Deadlocks

(Chapter 3, Tanenbaum)

What is a *deadlock*?

Deadlock is defined as the *permanent* blocking of a set of processes that compete for system resources, including database records or communication lines.

Unlike other problems in multiprogramming systems, there is no efficient solution to the deadlock problem in the general case.

Deadlock **prevention**, **by design**, is the "best" solution.

Deadlock occurs when a set of processes are in a wait state, because each process is waiting for a resource that is held by some other waiting process. Therefore, all deadlocks involve conflicting resource needs by two or more processes.

Classification of resources—I

Two general categories of resources can be distinguished:

- **Reusable**: something that can be safely used by one process at a time and is not depleted by that use. Processes obtain resources that they later release for reuse by others.

  E.g., CPU, memory, specific I/O devices, or files.
- **Consumable**: these can be created and destroyed. When a resource is acquired by a process, the resource ceases to exist.

  E.g., interrupts, signals, or messages.

Classification of resources—II

One other taxonomy again identifies two types of resources:

- **Preemptable**: these can be taken away from the process owning it with no ill effects (needs save/restore).

  E.g., memory or CPU.
- **Non-preemptable**: cannot be taken away from its current owner without causing the computation to fail.

  E.g., printer or floppy disk.

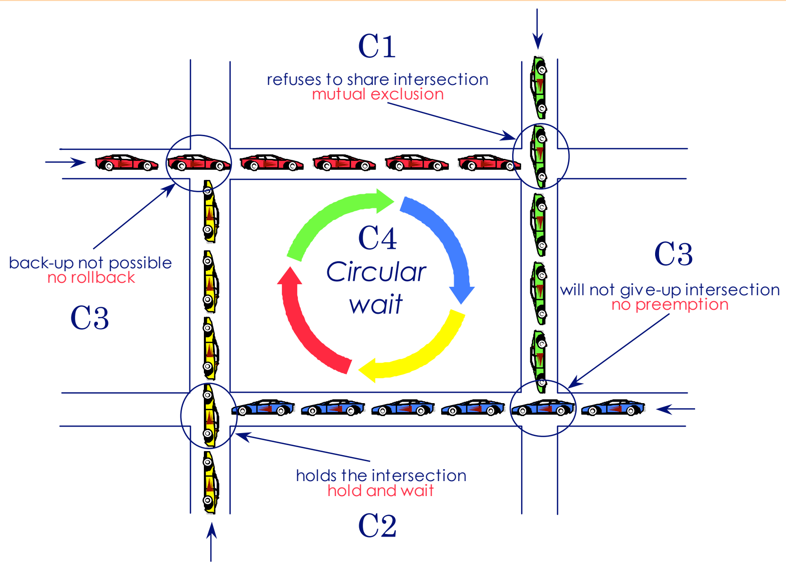
Deadlocks occur when sharing *reusable* and *non-preemptable* resources.

Conditions for deadlock

Four conditions that must hold for a deadlock to be possible:

1. **Mutual exclusion**: processes require exclusive control of its resources (not sharing).
2. **Hold and wait**: process may wait for a resource while holding others.
3. **No preemption**: process will not give up a resource until it is finished with it. Also, **processes are irreversible**: unable to reset to an earlier state where resources not held.
4. **Circular wait**: each process in the chain holds a resource requested by another

An example



Discussion

If any one of the necessary conditions is prevented a deadlock need not occur. For example:

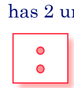
- Systems with only simultaneously shared resources cannot deadlock.
  - Negates *mutual exclusion.*
- Systems that abort processes which request a resource that is in use.
  - Negates *hold and wait.*
- Preemptions may be possible if a process does not use its resources until it has acquired all it needs.
  - Negates *no preemption.*
- Transaction processing systems provide checkpoints so that processes may back out of a transaction.
  - Negates *irreversible process.*
- Systems that prevent, detect, or avoid cycles.
  - Negates *circular wait.* Often, the preferred solution.

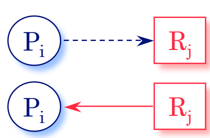
Resource allocation graphs

Set of Processes $P = \{P_1, P_2, \ldots, P_n\}$

Set of Resources $R = \{R_1, R_2, \ldots, R_m\}$ R_j has 2 units

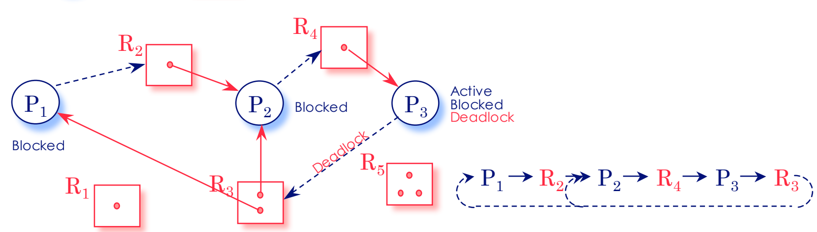
Some resources come in multiple units.



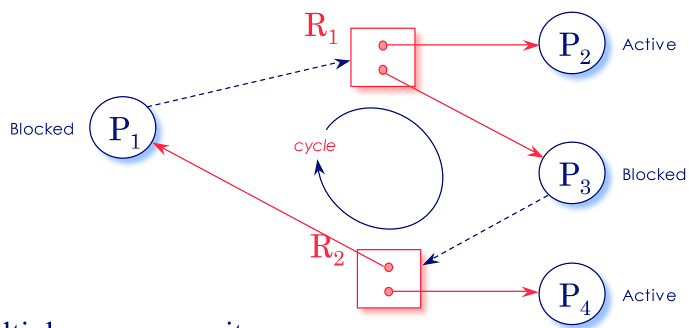


Process P_i waits for (has requested) R_j

Resource R_j has been allocated to P_i



Cycle is necessary, but …



Multiple resource unit case:

*No Deadlock—**yet**!*

Because, either P_2 or P_4 could relinquish a resource allowing P_1 or P_3 (which are currently blocked) to continue. P_2 is still executing, even if P_4 requests R_1.

… a knot is required

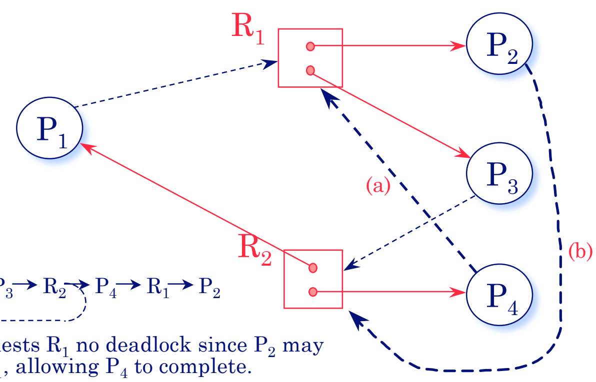
Cycle is a *necessary condition* for a deadlock. But when dealing with multiple unit resources—*not sufficient.*

A *knot* must exist—a cycle with no non-cycle outgoing path from any involved node.

At the moment assume that:

- a process *halts* as soon as it waits for one resource, and
- processes can wait for only *one* resource at a time.

Further requests



OK $P_1 \rightarrow R_2 \rightarrow P_3 \rightarrow R_2 \rightarrow P_4 \rightarrow R_1 \rightarrow P_2$

(a) If P_4 requests R_1 no deadlock since P_2 may release R_1, allowing P_4 to complete.

Not OK $P_1 \rightarrow R_2 \rightarrow P_3 \rightarrow R_2 \rightarrow P_4 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2$

(b) If P_2 requests R_2: Deadlock—*Cycle*—*Knot.*
No active processes to release resources.

Strategies for deadlocks

In general, four strategies are used for dealing with deadlocks:

- **Ignore**: stick your head in the sand and pretend there is no problem at all.
- **Prevent**: design a system in such a way that the possibility of deadlock is excluded *a priori* (e.g., compile-time/statically, by design)
- **Avoid**: make a decision dynamically checking whether the request will, if granted, potentially lead to a deadlock or not (e.g., run-time/dynamically, before it happens)
- **Detect**: let the deadlock occur and detect when it happens, and take some action to recover after the fact (e.g., run-time/dynamically, after it happens)

Ostrich algorithm!

Different people react to this strategy in different ways:

- **Mathematicians**: find deadlock totally unacceptable, and say that it must be prevented at all costs.
- **Engineers**: ask how serious it is, and do not want to pay a penalty in performance and convenience.

The UNIX approach is just to ignore the problem on the assumption that most users would prefer an occasional deadlock, to a rule restricting user access to only one resource at a time.

The problem is that the prevention price is high, mostly in terms of putting inconvenient restrictions on processes.

Deadlock prevention

The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded *a priori*. Methods for preventing deadlock are of two classes:

- *indirect methods* prevent the occurrence of one of the necessary conditions listed earlier.
- *direct methods* prevent the occurrence of a circular wait condition.

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes.

More on deadlock prevention

- Mutual exclusion
  - In general, this condition cannot be disallowed.
- Hold-and-wait
  - The hold and-wait condition can be prevented by requiring that a process request all its required resources at one time, and blocking the process until all requests can be granted simultaneously.
- No preemption
  - One solution is that if a process holding certain resources is denied a further request, that process must release its unused resources and request them again, together with the additional resource.
- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types (e.g. Directed Acyclic Graph). If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.

Deadlock avoidance

Deadlock avoidance, allows the necessary conditions but makes judicious choices to ensure that a deadlock-free system remains free from deadlock. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future requests for process resources.

Ways to avoid deadlock by careful resource allocation:

- Resource trajectories.
- Safe/unsafe states.
- Dijkstra's Banker's algorithm.

Banker's algorithm—definitions

Assume N Processes $\{P_i\}$
M Resources $\{R_j\}$

Availability vector **Avail**$_j$, units of each resource (initialized to maximum, changes dynamically).

Let [**Max**$_{ij}$] be an N x M matrix.

Max$_{ij}$= **L** means Process P_i will request at most **L** units of R_j.

[**Hold**$_{ij}$] Units of R_j currently held by P_i

[**Need**$_{ij}$] Remaining need by P_i for units of R_j

Need$_{ij}$ **= Max**$_{ij}$ **- Hold**$_{ij}$, for all i & j

Banker's Algorithm—*resource request*

At any instance, P_i posts its request for resources in vector **REQ**$_j$ (i.e., no hold-and-wait)

Step 1: verify that a process matches its needs.
if **REQ**$_j$ **>** **Need**$_i$ ***abort***—*error, impossible*

Step 2: check if the requested amount is available.
if **REQ**$_j$ **>** **Avail**$_j$ ***goto*** ***Step 1***—*P_i must wait*

Step 3: provisional allocation (i.e., guess and check)
Avail$_j$ **=** **Avail**$_j$ **-** **REQ**$_j$
Hold$_{ij}$ **=** **Hold**$_{ij}$ **+** **REQ**$_j$
Need$_{ij}$ **=** **Need**$_{ij}$ **-** **REQ**$_j$
if **isSafe()** ***then*** grant resources—*system is safe*
else cancel allocation; ***goto*** ***Step 1***—*P_i must wait*

Banker's Algorithm—*isSafe*

Find out whether the system is in a safe state.
`Work` and **`Finish`** are two temporary vectors.

`Step 1:` initialize.
$\mathbf{Work_j = Avail_j}$ for all j; $\mathbf{Finish_i = false}$ for all i.

`Step 2:` find a process P_i such that
$\mathbf{Finish_i = false}$ and $\mathbf{Need_{ij} \leq Work_j}$, for all j
if no such process, ***`goto Step 4`***.

`Step 3:` $\mathbf{Work_j = Work_j + Hold_{ij}}$ (i.e., pretend it finishes and frees up the resources)
$\mathbf{Finish_i = true}$
`goto Step 2`.

`Step 4:` ***`if`*** $\mathbf{Finish_i = true}$ for all i
`then return` **`true`**—*yes, the system is safe*
`else return` **`false`**—*no, the system is NOT safe*

Banker's algorithm—*what is safe?*

Safe with respect to some resource allocation.

- very safe
  $\mathbf{NEED_i \leq AVAIL}$ for all Processes P_i.
  Processes can run to completion in any order.
- safe (but take care)
  $\mathbf{NEED_i > AVAIL}$ for some P_i
  $\mathbf{NEED_i \leq AVAIL}$ for at least one P_i such that
  There is at least one correct order in which the processes may complete their use of resources.
- unsafe (deadlock inevitable)
  $\mathbf{NEED_i > AVAIL}$ for some P_i
  $\mathbf{NEED_i \leq AVAIL}$ for at least one P_i
  But some processes cannot complete successfully.
- deadlock
  $\mathbf{NEED_i > AVAIL}$ for all P_i
  Processes are already blocked or will become so as they request a resource.

Example—safe allocation

	Max	Hold	Need	Finish	Avail	Work
P_1	5	~~2~~ 3	~~3~~ 2	F	2	~~2~~ 1
P_2	4	1	3	F		
P_3	2	1	1	F		

For simplicity, assume that all the resources are identical.
Assume P_1 acquires one unit. *Very safe?* **No!** **`Need`$_2$ `> 2`**
Safe? Let us see with the safe/unsafe algorithm...

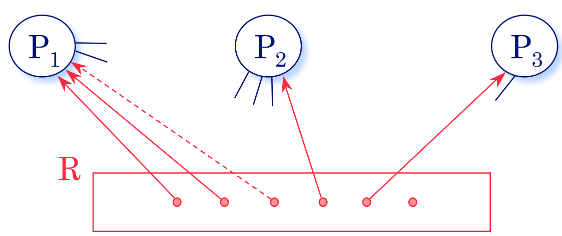
`i = 1`; does P_1 agree with **`Step 2`**? No.
`i = 2`; does P_2 agree with **`Step 2`**? No.
`i = 3`; does P_3 agree with **`Step 2`**? Yes. **`Work = Work+Hold`$_3$`; Finish`$_3$ `= T`**
`i = 1`; does P_1 agree with **`Step 2`**? Yes. **`Work = Work+Hold`$_1$`; Finish`$_1$ `= T`**
`i = 2`; does P_2 agree with **`Step 2`**? Yes. **`Work = Work+Hold`$_2$`; Finish`$_2$ `= T`**

No more (unfinished) P_i, therefore safe.

Example—safe allocation

	Max	Hold	Need	Finish	Avail	Work
P_1	5	~~2~~ 3	~~3~~ 2	F	2	~~2~~ 1
P_2	4	1	3	F		
P_3	2	1	1	F		

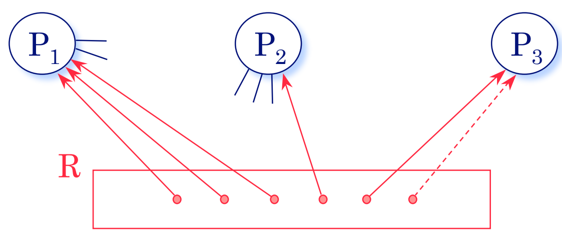
Assume P_1 acquires one unit.



Example—safe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	3	2	F	1	~~1~~ 0
P_2	4	1	3	F		
P_3	2	~~1~~ 2	~~1~~ 0	F		

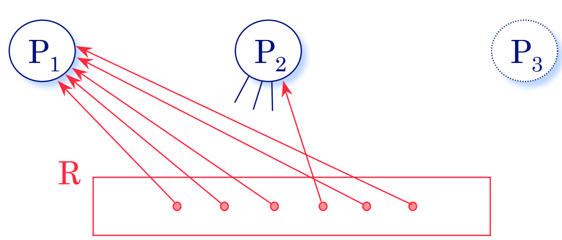
P_3 can acquire the last unit and finish. Then, P_3 frees up resources.



Example—safe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	5	0	F	2	~~2~~ 0
P_2	4	1	3	F		
P_3	2	0	0	T		

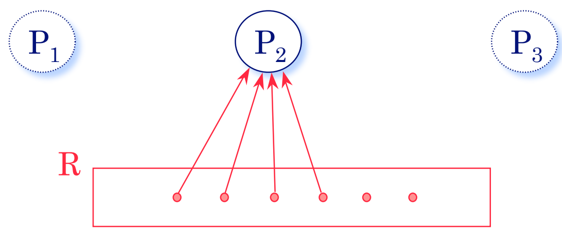
Then, P_1 can acquire two more units and finish.



Example—safe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	0	0	T	5	~~5~~ 2
P_2	4	~~1~~ 4	~~3~~ 0	F		
P_3	2	0	0	T		

Finally, P_2 can acquire three more units and finish.



Example—unsafe allocation

NOTE: New numbers here!

	Max	Hold	Need	Finish	Avail	Work
P_1	5	2	3	F	2	~~2~~ 1
P_2	5	~~1~~ 2	~~4~~ 3	F		
P_3	2	1	1	F		

Assume P_2 acquires one unit.
As before, P_3 can finish and release its resources.

BUT…

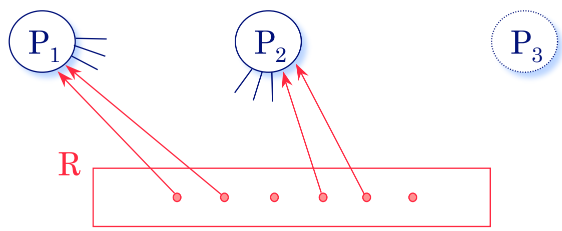
`i = 1`; does P_1 agree with **`Step 2`**? No.
`i = 2`; does P_2 agree with **`Step 2`**? No.
`i = 3`; does P_3 agree with **`Step 2`**? Yes. **`Work = Work+Hold`**$_2$**`;`** **`Finish`**$_2$ **`= T`**

Any more unfinished P_i? Yes.
P_1 and P_2 cannot finish. Therefore unsafe.

Example—unsafe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	2	3	F	2	2
P_2	5	2	3	F		
P_3	2	0	0	T		

NOW...



Deadlock detection

This technique does not attempt to prevent deadlocks; instead, it lets them occur. The system detects when this happens, and then takes some action to recover after the fact (i.e., is reactive). With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition.

A check for deadlock can be made as frequently as resource request, or less frequently, depending on how likely it is for a deadlock to occur. Checking at each resource request has two advantages: It leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system. On the other hand, such frequent checks consume considerable processor time.

Recovering from deadlocks

Once the deadlock algorithm has successfully detected a deadlock, some strategy is needed for recovery. There are various ways:

- Recovery through *Preemption*
  In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another.
- Recovery through *Rollback*
  If it is known that deadlocks are likely, one can arrange to have processes *checkpointed* periodically. For example, can undo transactions, thus free locks on database records. This often requires extra software functionality.
- Recovery through *Termination*
  The most trivial way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle.
  Warning! *Irrecoverable losses or erroneous results may occur, even if this is the least advanced process.*

Summary of strategies

Principle	*Resource Allocation Strategy*	*Different Schemes*	*Major Advantages*	*Major Disadvantages*
DETECTION	• Very liberal; grant resources as requested.	• Invoke periodically to test for deadlock.	• Never delays process initiation. • Facilitates on-line handling.	• Inherent preemption losses.
PREVENTION	• Conservative; under-commits resources.	• Requesting all resources at once.	• Works well for processes with single burst of activity. • No preemption is needed.	• Inefficient. • Delays process initiation.
		• Preemption	• Convenient when applied to resources whose state can be saved and restored easily.	• Preempts more often then necessary. • Subject to cyclic restart.
		• Resource ordering	• Feasible to enforce via compile-time checks. • Needs no run-time computation.	• Preempts without immediate use. • Disallows incremental resource requests.
AVOIDANCE	• Selects midway between that of detection and prevention.	• Manipulate to find at least one safe path.	• No preemption necessary.	• Future resource requirements must be known. • Processes can be blocked for long periods.

Other issues

Two-phase Locking

Although both avoidance and prevention are not very promising in general, many excellent special-purpose algorithms are known. The best data base algorithm is known as **two-phase locking** (covered in detail in another course).

Non-resource Deadlocks

Deadlocks can also occur in other situations, where no single resource is involved. E.g., two processes exchanging messages, where both are listening and waiting for the other to send a message.

Starvation

A problem closely related to deadlock is **starvation**. In a dynamic system, requests for resources happen all the time. The key is to make a decision about who gets which resources when. This decision sometimes may lead to some processes never receiving service, though they are not deadlocked!