Liveness properties

From a theoretical viewpoint we must ensure that we eventually make progress

i.e. we want to avoid

Deadlock: blocked threads/processes waiting for each other

Livelock: processes/threads execute but make no progress

From a practical viewpoint we also want good performance:

No starvation – a single thread must make progress

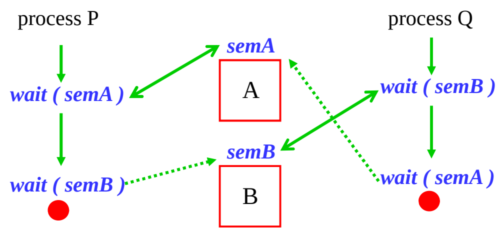
more generally, may aim for fairness

Minimality – no unnecessary waiting or signalling

Deadlock

Real-life example (Kansas, 1920s) *"When two trains approach each other at a crossing, both shall stop and neither shall start up again until the other has gone"*

In concurrent programs, deadlock tends to arise from taking mutual exclusion locks

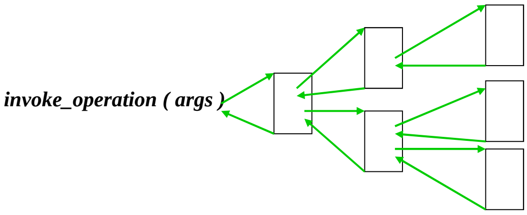


At this point ● we have deadlock. Process P holds ***semA*** and is blocked, queued on ***semB***
Process Q holds ***semB*** and is blocked, queued on ***semA***

Neither process can proceed to use both A and B and signal the respective semaphores.

A **cycle** of processes exists, where each holds one resource and is blocked waiting for another, held by another process in the cycle.

Concurrent composite operations



We can make one operation on shared data atomic (Lectures 2,3).

Composite operations seem to be causing the deadlock - can we avoid them?

No: the high-level, composite operation may be meaningful to the application

e.g. ***write***-to-file means get free block(s), change metadata, write data

e.g. ***transfer*** = ***subtract*** from one object then ***add*** to another object

e.g. add/remove an item to/from an ordered linked list – change multiple pointers

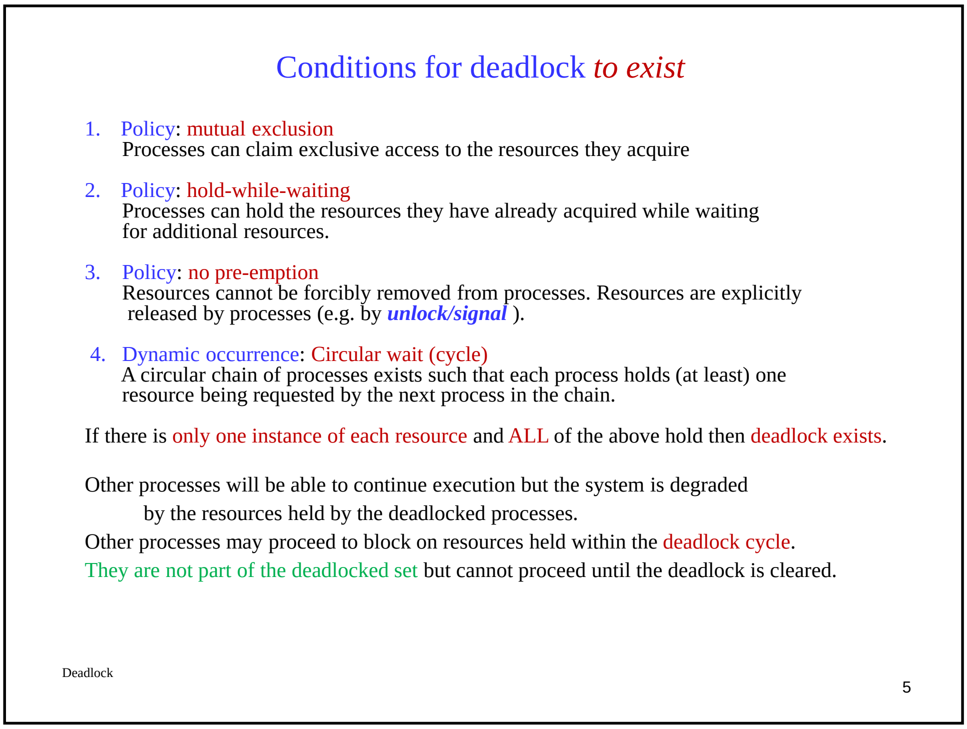
Transaction semantics mean that either ***all*** the sub-operations complete ***or none*** does
- in the presence of ***concurrency*** and ***crashes***.

We’ll develop this idea later and focus in this lecture on deadlock.

Deadlock - Outline

Systems that allocate multiple resources ***dynamically*** are subject to deadlock.
(e.g. dynamic order of locking multiple objects).

1. Background resource allocation policies that make deadlock possible,
and what events cause it to occur dynamically?
2. Deadlock prevention – discussion of the conditions for avoidance and recovery.
3. Example: Dining philosophers program – discussion of policies
4. Modelling deadlock – to support deadlock detection and avoidance.
    - object allocation, resource requests and cycle detection
    - data structures and an algorithm for deadlock detection



Conditions for deadlock *to exist*

1. Policy: mutual exclusion
   Processes can claim exclusive access to the resources they acquire

2. Policy: hold-while-waiting
   Processes can hold the resources they have already acquired while waiting for additional resources.

3. Policy: no pre-emption
   Resources cannot be forcibly removed from processes. Resources are explicitly released by processes (e.g. by ***unlock/signal***).

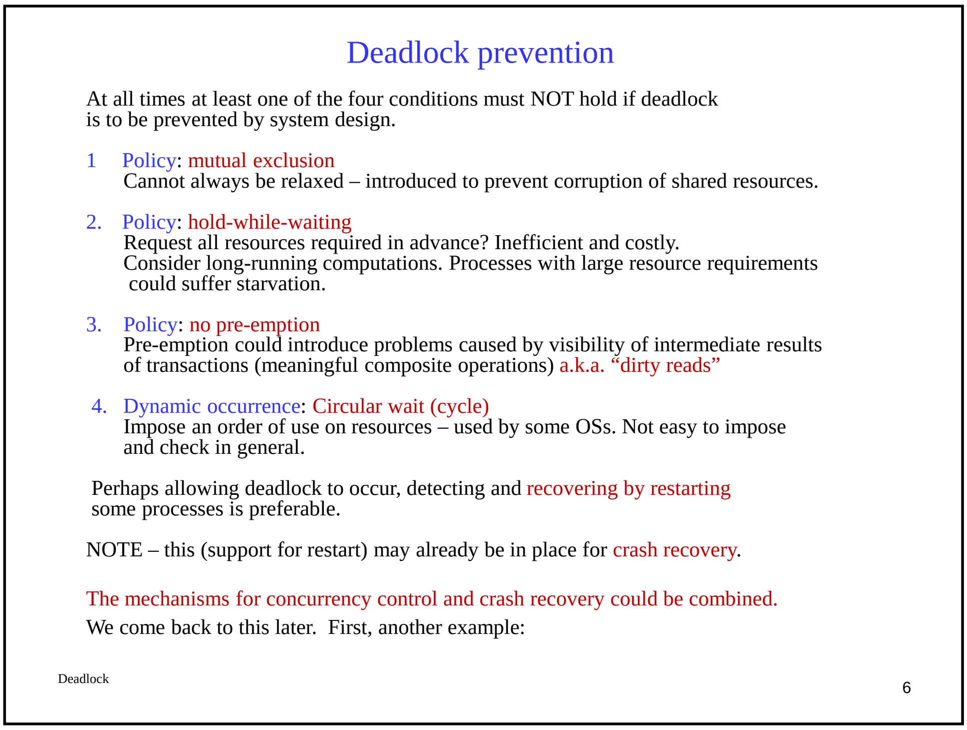
4. Dynamic occurrence: Circular wait (cycle)
   A circular chain of processes exists such that each process holds (at least) one resource being requested by the next process in the chain.

If there is only one instance of each resource and ALL of the above hold then deadlock exists.

Other processes will be able to continue execution but the system is degraded by the resources held by the deadlocked processes.

Other processes may proceed to block on resources held within the deadlock cycle.

They are not part of the deadlocked set but cannot proceed until the deadlock is cleared.



Deadlock prevention

At all times at least one of the four conditions must NOT hold if deadlock is to be prevented by system design.

1 Policy: mutual exclusion
  Cannot always be relaxed – introduced to prevent corruption of shared resources.

2. Policy: hold-while-waiting
   Request all resources required in advance? Inefficient and costly.
   Consider long-running computations. Processes with large resource requirements could suffer starvation.

3. Policy: no pre-emption
   Pre-emption could introduce problems caused by visibility of intermediate results of transactions (meaningful composite operations) a.k.a. “dirty reads”

4. Dynamic occurrence: Circular wait (cycle)
   Impose an order of use on resources – used by some OSs. Not easy to impose and check in general.

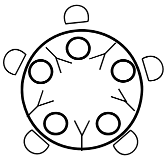
Perhaps allowing deadlock to occur, detecting and recovering by restarting some processes is preferable.

NOTE – this (support for restart) may already be in place for crash recovery.

The mechanisms for concurrency control and crash recovery could be combined.

We come back to this later. First, another example:

Dining philosophers (due to Dijkstra, 1965) - 1



Five philosophers spend their time thinking and eating. They each have a chair at a shared table with a shared bowl of food and shared forks – they need two forks to eat.
To eat they “execute” an identical algorithm –
pick up left fork, pick up right fork, eat, put down forks.

var fork : array [0 .. 4] of semaphore \\ all initialised to 1

philosopher i may then be specified as:

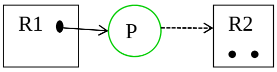
```
repeat
    wait ( fork [i] ) ;
    wait ( fork [i+1 mod 5 ] ) ;
    EAT
    signal ( fork [i] ) ;
    signal ( fork [i+1 mod 5 ] ) ;
    THINK
until false
```

Dining philosophers - 2

We have the policies in place for deadlock to be possible:
exclusive hold, hold-while-wait, no preemption.
Dynamically, deadlock can occur:
a cycle is created when the philosophers each acquire their left fork
and block waiting for their right fork.

The problem can be solved in a number of ways, essentially by ensuring
that at least one of the conditions necessary for deadlock to exist cannot hold
Breaking the symmetry of the algorithm can achieve this
e.g. make odd-numbered processes pick up their forks as specified, L then R,
and even ones pick up their forks in reverse order, R then L.

Object allocation and request – graphical notation



R1 and R2 are object/resource types. R1 has one instance and R2 has two.

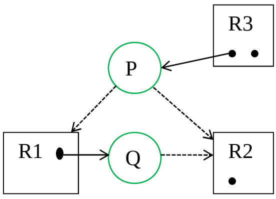
The directed edge from the single instance of R1 to process P indicates that P holds that resource.

The dashed directed edge from P to the object type R2 indicates that P has an outstanding request for an object of type R2. Assuming both are allocated, P is blocked, waiting for an R2.

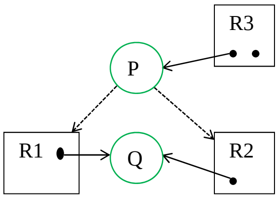
If a cycle exists in such a graph and there is only one instance of each of the types involved in the cycle, then deadlock exists (necessary and sufficient condition).

If there is more than one object of some or all of the types, then a cycle is a necessary but not a sufficient condition for deadlock to exist.

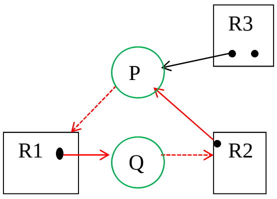
Dynamic object allocation and request – example



resource allocation must decide whether to give the R2 instance to P or to Q

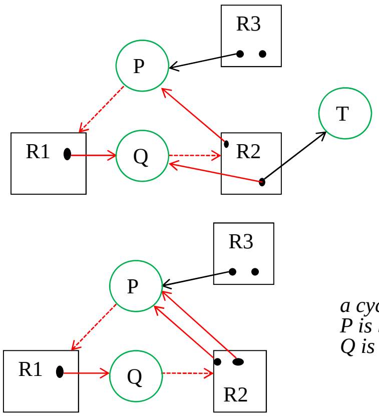


give the R2 instance to Q: no cycle
AFAWK, Q can complete and release R1 and R2, then P can have R1 and R2 and complete.
There may of course be further dynamic requests.



give the R2 instance to P: cycle = deadlock

Cycles without and with deadlock



a cycle exists, but no deadlock
T could release R2, and unblock Q

a cycle and deadlock
P is blocked waiting for R1
Q is blocked, waiting for R2

Data structures for resource/object allocation management

Allocation matrix A_{mn}

object/resource type →

$$\begin{array}{c|c|} & \\ \text{process} \downarrow & \begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix} \end{array}$$

a_{ij} is the number of objects of type j allocated to process i

Request matrix B_{mn}

object/resource type →

$$\begin{array}{c|c|} & \\ \text{process} \downarrow & \begin{matrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ & & & \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{matrix} \end{array}$$

b_{ij} is the number of objects of type j requested by process i

objects being managed: $R_n = (r_1 r_2 \ldots r_n)$, the number of type i is r_i

objects available: $V_n = (v_1 v_2 \ldots v_n)$, the number of type i is v_i,
computable from R_n minus the objects allocated

Algorithm for deadlock detection

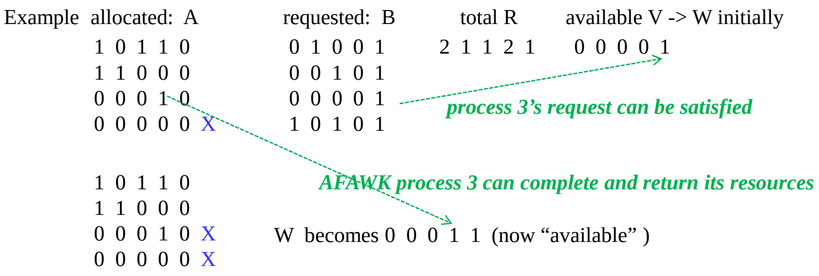
Mark the rows of the **allocation** matrix that are NOT part of a deadlocked set **NOW**

1. Mark all null rows of A (a process holding no resources cannot be part of a deadlocked set
2. Initialise a working vector W = V initially, the available objects
3. Search for an unmarked row, say row i, such that $B_i \leq W$
   i.e. the objects that process i is requesting are “available” in W.
   (note that a null request can always be satisfied)
   If none is found, terminate the algorithm.
4. Set $W = W + A_i$ and mark row i. Return to step 3.

Example **allocated**: A	**requested**: B	e.g. total R	available V ⟶W initially
1 0 1 1 0	0 1 0 0 1	2 1 1 2 1	0 0 0 0 1
1 1 0 0 0	0 0 1 0 1		
0 0 0 1 0	0 0 0 0 1		
0 0 0 0 0	1 0 1 0 1		

Algorithm for deadlock detection - example

3. Search for an unmarked row, say row i, such that $B_i \leq W$
   If none is found, terminate the algorithm.
4. Set $W = W + A_i$ and mark row i. Return to step 3.



Example allocated: A	requested: B	total R	available V -> W initially
1 0 1 1 0	0 1 0 0 1	2 1 1 2 1	0 0 0 0 1
1 1 0 0 0	0 0 1 0 1		
0 0 0 1 0	0 0 0 0 1		
0 0 0 0 0 X	1 0 1 0 1		

process 3’s request can be satisfied

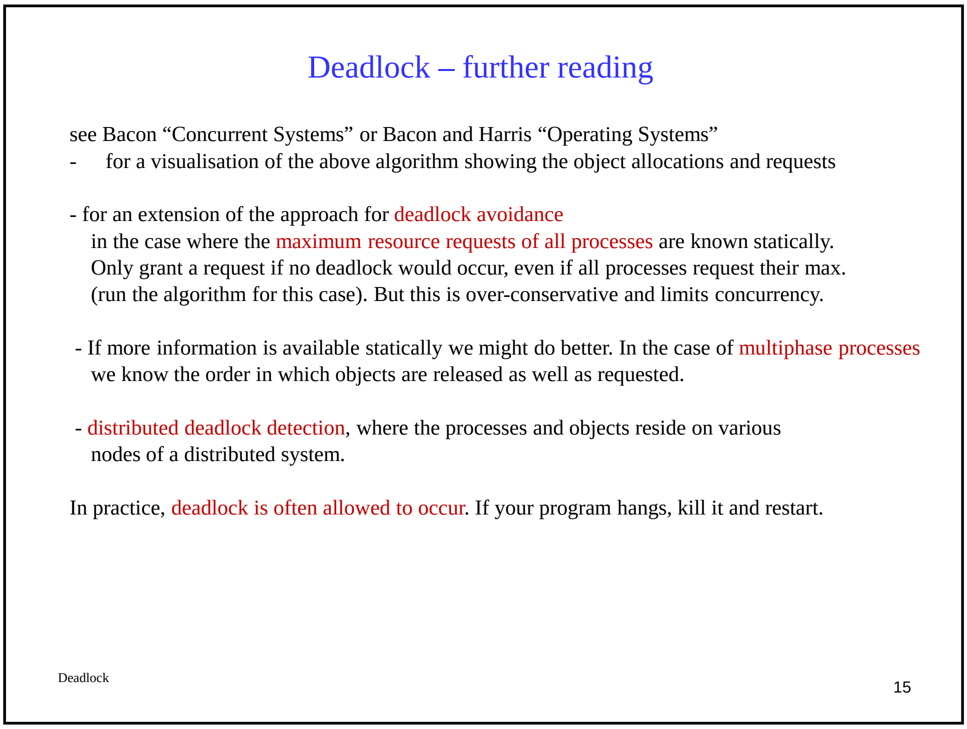
1 0 1 1 0
1 1 0 0 0
0 0 0 1 0 X
0 0 0 0 0 X

AFAWK process 3 can complete and return its resources

W becomes 0 0 0 1 1 (now “available”)

processes 1 and 2 are deadlocked over objects 2 and 3

1 0 1 1 0	0 1 0 0 1	
1 1 0 0 0	0 0 1 0 1	R = 2 1 1 2 1
0 0 0 1 0 X	0 0 0 0 1	W = 0 0 0 1 1
0 0 0 0 0 X	1 0 1 0 1	



Deadlock – further reading

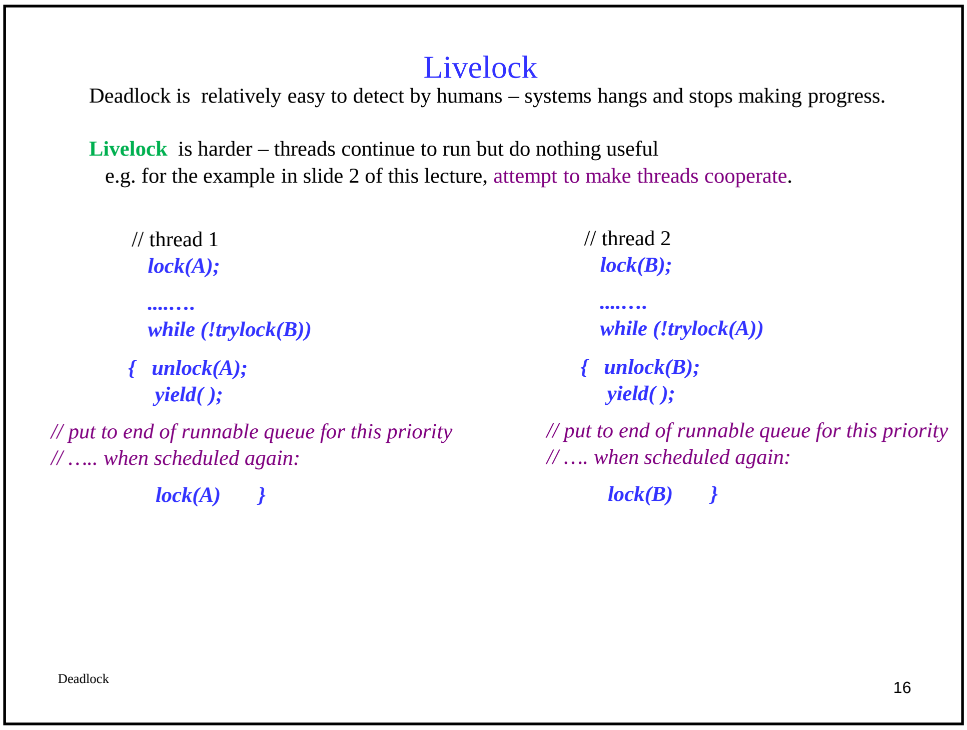
see Bacon "Concurrent Systems" or Bacon and Harris "Operating Systems"
- for a visualisation of the above algorithm showing the object allocations and requests

- for an extension of the approach for deadlock avoidance
in the case where the maximum resource requests of all processes are known statically.
Only grant a request if no deadlock would occur, even if all processes request their max.
(run the algorithm for this case). But this is over-conservative and limits concurrency.

- If more information is available statically we might do better. In the case of multiphase processes
we know the order in which objects are released as well as requested.

- distributed deadlock detection, where the processes and objects reside on various
nodes of a distributed system.

In practice, deadlock is often allowed to occur. If your program hangs, kill it and restart.



Livelock

Deadlock is relatively easy to detect by humans – systems hangs and stops making progress.

Livelock is harder – threads continue to run but do nothing useful
e.g. for the example in slide 2 of this lecture, attempt to make threads cooperate.

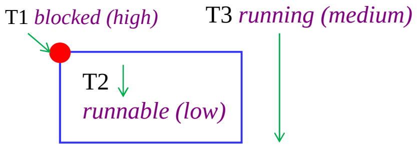
```
// thread 1
  lock(A);

  …..….
  while (!trylock(B))

{   unlock(A);
    yield( );

// put to end of runnable queue for this priority
// ….. when scheduled again:

    lock(A)      }
```

```
// thread 2
  lock(B);

  …..….
  while (!trylock(A))

{   unlock(B);
    yield( );

// put to end of runnable queue for this priority
// …. when scheduled again:

    lock(B)      }
```

Priority Inversion and Priority Inheritance

A liveness problem due to interaction between locking and scheduling.

e.g. Consider three threads T1 (high priority), T2 (low priority), T3 (medium priority)
suppose T2 acquires lock L
T1 preempts T2 and blocks, waiting for L
T3 is scheduled (as higher priority than T2)
T3 prevents T2 from running and releasing L



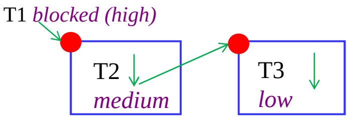
Typical solution is priority inheritance: temporary boost of priority of lock holder to that of highest waiting thread
- some realtime systems (VxWorks) allow this policy to be specified per lock

Windows “solution”
- if any ready thread hasn’t run for 300 ticks, double its quantum and boost its priority.

Problems with Priority Inheritance

Hard to reason about resulting behaviour: heuristic
but solutions based on some experience may not generalise – unintended side-effects …

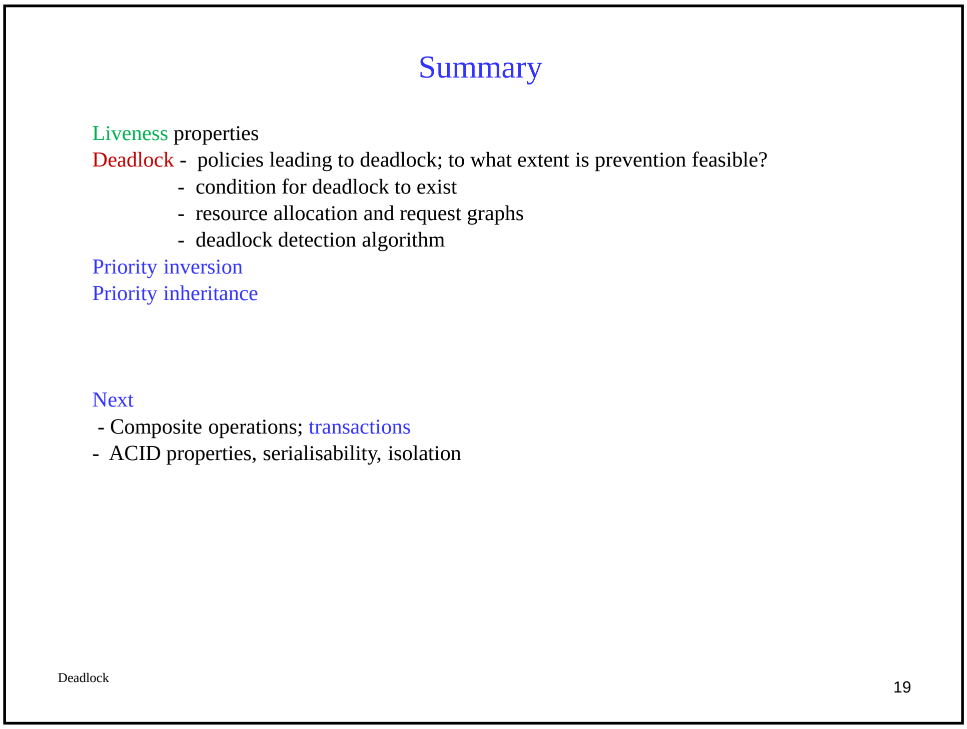
More complex in practice than applying to a single lock
- propagate through multiple locks?



- how to handle reader-writer locks (wish to give priority to writers)?

How to take into account condition synchronisation?
- with locks we know what thread holds the lock
- threads also block waiting for signals
- we don’t know which thread might issue a signal or release an allocated resource
- should we also record this e.g. in a condition queue associated with a mutex

Avoid the need for priority inheritance.
- avoid resource sharing between threads of different priorities.



Summary

Liveness properties

Deadlock - policies leading to deadlock; to what extent is prevention feasible?
- condition for deadlock to exist
- resource allocation and request graphs
- deadlock detection algorithm

Priority inversion

Priority inheritance

Next
- Composite operations; transactions
- ACID properties, serialisability, isolation