are R0 = 10, R1 = 5, R2 = 7. If they replaced by R0 = 11, R1 = 5, R2 = 5, and by giving the fraction 0.4 to the factor (factor = 0.4), and by using our theory with implementing our software, this will be reach us to the following information:

Table 15: Total Resources' Instances (R0, R1, R2) = (10, 5, 7) with Factor = 0.4

	Maximum Needs			Allocation			Available			Current Needs			
Process	R0	R1	R2	R0	R1	R2	R0	R1	R2	R0	R1	R2	Finish
P0	7	5	3	2	2	1	4	2	3	5	3	2	F→T
P1	3	2	2	1	0	0				2	2	2	F→T
P2	9	0	2	3	0	0				6	0	2	F→T
P3	2	2	2	0	0	0				2	2	2	F→T
P4	4	3	3	1	1	1				3	2	2	F→T

And according to the safety algorithm, the system will not be in a deadlock state, and the safety sequence is:
< P1, P3, P4, P0, P2 >.

7. Conclusion

1) Referring to example 1 and 2, we conclude that our software tool can calculate and discover a suitable allocation of a resource' instances for each process without the need for declaring the allocation needs for each process in advance, that is the key behind our paper.
2) Referring to equation (1), we discard factor values 1, 0.9, 0.8, 0.7, 0.6 from the beginning since it will be led to the sum of allocation of instances to the processes beyond the total resource's instances; therefore, we suggest starting the factor with the value 0.5 down to 0.1 only.
3) Referring to example (1) and (2), we can proceed and continue decrementing the factor by 0.1 until the factor become 0.1 to discover more undeadlock states.
4) If the software did not discover undeadlock states when decrementing the factor from 0.5 down to 0.1, then the operating system is certainly in a deadlock state.
5) Referring to example 1, we start the values of the factor with one digit after the decimal point such as 0.5, 0.4, and 0.3. We check that there are more undeadlock states if we take into account two digits after the decimal fraction point. According to our theory, we calculate and conclude that the following factor values will be lead also to undeadlock states: 0.39, 0.38, 0.37, 0.36, 0.35, 0.34, 0.33, 0.32, and 0.31.
6) Referring to example 2, and to the conclusion point 5 above, we calculate and conclude that the following factor values will be lead also to undeadlock states: 0.44, 0.43, 0.42, and 0.41.
7) Finally, deadlock is a potential problem, thus the operating system designer should be well acquainted with solutions to this problem. The methods for handling deadlocks problems presented in the operating systems text books seem to be a viable approach to the deadlock problems; we think it is not very active. In this paper, we present a powerful software tool to calculate, discover and conclude suitable situations to reach to undeadlock states, so it may be taken into account when designing operating systems, therefore, it is recommended.

References

[1] Abraham Silberschatz , Peter Baer Galvin , Greg Gagne" Operating System Concepts " , 9th edition , chapter 7, deadlock, pages 318-322, copyright 2013 by John Wiley & Sons , Inc. Printed in the United State of America.

[2] Abraham Silberschatz, James L. Peterson "Operating System Concepts", 2nd edition, chapter 8 deadlock, pages 285, 288. Copyright 1985 by Addison-Wesley publishing company. Inc. printed in the United State of America.
[3] Andrew S. Tanenbaum "Modern Operating Systems", 3rd Edition, 2009, chapter 6 deadlock, page 436. Pearson Prentice Hall, Pearson Education, Inc. Printed in the United State of America.
[4] Harvey M. Deitel "An Introduction to Operating Systems", Chapter 6 Deadlock, page 131, Addision-Wesley. Publishing Company 3rd Edition, 1990.
[5] Ann mciver mchoes and ida m. Flynn "Understanding Operating Systems", chapter 5 processing management, deadlock, page 141, seventh edition, 2014 CENGAGE Learing, printed in the United State of America.
[6] J Archer Harris "Schaum's Outline of Operating System", New York, McGraw-Hill, 2002.
[7] www.googlechrome.com, [PDF] Abraham silberschatz, Peter Baer Galvin, and Greg Gagne, "Operating System Concepts " , 7th Edition , copyright 2005 by John Wiley & Sons , Inc. Printed in the United State of America.
[8] www.googlechrome.com, Deadlock operating systems," [PDF] ch5 deadlock pdf.
[9] www.googlechrome.com, Deadlock operating systems," [PDF] cos 318: operating system deadlock Princeton University "Andy Bavier Computer Science Department Princeton University http://www.cs.princeton.ed/cources/archive/fall10/cos318/lectures/deadlock.pdf.
[10] Www.googlechrome.com. Deadlock operating systems," [PDF] operating systems deadlocks ", Jerry Breecher.
[11] www.googlechrome.com. Deadlock operating systems," deadlock-wikipedia, the free encyclopedia ", https://en.wikipedia.org/wiki/deadlock.
[12] Www.googlechrome.com. Deadlock operating systems", [PDF] os chp8 deadlocks pdf ".