Preventing Deadlock with The Banker's Algorithm

Atika Yusuf 13510055
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13510055@std.stei.itb.ac.id

***Abstract*—Deadlock is a paradox in which two or more actions are each waiting for the other to finish and neither ever does. In this context, the writer will focus on avoiding deadlocks with the banker's algorithm. Deadlock is a very common issue in multiprocessing systems.**

***Index Terms*—deadlock, banker's algorithm, resource allocation**

I. INTRODUCTION

A recent development in operating systems involving multiprogramming, multiprocessing, etc – has been to improve the utilization of system resources and therefore would reduce the cost. In any type of operating system a deadlock condition must be considered and anticipated. Requests by separate tasks for resources may possibly be granted in a sequence that a group of two or more task is unable to continue, each task holds resources solely on its self and waits for the release of resources currently held by others in the system, which will never happen. This is called a deadlock.

A deadlock is a paradox, once a deadlock occurs, a system performance will be degraded and the only way to end it is to avoid a deadlock in the first place. Basically, it's a condition where two or more process request a resource that it used by another process thus such request will never be satisfied. Deadlocks are particularly complicated and troubling because there is no general solution to avoid one. However there are three algorithms with certain constraints that help to avoid deadlocks, one of them is the banker's algorithm.

The banker's algorithm was developed by a Dutch computer scientist, Edsger Dijkstra that tests for safety by simulating the allocation of pre-determination maximum possible amounts of all resources and then makes a safe state check to test for possible deadlock for all other pending activities before deciding whether allocation should be allowed to continue. The algorithm was developed in the design process for the THE (Technische Hogeschool Eindhoven) operating system and originally described in Dutch in EWD108. The name is an analogy for the way that bankers account for liquidity constraints, as in market liquidity is an asset's ability to be sold without causing a significant movement in the price and with minimum loss of value.

II. DEADLOCK

Deadlock is a situation in which two or more action are each waiting for the other to finish and neither ever does. Deadlock is common in multiprocessing where processes share a specific type of mutually exclusive resource.

In computer science, Coffman deadlock refers to a specific condition when two or more process are each waiting for the other to release a resource or more than two process are waiting for resources in a circular chain. There are four necessary conditions for a Coffman deadlock to occur: 1) Mutual Exclusion: a resource that cannot be used by more than one process at a time, 2) Hold and Wait/wait-for: processes already holding resources may request new resources held by other processes, 3) No Preemption: no resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process, 4) Circular Wait: two or more process form a circular chain where each process waits for a resource that the next process in the chain holds.

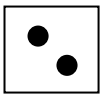
Basically, to prevent deadlock one of Coffman conditions (also known as the necessary conditions) must be eliminated:

- No mutual elimination: there's no deadlock if mutual exclusion is not needed. Sometimes resources can be partitioned to avoid mutual exclusion.
- No hold and wait/wait-for: by not letting a process wait for one resource while holding another, either by requiring each process to hold only one resource at a time or to request all of the resources it needs simultaneously.
- Allow preemption: the reason preemption should be allowed is because a lockout (no preemption) condition may be difficult or impossible to avoid as a process to be able to have a resource for a specific amount of time, or the processing outcome may be inconsistent.
- Eliminate cycles: circular wait could be eliminated by disabling interrupts during critical sections and use a hierarchy to determine a partial ordering of resources.

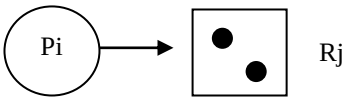
A resource allocation graph represents which processes are waiting for or holding each resource.

Each node in the graph represents either a process or a resource. A directed edge is drawn from process T1 to resource T2 if T1 is waiting for T2, and from T2 to T1 if T1 holds T2.

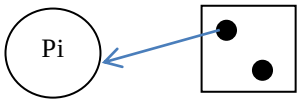
A process



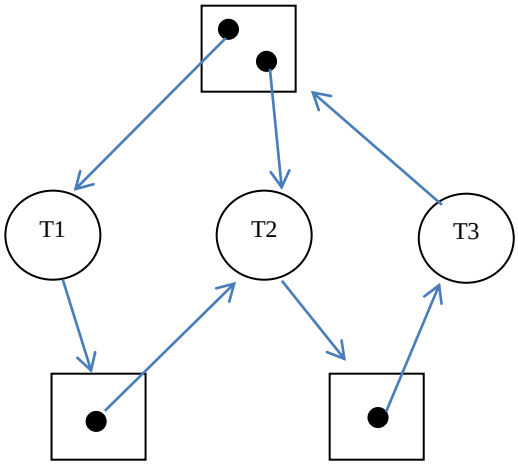
A resource with 2 instances



Pi requests Rj instances



One of Rj's instances is allocated to Pi



A deadlock

Consider two processes T1 and T2 each requiring the exclusive use of two different resources R1 and R2. The combined process can be represented in the following way. For each process, the number of instructions executed subsequent to some selected initial time is used as a measure of its progress and a pair of those values define a point in two dimensional progress space. The joint progress of T1 and T2 is then represented by a sequence of discrete points in this space. Sub-sequences in which only one coordinate increases correspond to time intervals in which one task is in control of the CPU, while simultaneously increases both coordinates can only occur in multiple processor system. It is obvious that such trajectory can never decrease either coordinate and the progress is irreversible.

Because the sequences of resource are used by T1 and T2, then if the trajectory is allowed to enter the state of a deadlock is unavoidable. T1 holds resource R1 and T2 holds resource R2 and the subsequent requests from each process for its second resource must be denied. Other processes in the system may be able to continue if they do not require there resources but the performance may be degraded due to R1 and R2 unavailability.

According to Coffman, deadlock situation has arisen only because all of the following general conditions were operative:

1. Processes claim exclusive control of the resources they require which is called mutual exclusive condition
2. Process hold resources that are already allocated to them while waiting for additional resource, this is called hold and wait condition.
3. Resources cannot be forcibly removed from tasks holding them until the resources are used to completion, this is called the no preemption condition
4. A circular chain of process exists, where each process holds one or more resources that are being requested by the next process in the chain, this is called the circular wait.

Deadlock detection is pretty simple. First we have threads that own locks, then threads that wait for lock to acquire. Think of a directed graph with locks and threads being nodes and each lock ownership and lock wait is a vertex that connects those nodes. If you can traverse the directed graph starting from a certain node and reach that node again, then we have a deadlock. This is because a thread waits for itself to release a lock to finally acquire a desired lock which obviously will never happen unless there's an action to release the block for one thread telling it that there was a deadlock and the thread should resolve it now it is able to perform actions.

By undertaking the design of a system in which the possibility of deadlock is to be excluded, we must be sure that at every point in time at least one of the necessary conditions is not met. It results in certain constraints in the way a request for resources may be granted.

Some approaches could be done for example each process must request all its required resources and cannot proceed until all have been granted. When a process is holding a certain resources and denied for another request, that process must release its original resources and if necessary request them again along with the additional resources. If a process has been allocated resources of certain type, it may subsequently request only the same type of resources.

Deadlock can be avoided if certain information about processes is available prior to resource allocation. One

known algorithm that is used to avoid deadlock is the Banker's algorithm which requires the resource usage limit number to be known in advance. Two other algorithms are wait/die and wound/wait. Both these algorithms have and older process and a younger process. Process age can be determined by timestamp at a process creation time. The smaller the time stamps the older the process.

III. THE BANKER'S ALGORITHM

The banker's algorithm is a resource allocation and deadlock prevention algorithm that tests for safety by simulating the allocation. This algorithm is not widely used in the real world because to use it the operating system must know the maximum amount of resources that every process is going to need at all times. That is why the banker's algorithm has some limitations when implemented. Specifically it needs to know how much of resource a process could possibly request. In most systems, this is information is could not be obtained, making it impossible to implement the banker's algorithm. Assuming that the number of process is static is also unrealistic since in most systems the number of process varies dynamically.

The banker's algorithm prevents deadlock by becoming involved in the granting and denying of system resources. Each time that a process needs a particular resource that is not shared, the request must go through the banker's approval.

Let us use real life bank as a metaphor. Think of the banker as a loaner. Every time a process makes a request for a resource (proposes for a loan), the banker takes a thorough look at the bank book and attempts to determine whether a deadlock situation could possibly arise in the future if the request is granted. The determination is made by simulating the process ad if the request is granted and then looking at the resulting post-granted request system state. After granting a resource there will be an amount of that resource left free in the system. As of the other processes in the system, the banker demand that each of these other processes state the maximum amount of all system resources they needed to terminate so the banker know how much of each resource every process is holding.

If the banker has enough free resource to guarantee that one process can terminate, then it would be able to take the resource held by that process and add it to the list. At this point the banker can look at the larger free list and attempt to guarantee that another process will terminate by checking whether the condition can be met. If the banker can guarantee that all process in the system will terminate, it grants the request.

However if the banker cannot guarantee any process will terminate because there is not enough free resource to meet the requirements, a deadlock may occur. This is called an unsafe state. In this case the loan request in question is denied and the requesting process is blocked.

The efficiency if the banker's algorithm lies on how it is implemented, for example if the bank books are kept sorted by process claim size, adding new process information to the table is *O(n)* but reducing the table is simplified. However is the table is kept in no order, adding new entry is *O(1).*

Here is an example of a safe state. Let n be the number of process running in the system and m be the number of resources types. Then the following data structures is required:

- A vector length of m represents the number of available resources of each type. Denotes available[j] = k, means there are k instances of resources type Rj available.
- An nxm matrix defines the maximum demand of each process. Max[i,j] = k then Pi may request at most k instances of resource type Rj.
- An nxm matrix defines the number of resources of each type currently allocated to each process. Allocation[i,j] = k then process Pi is currently allocated k instance of resource type Rj.
- An nxm matrix indicates the remaining resource need of each process. If need[i,j] = k then Pi may need k more instances of resource type Rj to completion.
- Need = max – allocation

Let there be four types of resources A, B, C and D.

```
Total resources in system:
A B C D
6 5 7 6
Available system resources are:
A B C D
3 1 1 2
Processes (currently allocated
resources):
   A B C D
P1 1 2 2 1
P2 1 0 3 3
P3 1 2 1 0
Processes (maximum resources):
   A B C D
P1 3 3 2 2
P2 1 2 3 4
P3 1 3 5 0
Need= maximum resources - currently
allocated resources
Processes (need resources):
   A B C D
```

```
P1  2  1  0  1
P2  0  2  0  1
P3  0  1  4  0
```

A state is considered safe if it is possible for all process to terminate. Since the system cannot know when a process will terminate or how many resources it will have requested by then, the system assumes that all processes will eventually acquire their stated maximum resources and terminate afterward. This is reasonable in most cases because the system is not concerned with how long each process run. And if in the end a process terminates without acquiring its maximum resources, it won't matter.

The banker determines a safe state by trying to find a set of requests by the process that would allow each to acquire its maximum resources and then terminate. And when such set does not exists, it is an unsafe state.

Here is a portion of the banker's algorithm in C that checks whether a state is safe:

```
#include<stdio.h>

//global variables.
int Pcurr[3][3]; //max of 3 processes and 3 resources

int Pmax[3][3];
int avl[]={6,4,7};
int avltemp[]={6,4,7};
int maxres[]={6,4,7};

int running[3]; //determines which processes are running

int i,j, safe=0,count=0;;

main()

{

	for(i=0;i<3;i++)
	running[i]=1;  //set all the processes to "running" = true (1)
	int ch;

	initresources();

	while(1)  //loop forever

	{

		system("clear");

		count=0;
		{
			if(running[i])
			count++;
		}
		if(count==0)
		{

	printf("\n\n\n\n\nCongratulations! We have completed execution of all processes successfully without any deadlock!");
			getchar();
			break;
		}

		//The following is a menu for the user to see what is going one at each iteration.
		else
		{
			printf("\nDeadlock Prevention using Banker's Algorithm:\n");
			viewresources();
			printf("\n1. Request resource(s) for a process\n");

			printf("\n2. View Allocated Resources\n");

			printf("\n3. Exit\n");

			printf("\nEnter your choice:\n");

			scanf("%d",&ch);

			if(ch==1)
			{

	requestresource();
				getchar();
			}
			else if(ch==2)
			{
				viewresources();
				getchar();
			}

			else if(ch==3)

			break;

			else

	printf("\nInvalid Choice, please try again!\n");

		}
	}

}

//initialization routine, this defines the current "problem" to be tested.
initresources()

{

	//for each process, get curr. requirement and max. requirement->check if max. req....
	Pmax[0][0]=3; Pcurr[0][0]=1; avl[0]=3;
	Pmax[0][1]=3; Pcurr[0][1]=2; avl[1]=1;
	Pmax[0][2]=2; Pcurr[0][2]=2; avl[2]=1;
```

```
	Pmax[1][0]=1; Pcurr[1][0]=1;
	Pmax[1][1]=2; Pcurr[1][1]=0;
	Pmax[1][2]=3; Pcurr[1][2]=3;
	Pmax[2][0]=1; Pcurr[2][0]=1;
	Pmax[2][1]=1; Pcurr[2][1]=1;
	Pmax[2][2]=5; Pcurr[2][2]=1;

}

requestresource()

{

	//check if it is allocated, whether
it will go to deadlock or not
	int proc, res[3];
	printf("\nFor which Process, you
need resources?(1-3):\n");
	scanf("%d",&proc);
	proc--;
	if(running[proc])
	{
		printf("\nCurrently this
process needs the foll. resources:\n");
		printf("R1\tR2\tR3\n");
		for(i=0;i<3;i++)

	printf("%d\t",Pmax[proc][i]-
Pcurr[proc][i]);
		for(i=0;i<3;i++)
		{
			loop_3:
			printf("\nEnter no. of
Resource %d to Allocate to Process
%d:\n",i+1,proc+1);
			scanf("%d",&res[i]);

	if((res[i]>(Pmax[proc][i]-
Pcurr[proc][i]))||(res[i]>avl[i]))
			{

	printf("\nInvalid Value!, try
again!");
				goto loop_3;

			}
		}
		getchar();
		if(allocate(proc,res))
		{
			printf("\nResources
successfully allocated.\n");

	if(checkcompletion(proc))
			printf("\nProcess %d
has completed its execution and its
resources have been released.\n",proc+1);
		}
		else
			printf("\nResouces
cannot be allocated as it may lead to
Deadlock!\n");
	}
	else
	{
		printf("\nInvalid Process
no.\n");
		getchar();
	}
}

///allocate a resource to a process
int allocate(int proc, int res[3])
{
	for(i=0;i<3;i++)
	{
		Pcurr[proc][i]+=res[i];
		avl[i]-=res[i];
	}
	if(!checksafe())
	{
		for(i=0;i<3;i++)
		{
			Pcurr[proc][i]-=res[i];
			avl[i]+=res[i];
		}
		return 0;
	}
	return 1;
}

int checkcompletion(int proc)
{
	if((Pcurr[proc][0]==Pmax[proc][0])&&
(Pcurr[proc][1]==Pmax[proc][1])&&(Pcurr[pro
c][2]==Pmax[proc][2]))
	{
		for(i=0;i<3;i++)
		{
			avl[i]+=Pmax[proc][i];
		}
		running[proc]=0;
		return 1;
	}
	return 0;
}


//print the state of the resources for the
user
viewresources()

{

	printf("\n----Current Snapshot of
the system----\n");
	printf("\nMax. resources in the
system:\n");
	printf("R1\tR2\tR3\n");
	for(i=0;i<3;i++)
	printf("%d\t",maxres[i]);
	printf("\nCurrent resources
available in the system:\n");
	printf("R1\tR2\tR3\n");
	for(i=0;i<3;i++)
	printf("%d\t",avl[i]);
	printf("\n\nMax. resources required
for Completion of each process:\n");
	printf("\tR1\tR2\tR3\n");
	for(i=0;i<3;i++)
	{
		if(running[i])
		{
			printf("P%d\t",i+1);
			for(j=0;j<3;j++)

	printf("%d\t",Pmax[i][j]);
			printf("\n");
```

```
                }
        }

        printf("\n\nCurr. resources
allocated for each process:\n");
        printf("\tR1\tR2\tR3\n");
        for(i=0;i<3;i++)
        {
                if(running[i])
                {
                        printf("P%d\t",i+1);
                        for(j=0;j<3;j++)

        printf("%d\t",Pcurr[i][j]);
                        printf("\n");
                }
        }
}


//the bankers algorithm portion of the code
that uses the algorithm to generate a safe
or unsafe value in boolean

int checksafe()
{
        //Check if at least one process can
get all resources it needs

        safe=0;
        for(i=0;i<3;i++)
        {
                avltemp[i]=avl[i];
        }
        for(i=0;i<3;i++)
        {
                if(running[i])
                {
                        if((Pmax[i][0]-
Pcurr[i][0]<=avltemp[0])&&(Pmax[i][1]-
Pcurr[i][1]<=avltemp[1])&&(Pmax[i][2]-
Pcurr[i][2]<=avltemp[2]))
                        {
                                for(j=0;j<3;j++)

        avltemp[j]+=Pcurr[i][j];
                                        safe=1;
                        }
                }
        }
        return safe;
}
```

IV. Conclusion

The banker's algorithm has one limitation that is critical, it cannot be implemented in a system that is unable to obtain certain information such as how much of each resource a process could get. Nevertheless, it is a good algorithm for deadlock avoidance since it checks the request before granting it.

References

[1] http://en.wikipedia.org/wiki/Banker's_algorithm
Sunday, December 11, 2011

[2] http://en.wikipedia.org/wiki/Deadlock
Sunday, December 11, 2011

[3] http://www.fearme.com/misc/alg/node149.html
Sunday, December 11, 2011

[4] http://forums.devx.com/showthread.php?t=169397
Sunday, December 11, 2011

[5] Rinaldi Munir, *Matematika Distrit.* Bandung: Informatika, 2005.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Desember 2011

Atika Yusuf 13510055