MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Deadlock (Part 2)

Daniel Playne
d.p.playne@massey.ac.nz

Deadlock - Model

The four necessary conditions for deadlock to occur are:

Deadlock - Model

The four necessary conditions for deadlock to occur are:

- **Mutual Exclusion** at least one resource must be held in a non-sharable mode (only one process at a time can use the resource). If another process requests the resource then it must wait until the resource is released.
- **Hold and Wait** a process must be holding at least one resource and waiting to acquire additional resources currently held by other processes.
- **No preemption** processes cannot be preempted (resources are only released when the process holding it voluntarily releases it.
- **Circular Wait** a set of processes $\{P_0, P_1, ...P_n\}$ must exist such that each process P_i is waiting for P_{i+1} and P_n is waiting for P_0.

Resource Allocation Graph

In general, if there is a cycle in a resource-allocation graph then the system **may** be in a deadlocked state.

If each resource has only **one** instance then the presence of a cycle means the system is in a deadlocked state.

If there is no cycle in the resource-allocation graph then the system is not deadlocked.

Deadlock Prevention

One method of ensuring a system does not end up deadlocked is to use **Deadlock Prevention** that changes the way that resources are allocated to processes.

These methods are designed to prevent one of the four necessary conditions for deadlock.

Unfortunately the approaches for deadlock-prevention often reduce device utilisation and system throughput.

Deadlock Avoidance

Another approach is called **deadlock avoidance** which allows the four conditions of deadlock to hold but before granting a a request for a resource, the system will check that this will not lead to deadlock.

To do this, the system requires additional information about what resources a process may request to decide whether granting a particular request may lead to possible deadlock in the future.

Deadlock Avoidance

A system is in a **safe state** if there is some possible order in which the system can allocate resources to processes and still avoid deadlock.

There is a **safe sequence** in which the system can allocate resources such that the processes can complete their tasks without getting stuck in deadlock.

If no such sequence exists then the system state is **unsafe**.

Deadlock Avoidance

An **unsafe** state does not necessarily mean the same thing as deadlock as the processes may release resources before requesting access to others.

However, as the behaviour of processes is unpredictable then the processes may request access to all the other resources and then the system will become deadlocked.

Deadlock avoidance ensures that the system is never in an unsafe state.

Deadlock Avoidance

Processes may potentially request a large range of different resources but may only use them very occasionally or just one at a time.

This is especially common with error handling code. When a process detects an error it may need to open a log file and record the error.

The log file is a potential resource the process needs to access but will only ever need to occasionally when something goes wrong.

Banker's Algorithm

The **banker's algorithm** is a deadlock avoidance algorithm that ensures that the system can never get into an unsafe state.

When a process enters the system, it must declare the maximum number of instances of each resource type that it might need.

This may not exceed the maximum number of instances of that resource available.

Banker's Algorithm

The algorithm is based on the idea of a banker lending out money to different borrowers.

In this scenario there is a banker (the Operating System), money (the resource) and borrowers (the processes).

Each borrower may need to borrow up to a certain maximum limit, at which point they can complete their projects and pay back their loan to the banker.

Banker's Algorithm

Each borrower must state at the beginning of their project what the maximum they may need to borrow will be (they may not actually need to borrow this whole amount but will never need more).

Name	Borrowed	Max
Donald	$0	$60,000
Angela	$0	$50,000
Boris	$0	$40,000
Emmanuel	$0	$70,000
Banker	Available	$100,000

Banker's Algorithm

First an obvious thing - borrowers can never borrow more money than the banker has in total (this is a very nice banker who doesn't charge any interest on loans).

When a borrower makes a request for money, the banker does not have to grant it immediately. The borrower cannot continue until the request is granted but is quite happy to wait until the banker lends them the money.

Banker's Algorithm

The system will be deadlocked if all the borrowers have requested to borrow money and the banker does not have enough for any of them.

Name	Borrowed	Max	Requested
Donald	\$30,000	\$60,000	\$20,000
Angela	\$20,000	\$50,000	\$15,000
Boris	\$15,000	\$40,000	\$25,000
Emmanuel	\$25,000	\$70,000	\$40,000
Banker	Available	\$10,000	

The banker cannot grant any requests so the borrowers cannot complete their projects and thus never pay anything back.

Banker's Algorithm

The following state is a **safe** state because (even if all borrowers request the maximum loan) the banker will still be able to lend them the money (in a certain order).

Name	Borrowed	Max	Requested
Donald	\$10,000	\$60,000	
Angela	\$10,000	\$50,000	
Boris	\$20,000	\$40,000	
Emmanuel	\$40,000	\$70,000	
Banker	Available	\$20,000	

Banker's Algorithm

The following state is not deadlocked but is **unsafe** because if the borrowers all ask to borrow the maximum amount of money, the banker will be unable to grant their requests.

Name	Borrowed	Max	Requested
Donald	$10,000	$60,000	
Angela	$20,000	$50,000	
Boris	$20,000	$40,000	
Emmanuel	$40,000	$70,000	
Banker	Available	$10,000	

Banker's Algorithm

The banker's algorithm works by making sure the system never enters an ***unsafe*** state.

Whenever it receives a request from a borrower, it will look to see whether granting that request would put the system into an unsafe state.

If it would, it will make the borrower wait until another borrower pays back their loan.

Banker's Algorithm

The complete Banker's Algorithm works with multiple resources rather than just one resource with a lot of instances (money).

To keep the analogy going you could think of a bank lending out different currencies but these will obviously be different resources that processes may request.

We will consider the banker's algorithm with multiple resource types.

Deadlock Avoidance - Banker's Algorithm

In this algorithm, several values must be tracked. If the processes are $P_1, P_2, ...P_n$ and the resources are $R_1, R_2, ...R_m$.

- **Available** - the number of available instances of each resource type.
- **Max** - the maximum number of instances of each resource type that a process may request.
- **Allocation** - the number of instances of each resource type currently allocated to each process.
- **Need** - the remaining number of instances of each resource type that a process may need.

Deadlock Avoidance - Banker's Algorithm

Given these definitions we can define an algorithm for determining whether a given state is safe or not.

1. Define **Work** where **Work[i]** $=$ **Available[i]**.
2. Define **Finish** where **Finish[i]** $=$ **false**.
3. Find an index **i** such that **Finish[i]** $==$ **false** and **Need[i][j]** $\leq$ **Work[j]**. If no index exists then go to step 5.
4. Set **Work[j]** $=$ **Work[j]** $+$ **Allocation[i][j]**
   Finish[i] $=$ **true**
   Go to step 3.
5. If **Finish[i]** $==$ **true** for all **i** then the system is safe.

Deadlock Avoidance - Banker's Algorithm

Essentially this algorithm tries to find an process that could have all the resources it needs fulfilled by the currently available resources. This process could be allocated these resources which would allow it to complete and release all its resources.

It will then try to find another process (and so on). If all processes are able to finish then there is a safe sequence and the state is safe. If it is unable to find a process that could have its needed resources allocated then the system is in an unsafe state.

Deadlock Avoidance - Banker's Algorithm

Example - five processes ($P_1 ... P_5$) and three resource types A (10 instances), B (5 instances) and C (7 instances). At a certain point in time the resource allocation of the system is as follows:

	Allocation	**Max**	**Available**
	A B C	A B C	A B C
P_1	0 1 0	7 5 3	3 3 2
P_2	2 0 0	3 2 2	
P_3	3 0 2	9 0 2	
P_4	2 1 1	2 2 2	
P_5	0 0 2	4 3 3	

Deadlock Avoidance - Banker's Algorithm

Question - is this system safe?

	Allocation	Max	Available
	A B C	A B C	A B C
P_1	0 1 0	7 5 3	3 3 2
P_2	2 0 0	3 2 2	
P_3	3 0 2	9 0 2	
P_4	2 1 1	2 2 2	
P_5	0 0 2	4 3 3	

Deadlock Avoidance - Banker's Algorithm

Calculate **Need**, **Work** and **Finish**

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
P_1	0 1 0	7 4 3	3 3 2	false
P_2	2 0 0	1 2 2		false
P_3	3 0 2	6 0 0	**Work**	false
P_4	2 1 1	0 1 1	A B C	false
P_5	0 0 2	4 3 1	3 3 2	false

Deadlock Avoidance - Banker's Algorithm

Find an index **i** such that **Finish[i]** $==$ **false** and **Need[i][j]** $\leq$ **Work[j]** (**i = 2** works).

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
P_1	0 1 0	7 4 3	3 3 2	false
$\underline{\mathbf{P_2}}$	**2 0 0**	**1 2 2**		false
P_3	3 0 2	6 0 0	**Work**	false
P_4	2 1 1	0 1 1	A B C	false
P_5	0 0 2	4 3 1	**3 3 2**	false

Deadlock Avoidance - Banker's Algorithm

Set **Work[j]** $=$ **Work[j]** $+$ **Allocation[i][j]** and **Finish[i]** $=$ **true**

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
P_1	0 1 0	7 4 3	3 3 2	false
$\cancel{P_2}$	~~0 0 0~~	~~1 2 2~~		~~**true**~~
P_3	3 0 2	6 0 0	**Work**	false
P_4	2 1 1	0 1 1	A B C	false
P_5	0 0 2	4 3 1	**5 3 2**	false

Deadlock Avoidance - Banker's Algorithm

and repeat...

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
P_1	0 1 0	7 4 3	3 3 2	false
~~P_2~~	~~0 0 0~~	~~1 2 2~~		~~**true**~~
P_3	3 0 2	6 0 0	**Work**	false
$\underline{\mathbf{P_4}}$	**2 1 1**	**0 1 1**	A B C	false
P_5	0 0 2	4 3 1	**5 3 2**	false

Deadlock Avoidance - Banker's Algorithm

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
P_1	0 1 0	7 4 3	3 3 2	false
$\cancel{P_2}$	~~0 0 0~~	~~1 2 2~~		~~**true**~~
P_3	3 0 2	6 0 0	**Work**	false
$\cancel{P_4}$	~~2 1 1~~	~~0 1 1~~	A B C	~~**true**~~
$\underline{\mathbf{P_5}}$	**0 0 2**	**4 3 1**	**7 4 3**	false

Deadlock Avoidance - Banker's Algorithm

	Allocation	Need	Available	Finish
	A B C	A B C	A B C	
$\underline{\mathbf{P_1}}$	**0 1 0**	**7 4 3**	3 3 2	false
$\cancel{P_2}$	~~0 0 0~~	~~1 2 2~~		~~**true**~~
P_3	3 0 2	6 0 0	**Work**	false
$\cancel{P_4}$	~~2 1 1~~	~~0 1 1~~	A B C	~~**true**~~
$\cancel{P_5}$	~~0 0 0~~	~~4 3 1~~	**7 4 5**	~~**true**~~

Deadlock Avoidance - Banker's Algorithm

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
~~P_1~~	~~0 0 0~~	~~7 4 3~~	3 3 2	~~**true**~~
~~P_2~~	~~0 0 0~~	~~1 2 2~~		~~**true**~~
$\underline{\mathbf{P_3}}$	**3 0 2**	**6 0 0**	**Work**	false
~~P_4~~	~~2 1 1~~	~~0 1 1~~	A B C	~~**true**~~
~~P_5~~	~~0 0 0~~	~~4 3 1~~	**7 5 5**	~~**true**~~

Deadlock Avoidance - Banker's Algorithm

Finally check whether **Finish[i]** is true for all **i**. It is so the state is safe.

	Allocation	**Need**	**Available**	**Finish**
	A B C	A B C	A B C	
$\cancel{P_1}$	~~0 0 0~~	~~7 4 3~~	3 3 2	~~**true**~~
$\cancel{P_2}$	~~0 0 0~~	~~1 2 2~~		~~**true**~~
$\cancel{P_3}$	~~0 0 0~~	~~6 0 0~~	**Work**	~~**true**~~
$\cancel{P_4}$	~~2 1 1~~	~~0 1 1~~	A B C	~~**true**~~
$\cancel{P_5}$	~~0 0 0~~	~~4 3 1~~	**10 5 7**	~~**true**~~

Deadlock Avoidance - Banker's Algorithm

1. If **Request[i][j]** $\leq$ **Need[j]** go to step 2. Otherwise raise an error (process has exceeded maximum claim).
2. If **Request[i][j]** $\leq$ **Available[j]**, go to step 3. Otherwise process must wait as requested resources are not available.
3. System calculates the new state that would be reached by granting the request as follows:
   Available[j] $=$ **Available[j] - Request[i][j]**
   Allocation[i][j] $=$ **Allocation[i][j]** $+$ **Request[i][j]**
   Need[i][j] $=$ **Need[i][j] - Request[i][j]**
   If the resulting state is safe then grant the request. Otherwise P_i must wait (old resource-allocation state is restored).

Deadlock Avoidance - Banker's Algorithm

Example: given the following resource allocation (that we determined was safe last time) should the following request be granted?

Request[1] $= \{1, 1, 0\}$

	Allocation	Need	Available
	A B C	A B C	A B C
P_1	0 1 0	7 4 3	3 3 2
P_2	2 0 0	1 2 2	
P_3	3 0 2	6 0 0	
P_4	2 1 1	0 1 1	
P_5	0 0 2	4 3 1	

Deadlock Avoidance - Banker's Algorithm

- Is **Request[i]** $\leq$ **Need** (yes)
- Is **Request[i]** $\leq$ **Available** (yes)
- Calculate new state if the request were granted:

	Allocation	**Need**	**Available**
	A B C	A B C	A B C
P_1	1 2 0	6 3 3	2 2 2
P_2	2 0 0	1 2 2	
P_3	3 0 2	6 0 0	
P_4	2 1 1	0 1 1	
P_5	0 0 2	4 3 1	

Deadlock Avoidance - Banker's Algorithm

Is this new state safe?

	Allocation	**Need**	**Available**
	A B C	A B C	A B C
P_1	1 2 0	6 3 3	2 2 2
P_2	2 0 0	1 2 2	
P_3	3 0 2	6 0 0	
P_4	2 1 1	0 1 1	
P_5	0 0 2	4 3 1	

Deadlock Avoidance - Banker's Algorithm

Is this new state safe?

	Allocation	Need	Available
	A B C	A B C	A B C
P_1	1 2 0	6 3 3	2 2 2
P_2	2 0 0	1 2 2	
P_3	3 0 2	6 0 0	
P_4	2 1 1	0 1 1	
P_5	0 0 2	4 3 1	

Yes, could use the sequence P_4, P_2, P_5, P_3 and P_1.

Deadlock Avoidance - Banker's Algorithm

Example: what if the request was:

Request[1] $= \{1, 3, 0\}$

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	4	3	3	3	2
P_2	2	0	0	1	2	2			
P_3	3	0	2	6	0	0			
P_4	2	1	1	0	1	1			
P_5	0	0	2	4	3	1			

Deadlock Avoidance - Banker's Algorithm

- Is **Request[i]** $\leq$ **Need** (yes)
- Is **Request[i]** $\leq$ **Available** (yes)
- Calculate new state if the request were granted:

	Allocation	**Need**	**Available**
	A B C	A B C	A B C
P_1	1 4 0	6 1 3	2 0 2
P_2	2 0 0	1 2 2	
P_3	3 0 2	6 0 0	
P_4	2 1 1	0 1 1	
P_5	0 0 2	4 3 1	

Deadlock Avoidance - Banker's Algorithm

Is this new state safe?

	Allocation	Need	Available
	A B C	A B C	A B C
P_1	1 4 0	6 1 3	2 0 2
P_2	2 0 0	1 2 2	
P_3	3 0 2	6 0 0	
P_4	2 1 1	0 1 1	
P_5	0 0 2	4 3 1	

No, there is no process that can have its requirements met so the request should be denied (process must wait).

Summary

- Handling Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Banker's Algorithm

Chapters:
Silberschatz - chapter 7.4, 7.5

Tanenbaum - chapter 6.4, 6.5, 6.6