

# Deadlocks

M1 MoSIG : Operating Systems

Rouby Thomas

18/11/2014

## 1 Example

s = 1; available = N; occupied = 0;	
<b>T1</b>	<b>T2</b>
wait(s)	wait(s)
(*) wait(available)	wait(occupied)
.	.
.	.
.	.
post(occupied)	post(available)
post(s)	post(s)

T1 holds s and wait for available. Available will be release by T2 which waits for s. If the thread T1 waits in (\*) instruction, there will be a deadlock.

In general, a deadlock is a situation in which each thread in a set of threads hold a resource and is waiting for a resource held by another thread. It is only possible if :

- mutual exclusion : a resource held by a thread cannot be acquired by another.
- no preemption : a resource cannot be taken, even temporarily, from a thread which holds it.
- hold and wait : all the threads in the set hold at least one resource and wait for another one.
- circular wait : t1 waits for t2 to release its resources. t2 waits for t3, t3 waits for t4... tn waits for t1 .

To avoid deadlocks, there are several strategies :

- manage to avoid one of the conditions :
  - share resources so that all threads can access concurrently to them. But it is not always possible.
  - preemption : possible with CPU, but still not always possible.
  - hold and wait : we could require from threads to acquire at once all the resources they need. But it is not practical, not efficient.
  - circular wait : enforce an order for acquiring resources. But it is too difficult in dynamic setups.
- detect deadlocks.
- prevent deadlocks.

## 2 Resource allocation graph

We need some internal representation of the current state of the system. We are in a situation in which there are mutual exclusion, no preemption and hold and wait => We need to know if circular wait is present/possible.

Model : resources & threads are vertices of a graph.

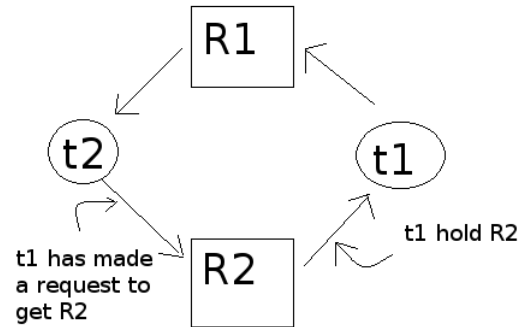


Figure 1: Resource allocation graph

There is a circular wait in the system if there is a cycle in this graph. => Detecting a deadlock is easy using this graph (using, for instance, a bellman-ford algorithm).

Another way to use this graph is to perform a cycle detection for each new request and to decide to refuse it if this induces a cycle in the graph. This model is not suited if resources are divided into types and if threads ask for one instance of a given resource. It can be extended :

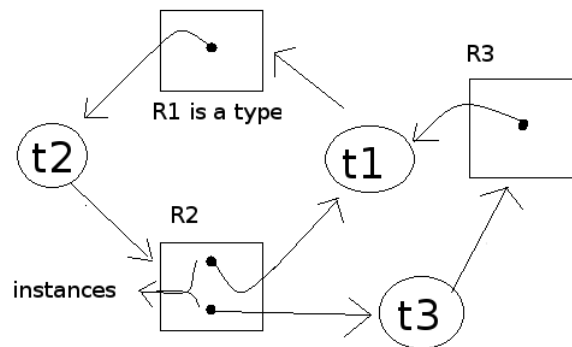


Figure 2: Extended resource allocation graph

In this model, the presence of a cycle doesn't necessarily means that there is a deadlock, only a set of cycles that saturates all the resources it contains, will be a deadlock. -- > Not practical.

## 3 Allocation matrices and banker algorithm

### 3.1 Hypothesis

At first we assume that we know :

- a vector Available such that :  $\text{Available}[i] = \text{number of instances of resources } i \text{ that are available.}$
- a matrix Allocation such that :  $\text{Allocation}[i][j] = \text{number of resources of type } i \text{ allocated to thread } j.$
- a matrix Max such that :  $\text{Max}[i][j] = \text{maximal number of resources of type } i \text{ the thread } j \text{ will need.}$
- a matrix Need such that :  $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$

We will try to find if there is an order that make the execution of all threads possible :

1.  $\text{Work} = \text{Available}$
2.  $\text{Finished}[j] = \text{false}$  for all threads.
3. while there is some thread  $j$  such that :  
 $\text{Finished}[j] = \text{false} \ \&\& \ \text{Need}[i][j] \leq \text{Work}[i] \ \forall i,$   
then :  $\text{Work}[i] += \text{Allocation}[i][j] \ \forall i ;$   
 $\text{Finished}[j] = \text{true};$
4. if there is a  $j$  such that  $\text{Finished}[j] = \text{false}$ , the system is not in a safe state.

This previous algorithm finds out if the system is in a safe state or not.

For deadlock prevention, we use the banker algorithm : when an allocation request is issue to the system :

1. Pretend the request is granted :  
 $\text{Available} -= \text{request};$   
 $\text{Allocation}[*][j] += \text{request};$
2. run the safe state detection
3. if the state is not safe, rollback and deny the request.

### 3.2 Example of the banker algorithm

	Allocation	A B C	Available	Need	A B C
	0	1 0 1	(0)1 0 2	0	0 2 0
Threads	1	0 0 1		1	2 1 0
	2	0 2 0		2	0 0 1
	3	(2)1 0 0		3	(1)2 2 0
	4	2 0 0		4	2 2 3

Assume that thread 3 requests 1 A resource (1,0,0) (represented in the example with the new numbers between paranthesis)

1. Pretend the request is accepted
2. Run the algorithm to know if the state is safe :  $\text{Work} = (0,0,2).$

Thread	Finished
0	f
1	f
2	f
3	f
4	f

Thread 2 has needs  $\leq$  Work : Work = (0,2,2).

Thread 0 has needs  $\leq$  Work : Work = (1,2,3).

Thread 3 has needs  $\leq$  Work : Work = (3,2,3).

Thread 4 has needs  $\leq$  Work : Work = (5,2,3).

Thread 1 has needs  $\leq$  Work : Work = (5,2,4).

So now we get :

Thread	Finished
0	t
1	t
2	t
3	t
4	t

3. Finished is filled with "true"  $\Rightarrow$  the request can be granted.

Assume that thread 4 requests 1A resource and 1C resource (1,0,1).

1. if the request is  $>$  Need(thread), or  $>$  Available, deny it.
2. Pretend the request is accepted.
3. run the algorithm to know if the state is safe : Work =(0,0,1)

Thread 2 has needs  $\leq$  Work : Work = (0,2,1).

Thread 0 has needs  $\leq$  Work : Work = (1,2,2).

Thread 3 has needs  $\leq$  Work : Work = (4,2,3).

Thread 4 has needs  $\leq$  Work : Work = (5,2,3).

Thread 1 has needs  $\leq$  Work : Work = (5,2,4).

So now we get again :

Thread	Finished
0	t
1	t
2	t
3	t
4	t

4. the request is granted

Running time of the safe state algorithm is  $n^2 * m$  where n is the number of threads and m the number of resource types  $\Rightarrow$  Costly, repeated for each allocation request.

Global method : you compare the available vector with the needs vectors. If there is any need vector which is lower than available, the threads order will start with this one, and will release his allocated resources. Therefore, Thread order += Thread X and Available += Allocation of X Then, you repeat until all threads are ordered or when you are stuck. All states will be safe or one won't be.

## 4 Detecting Deadlocks vs Preventing them

Let run the system without doing anything and look for a deadlock periodically. Using the resource allocation graph, it's the same algorithm to prevent or detect : find a cycle in the graph. using Allocation matrix : we take an optimistic approach, we assume that thread don't need any more resources => we replace the "Need" Matrix with pending requests, then use the safe state detection to find out if there is a deadlock

If a deadlock is detected :

- preempt resources – > not always possible.
- kill some of the threads :
  - the minimal number of threads that restores a safe state ?  
=> risk of killing important threads ( ex : Window manager, command interpreter...)
  - killing threads that are not related to the system.
  - let the user decide.
  - choose at random...

In most systems, deadlocks are not handled at all. One can find them in :

- some experimental OS
- some debugging environments