Deadlock

COMP3231 Operating Systems

2003 S2

Deadlock

What is a **deadlock**?

DEADLOCK

What is a **deadlock**?

➜ Permanent blocking of a set of processes that either

- compete for system resources or
- communicate with each other (message as resource)

DEADLOCK

What is a **deadlock**?

- ➜ Permanent blocking of a set of processes that either
  - compete for system resources or
  - communicate with each other (message as resource)
- ➜ Resources:
  - preemptable
  - nonpreemtable resources

DEADLOCK

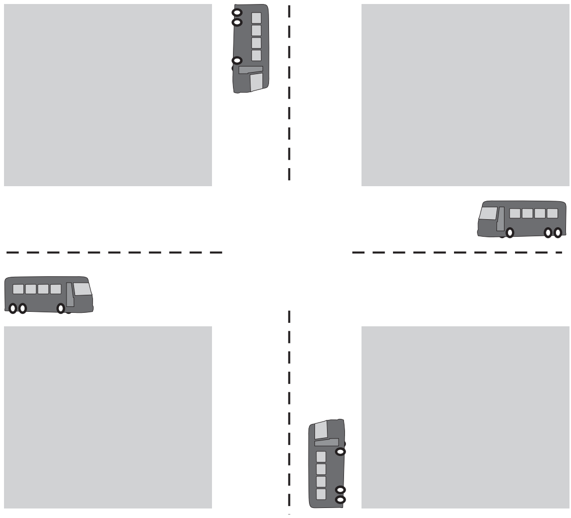
What is a **deadlock**?

- ➜ Permanent blocking of a set of processes that either
  - compete for system resources or
  - communicate with each other (message as resource)
- ➜ Resources:
  - preemptable
  - nonpreemtable resources
- ➜ Deadlocks involve conflicting needs for nonpreemtable resources by two or more processes
- ➜ Deadlocks can occur on many levels in the system

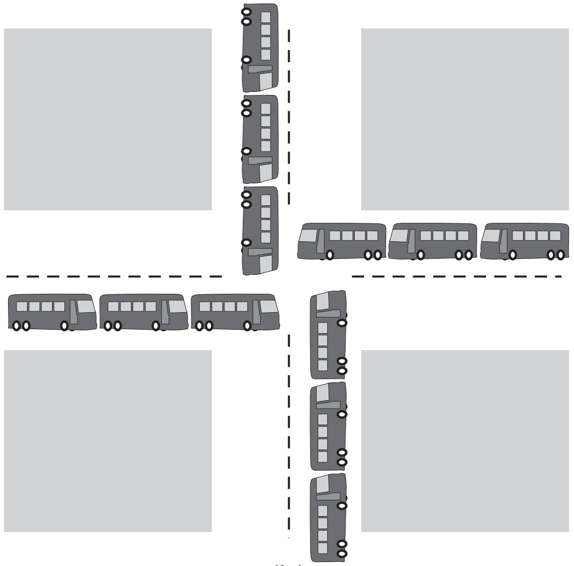
- ✘ Unfortunately, there is no efficient method to prevent deadlock in the general case

Let's look at some examples and at the conditions for deadlock

Danger of deadlock in continental driving rules:



(a)



(b)

Reusable Versus Consumable Resources

- Reusable resource: used by one process at a time and not depleted by that use
- Consumable resource: created (produced) and destroyed (consumed) by a process

Reusable Resources:

- Processes obtain resources that they later release for reuse by other processes
- Examples are processors, I/O channels, main and secondary memory, files, databases, and semaphores
- In case of two processes and two resources, deadlock occurs if each process holds one resource and requests the other

Typical deadlock with reusable resources:

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

The following sequence leads to a deadlock:

$$p_0, p_1, q_0, q_1, p_2, q_2$$

Typical deadlock with reusable resources:

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

The following sequence leads to a deadlock:

$$p_0, p_1, q_0, q_1, p_2, q_2$$

- Should this really be the problem of the OS designer?

Another example of deadlock with reusable resources:

➜ Space is available for allocation of 200K bytes and the following sequence of events occur

P$_1$	**P**$_2$
...	**...**
Request 80kB;	**Request 70kB;**
...	**...**
Request 60kB;	**Request 80kB;**

Another example of deadlock with reusable resources:

- ➜ Space is available for allocation of 200K bytes and the following sequence of events occur

$\mathbf{P}_1$	$\mathbf{P}_2$
...	**...**
Request 80kB;	**Request 70kB;**
...	**...**
Request 60kB;	**Request 80kB;**

- ➜ Deadlock occurs if both processes progress to their second request

Another example of deadlock with reusable resources:

- Space is available for allocation of 200K bytes and the following sequence of events occur

P_1	**P_2**
...	**...**
Request 80kB;	**Request 70kB;**
...	**...**
Request 60kB;	**Request 80kB;**

- Deadlock occurs if both processes progress to their second request
- In this case, the problem can be solved by using virtual memory (this is an example of resource preemption)

Consumable Resources:

- ➜ Interrupts, signals, messages, and information in I/O buffers
- ➜ Deadlock may occur if a Receive message is blocking
- ➜ May take a rare combination of events to cause deadlock

Example of deadlock:

- ➜ Deadlock occurs if receive is blocking

P_1	**P_2**
...	**...**
Receive(P_2);	**Receive(P_1);**
...	**...**
Send(P_2, M_1);	**Send(P_1, M_2);**

Conditions for Deadlock

How can we accurately characterise the conditions that lead to a deadlock?

Necessary conditions for deadlock:

Conditions for Deadlock

How can we accurately characterise the conditions that lead to a deadlock?

Necessary conditions for deadlock:

① Mutual exclusion: only one process may use a resource at a time

Conditions for Deadlock

How can we accurately characterise the conditions that lead to a deadlock?

Necessary conditions for deadlock:

① Mutual exclusion: only one process may use a resource at a time

② Hold-and-wait: a process holds a resource while awaiting assignment of others

Conditions for Deadlock

How can we accurately characterise the conditions that lead to a deadlock?

Necessary conditions for deadlock:

① Mutual exclusion: only one process may use a resource at a time

② Hold-and-wait: a process holds a resource while awaiting assignment of others

③ No preemption of resources:

- A process that is denied a request must not release the resources it already has
- When one process requests a resource held by another, the second one is not preempted by the OS

CONDITIONS FOR DEADLOCK

How can we accurately characterise the conditions
that lead to a deadlock?

Necessary conditions for deadlock:

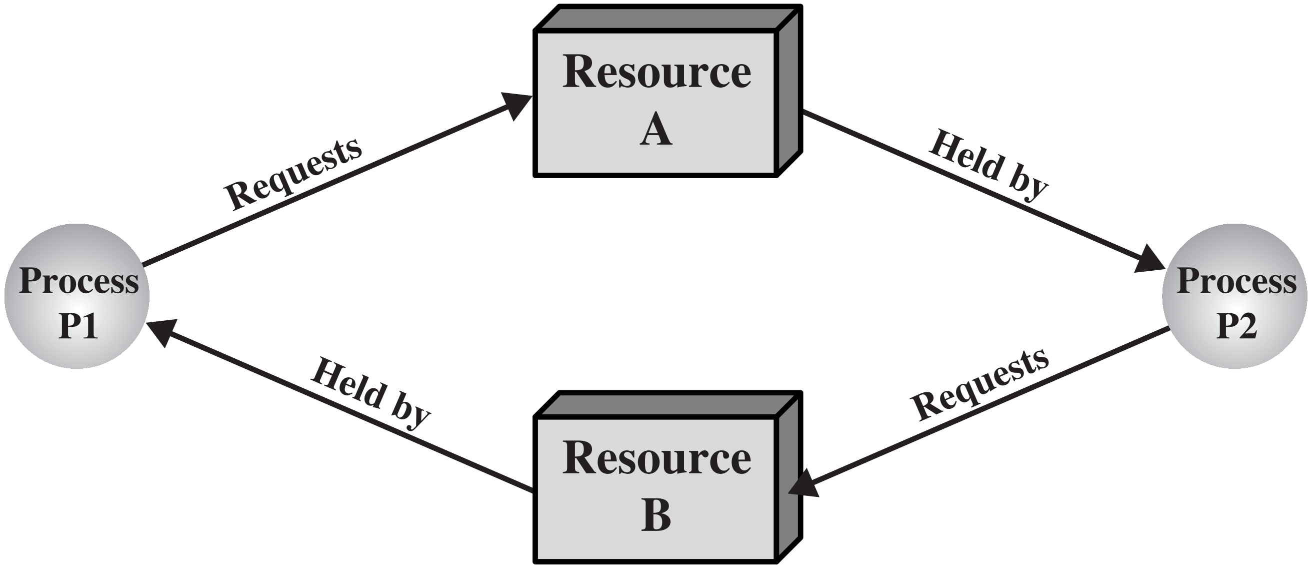
① Mutual exclusion: only one process may use a resource at a time

② Hold-and-wait: a process holds a resource while awaiting assignment of others

③ No preemption of resources:

- A process that is denied a request must not release the resources it already has
- When one process requests a resource held by another, the second one is not preempted by the OS

④ Circular wait: we have a closed chain of processes, such that each process holds at least one resource needed by the next in the chain, e.g.,


Resource
A
Requests
Held by
Process
P1
Process
P2
Held by
Requests
Resource
B

Strategies to deal with Deadlocks

1. The Ostrich Algorithm
2. Prevention
3. Avoidance by careful resource allocation
4. Detection and Recovery: let then occur, detect them and take action

The Ostrich Algorithm:

STRATEGIES TO DEAL WITH DEADLOCKS

① The Ostrich Algorithm

② Prevention

③ Avoidance by careful resource allocation

④ Detection and Recovery: let then occur, detect them and take action

The Ostrich Algorithm:

Stick your head in the sand and pretend there is no problem at all!

STRATEGIES TO DEAL WITH DEADLOCKS

① The Ostrich Algorithm

② Prevention

③ Avoidance by careful resource allocation

④ Detection and Recovery: let then occur, detect them and take action

The Ostrich Algorithm:

Stick your head in the sand and pretend there is no problem at all!

➜ Unix & Windows

➜ Avoid deadlock in the kernel!

Deadlock Prevention

What is deadlock prevention?

Make it impossible that one of the four conditions for deadlock arise

① mutual exclusion
② hold-and-wait
③ no preemption
④ circular wait

Deadlock Prevention

What is deadlock prevention?

Make it impossible that one of the four conditions for deadlock arise

① mutual exclusion
② hold-and-wait
③ no preemption
④ circular wait

Mutual exclusion:

Deadlock Prevention

What is deadlock prevention?

Make it impossible that one of the four conditions for deadlock arise

① mutual exclusion
② hold-and-wait
③ no preemption
④ circular wait

Mutual exclusion:

➜ we can't generally exclude it

Deadlock Prevention

What is deadlock prevention?

Make it impossible that one of the four conditions for deadlock arise

① mutual exclusion

② hold-and-wait

③ no preemption

④ circular wait

Mutual exclusion:

- ➜ we can't generally exclude it
- ➜ we can avoid assigning resources when not absolutely necessary

Deadlock Prevention

What is deadlock prevention?

Make it impossible that one of the four conditions for deadlock arise

① mutual exclusion

② hold-and-wait

③ no preemption

④ circular wait

Mutual exclusion:

- ➜ we can't generally exclude it
- ➜ we can avoid assigning resources when not absolutely necessary
- ➜ as few processes as possible should claim the resource

Hold-and-wait:

➜ Can we require processes to request all resources at once?

Hold-and-wait:

- ➜ Can we require processes to request all resources at once?
- ➜ Most processes do not statically know about the resources they need

Hold-and-wait:

- ➜ Can we require processes to request all resources at once?
- ➜ Most processes do not statically know about the resources they need
- ➜ Used in some mainframe batch systems
- ➜ Wasteful, but works
- ➜ Variation: before requesting new resource, temporarily release other resources

No preemption:

No preemption:

Preemption is feasible for some resources (e.g., processor and memory), but not for others (state must be saved and restored)

No preemption:

Preemption is feasible for some resources (e.g., processor and memory), but not for others (state must be saved and restored)

Circular wait:

No preemption:

Preemption is feasible for some resources (e.g., processor and memory), but not for others (state must be saved and restored)

Circular wait:

- order resources by an index: $R_1, R_2, \ldots$
- requires that resources are always requested in order
- P_1 holds R_i and requests R_j, and P_2 holds R_j and requests R_i is impossible, as it implies

$$i < j \quad \text{and} \quad i > j$$

- is sometimes a feasible strategy, but not generally efficient

Deadlock Avoidance

What is deadlock avoidance?:

- ➜ We don't exclude any of the four conditions for deadlock per se
- ➜ Instead we decide on a per case basis whether a process is deemed likely to deadlock
- ➜ Thus, we have to possess some knowledge about future allocation requests of processes

DEADLOCK AVOIDANCE

What is deadlock avoidance?:

- We don't exclude any of the four conditions for deadlock per se
- Instead we decide on a per case basis whether a process is deemed likely to deadlock
- Thus, we have to possess some knowledge about future allocation requests of processes

Generally, we can distinguish two approaches to deadlock avoidance:

- Process initiation denial: we just don't start a process if it might deadlock
- Resource allocation denial: we deny allocation requests, which are likely to lead to deadlock in the future

Process Initiation Denial

Consider a system of n processes and m types of resources:

- Resource vector: $(R_1, R_2, \ldots, R_n)$
- Available vector: $(V_1, V_2, \ldots, V_n)$
- Matrices:

Claim matrix:

$$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$$

Allocation matrix:

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

- C_{ij} requirement for process i for resource j
- A_{ij} allocation of resource j to process i

The following relationships hold:

① $R_i = V_i + \sum_{k=1}^{n} A_{ki}$: all resources are either available or allocated

② $C_{kj} \leq R_i$: no process can hold more than the total amount of resources in the system

③ $A_{kj} \leq C_{ki}$: no process is allocated more than it originally claimed to need

The following relationships hold:

① $R_i = V_i + \sum_{k=1}^{n} A_{ki}$: all resources are either available or allocated

② $C_{kj} \leq R_i$: no process can hold more than the total amount of resources in the system

③ $A_{kj} \leq C_{ki}$: no process is allocated more than it originally claimed to need

Deadlock avoidance policy:

- Start a new process P_{n+1} only if, for all i,

$$V_i \geq C_{n+1,i} + \sum_{k=1}^{n} C_{ki}$$

- Unfortunately, this strategy is very wasteful!
- Assumes all processes make their claims together

Resource Allocation Denial

- ➜ At any request of a resource, it is tested whether granting this request bears the potential of deadlock
- ➜ The standard algorithm to execute this test is due to Dijkstra and known as the banker's algorithm

RESOURCE ALLOCATION DENIAL

- ➜ At any request of a resource, it is tested whether granting this request bears the potential of deadlock
- ➜ The standard algorithm to execute this test is due to Dijkstra and known as the banker's algorithm

Banker's algorithm:

- ➜ Resource and available vector & claim and allocation matrix as before
- ➜ The algorithm passes out resources to processes if it has enough on hand to meet potential future demand
- ➜ Whenever we can guarantee that future demand can be met, we are in a safe state
- ➜ A request for resources is granted only if the state after the resource is granted is safe

How do we know whether a state is safe?

- ➜ A state is safe if there is at least one sequence of resource allocations that does not result in deadlock
- ➜ Pick a process whose outstanding resource claim can be met and run it to completion
- ➜ Repeat until either all process have completed, or the system locks up

Check that this state is safe:

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

Check that this state is safe:

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Resource Vector

R1	R2	R3
9	3	6

Available Vector

R1	R2	R3
0	1	1

P2 runs to completion:

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Allocation Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Available Vector

R1	R2	R3
6	2	3

P1 Runs to Completion:

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

P1 Runs to Completion:

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Allocation Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Available Vector

R1	R2	R3
7	2	3

P3 Runs to Completion:

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Allocation Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Available Vector

R1	R2	R3
9	3	4

Example of a request leading to an unsafe state:

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
1	1	2

Available Vector

P1 requests R1 & R3:

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
0	1	1

Available Vector

Disadvantages of the Banker's algorithm:

- ➜ Maximum resource requirement must be stated in advance
- ➜ Processes under consideration must be independent; no synchronization requirements
- ➜ There must be a fixed number of resources to allocate
- ➜ No process may exit while holding resources

Deadlock Detection

- ➜ An alternative to deadlock avoidance is deadlock detection

Deadlock Detection

- ➜ An alternative to deadlock avoidance is deadlock detection
- ➜ However, for this to be useful, we require to be able to either
  - roll processes back (in the extreme case, kill them) or
  - preempt resources

DEADLOCK DETECTION

- ➜ An alternative to deadlock avoidance is deadlock detection
- ➜ However, for this to be useful, we require to be able to either
  - roll processes back (in the extreme case, kill them) or
  - preempt resources

Modification of Banker's algorithm for deadlock detection:

- ➜ We need a request matrix Q (oustanding requests) instead of the claim matrix
- ➜ Disregard processes without any allocation (not holding resources)
- ➜ Consider process completed if outstanding requests are satisfied
- ➜ Checks can be made each time a resource is allocated
  - early deadlock detection
  - expensive

Algorithm:

Initially, all processes are unmarked

① mark each process with zero-row in Allocation matrix

② set temporary vector W to Available vector

③ find i such that process i is unmarked, $Q_{ik} \leq W_k$ for $1 \leq k \leq n$
  - no such process $\Rightarrow$ terminate

④ mark process i, add row of allocation matrix to W, go to step 3

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request Matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation Matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available Vector

Recovery:

① Abort all deadlocked processes (most common solution)

② Rollback each deadlocked process to some previously defined checkpoint and restart them (original deadlock may reoccur)

③ Successively abort deadlocked processes until deadlock no longer exists (invoke deadlock detection algorithm each time)

④ Successively preempt some resources from process until deadlock no longer exists

- a process that has a resource preempted must be rolled back prior to its acquisition