Concurrency: Deadlock and Starvation Problems, more on Mutual Exclusion

Most slides have been adapted from «O*perating Systems: Internals and Design Principles", 7/E* W. Stallings (Chapter 6 + Appendix A).

Sistemi di Calcolo 2

Instructor: Riccardo Lazzeretti

A real-world example

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

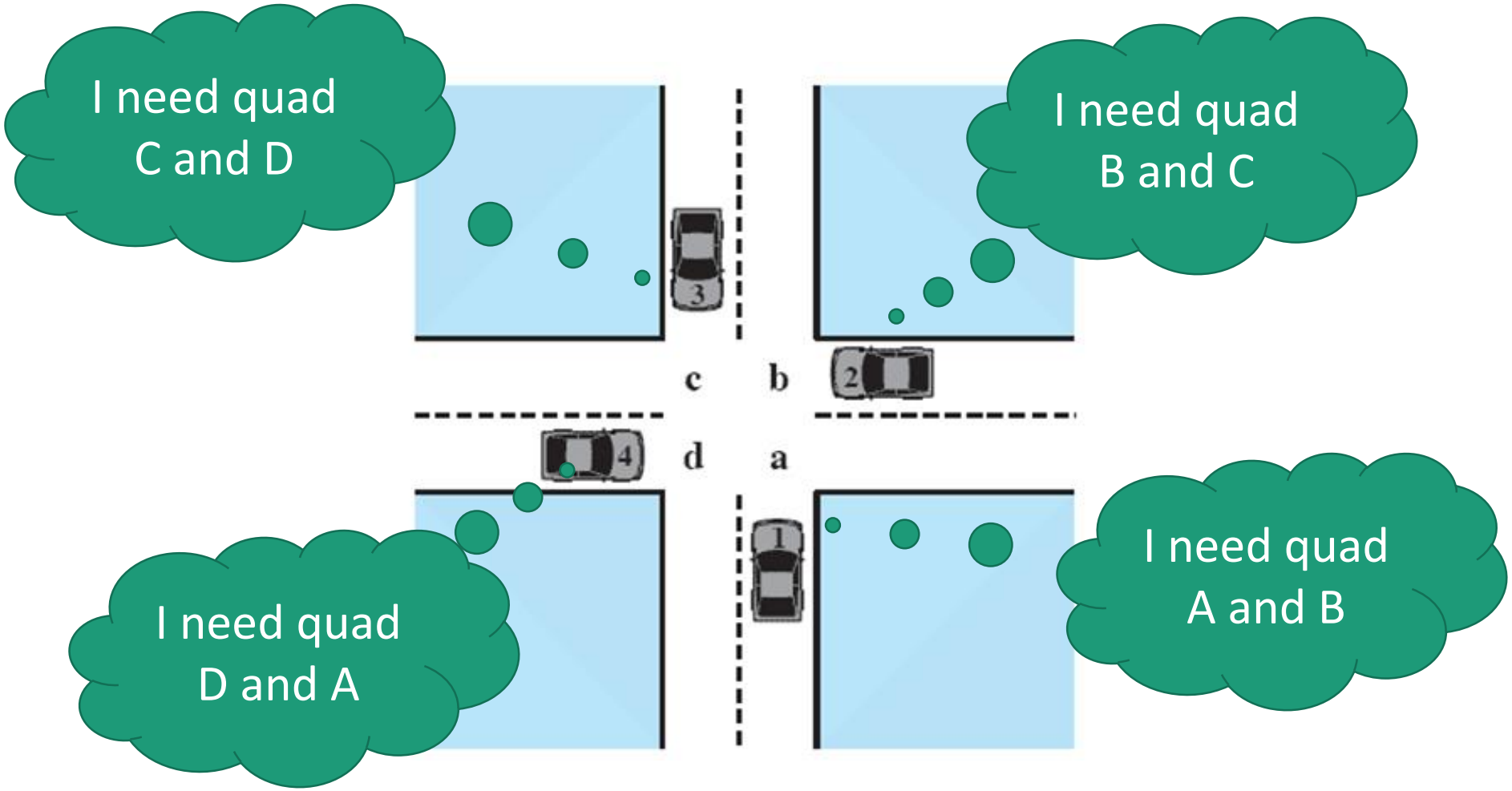
Statute passed by the Kansas State Legislature, early in the 20th century.

—A TREASURY OF RAILROAD FOLKLORE,
B. A. Botkin and Alvin F. Harlow

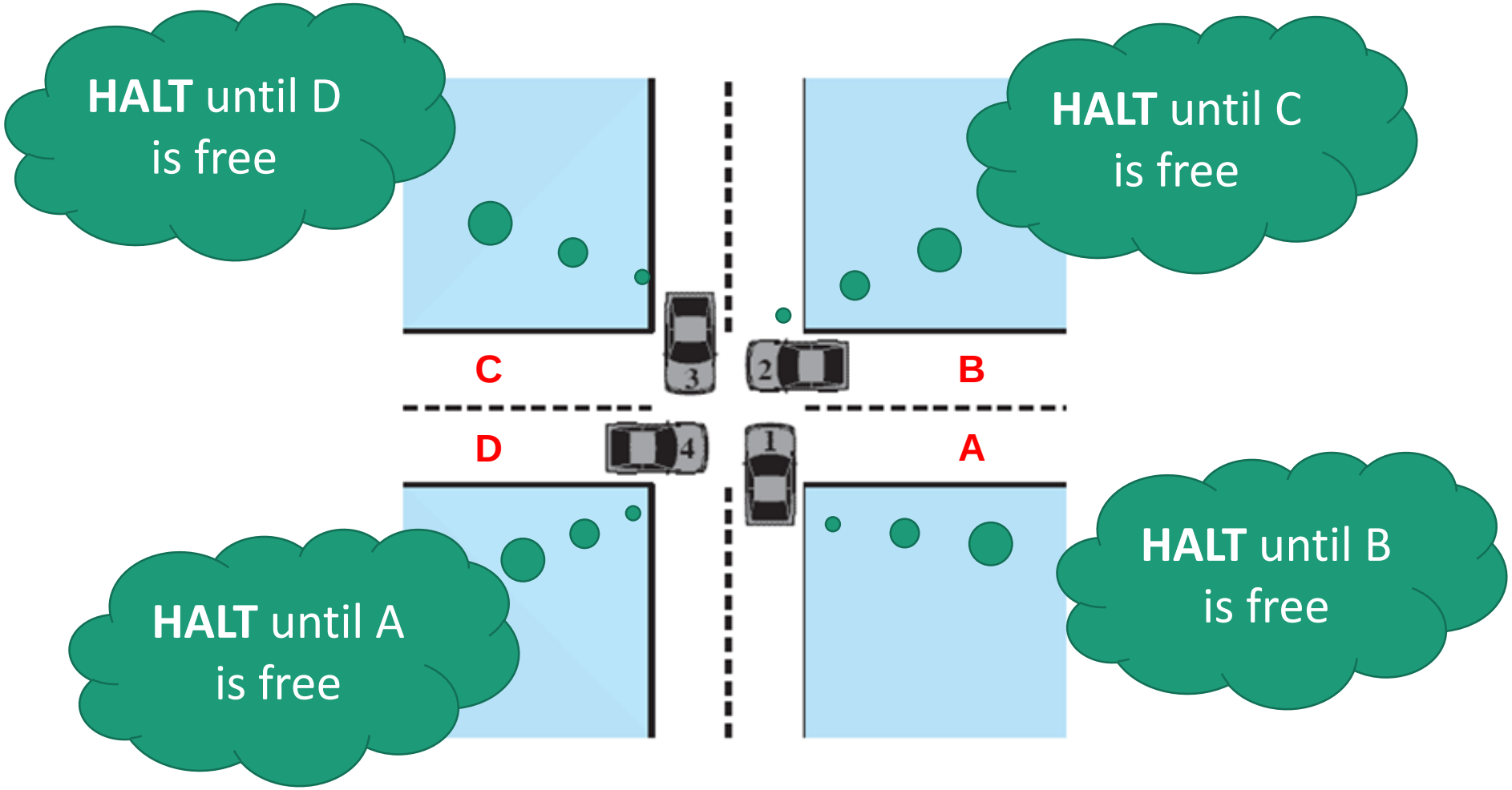
Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution

Potential Deadlock



Actual Deadlock



Resource Categories

Reusable resource

Can be safely used by only one process at a time, and it is not depleted by that use

Examples: processors, I/O channels, main and secondary memory, devices, and structures such as files, databases, and semaphores

Consumable resource

One that can be created (produced) and destroyed (consumed)

Examples: interrupts, signals, messages, and information in I/O buffers

Reusable Resources - Example

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)

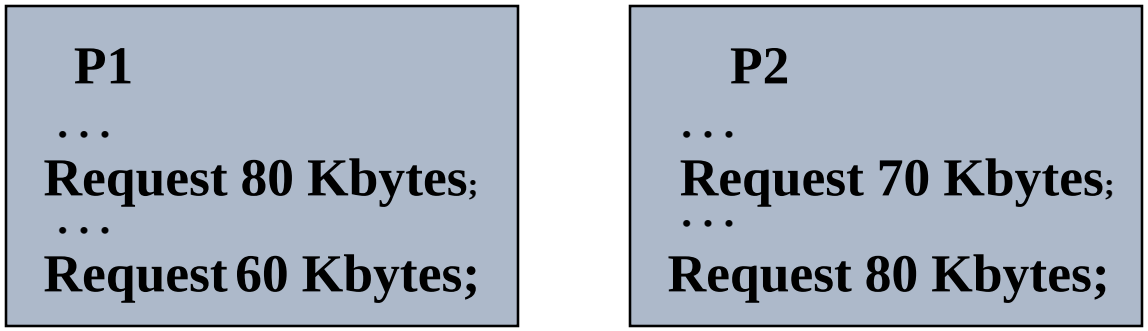
Process Q

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)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:



- Deadlock occurs if both processes progress to their second request

Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then sends a message to the other process:

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

- Deadlock occurs if the Receive is blocking

Conditions for Deadlock

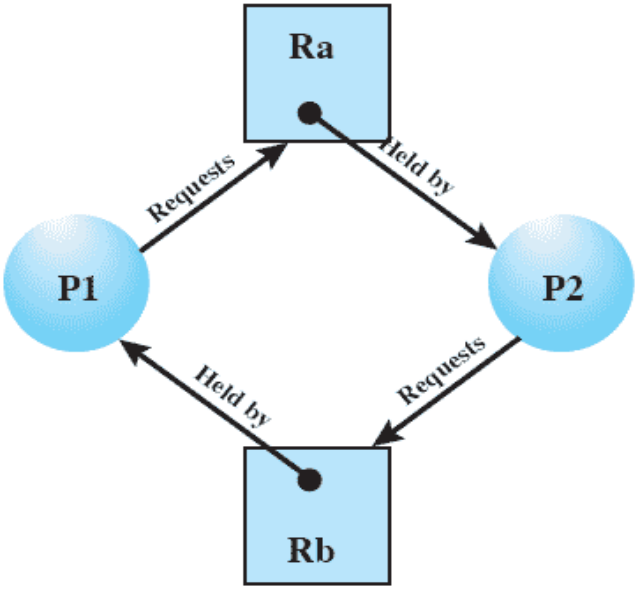
Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

The first three conditions are necessary but not sufficient

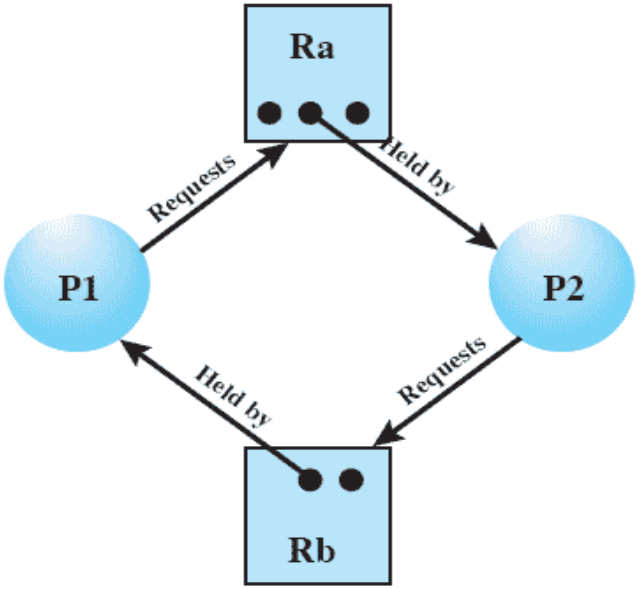
Resource Allocation Graphs

(a) Resouce is requested

(b) Resource is held

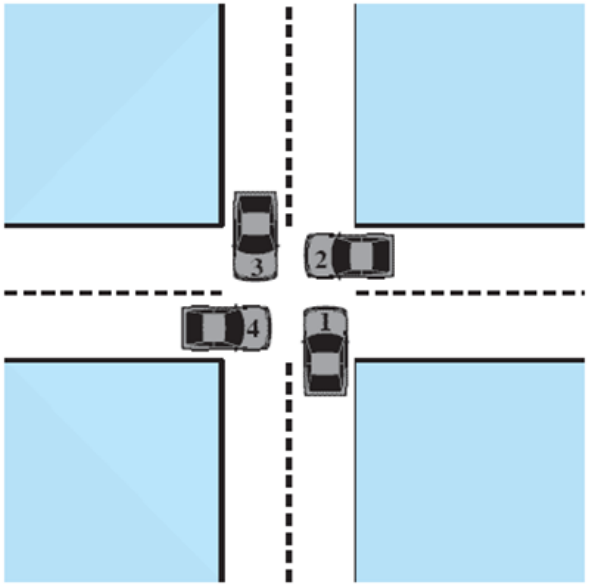


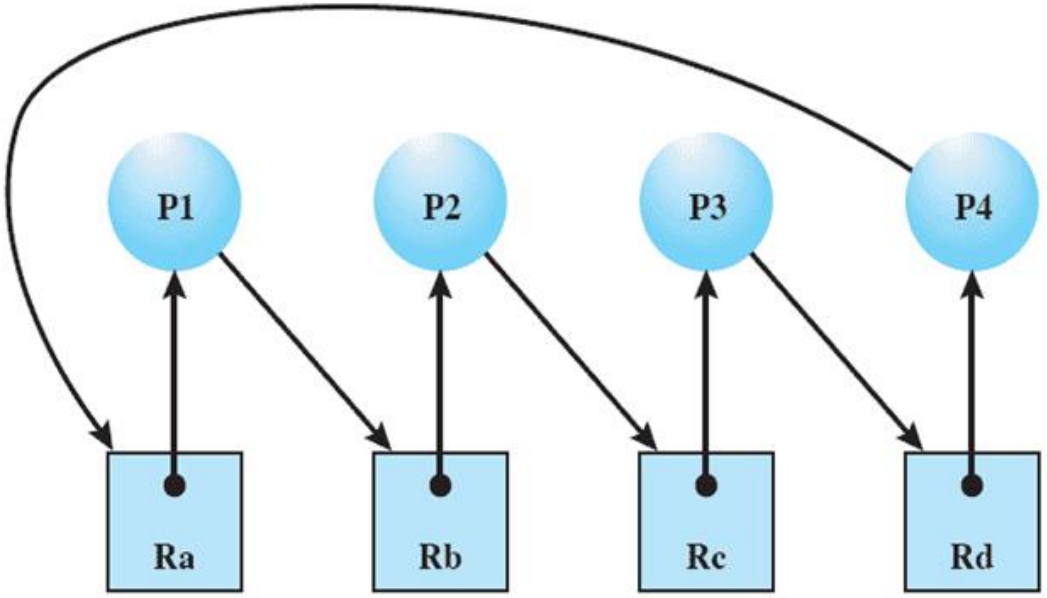
(c) Circular wait



(d) No deadlock

Resource Allocation Graphs





Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

Prevent Deadlock

- adopt a policy that eliminates one of the conditions

Avoid Deadlock

- make the appropriate dynamic choices based on the current state of resource allocation

Detect Deadlock

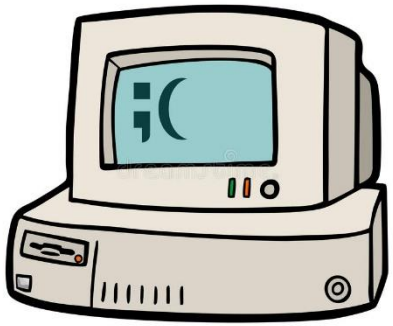
- attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
  - Indirect
    - prevent the occurrence of one of the three necessary conditions
  - Direct
    - prevent the occurrence of a circular wait

Deadlock Condition Prevention

- **Mutual Exclusion**
  - if access to a resource requires mutual exclusion then it must be supported by the OS
  - usually, cannot be disallowed
  - we can relax it under some conditions
    - Ex: access to file must be granted in mutual exclusion
    - Read access can be allowed to multiple processes

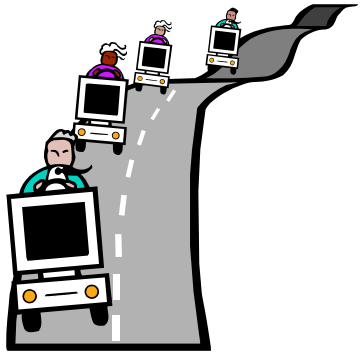


- **Hold and Wait**
  - require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
  - Inefficient:
    1. Process can wait for a long time before all the resources are available
    2. Process can hold resources for a long time, even when not used
  - Process could not know in advance which resources it will need

Deadlock Condition Prevention

- **No Preemption**
  - if a process holding certain resources is denied a further request, that process must release its original resources and request them again; alternatively, OS may preempt the second process and require it to release its resources
  - a process must have higher priority than the other one
  - approach practical only for resources for which the state can be saved and later restored (e.g., CPU)
- **Circular Wait**
  - define a linear ordering of resource types
  - if a resource of type R has been assigned to a process, the latter can only request resources whose types follow R in the order
  - same problems of hold and wait

Deadlock Avoidance



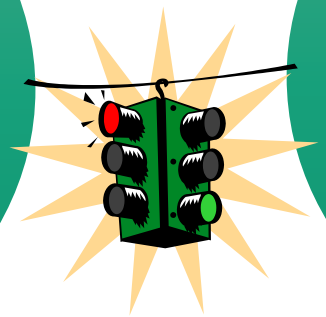
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests
- Allows more concurrency than deadlock prevention

Two Approaches to Deadlock Avoidance

Deadlock Avoidance

Resource Allocation Denial

- do not grant an incremental resource request to a process if this allocation might lead to deadlock



Process Initiation Denial

- do not start a process if its demands might lead to deadlock

Process Initiation Denial

R1	R2	R3
9	3	6

Resource vector **R**

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

Claim matrix **C**

- OS allows the execution of a new process only if the sum of the claim resources of running process + claim resources of new process is lower than the available resources
- If P1 and P2 are running, P3 and P4 cannot be executed

Determination of a Safe State

- A state is safe if a sequence of resource allocation exists such that does not result in deadlock

- State of a system consisting of four processes and three resources
- Allocations have been made to the four processes

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Amount of existing resources

Resources available after allocation

P2 Runs to Completion

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

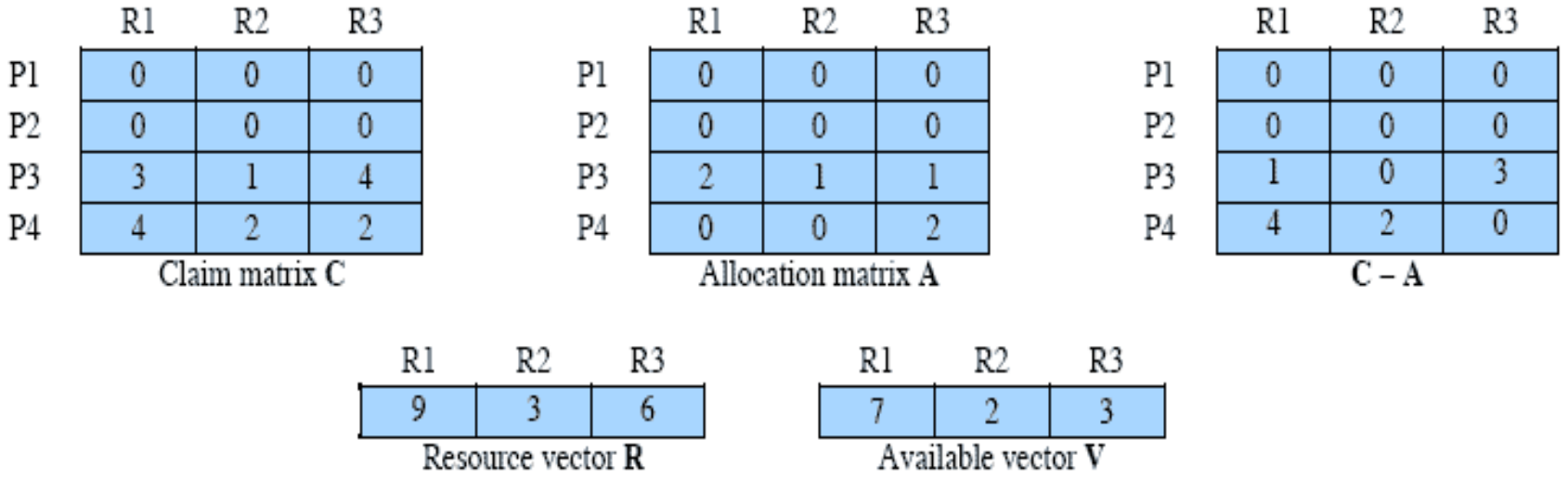
Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) **P2 runs to completion**

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 **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

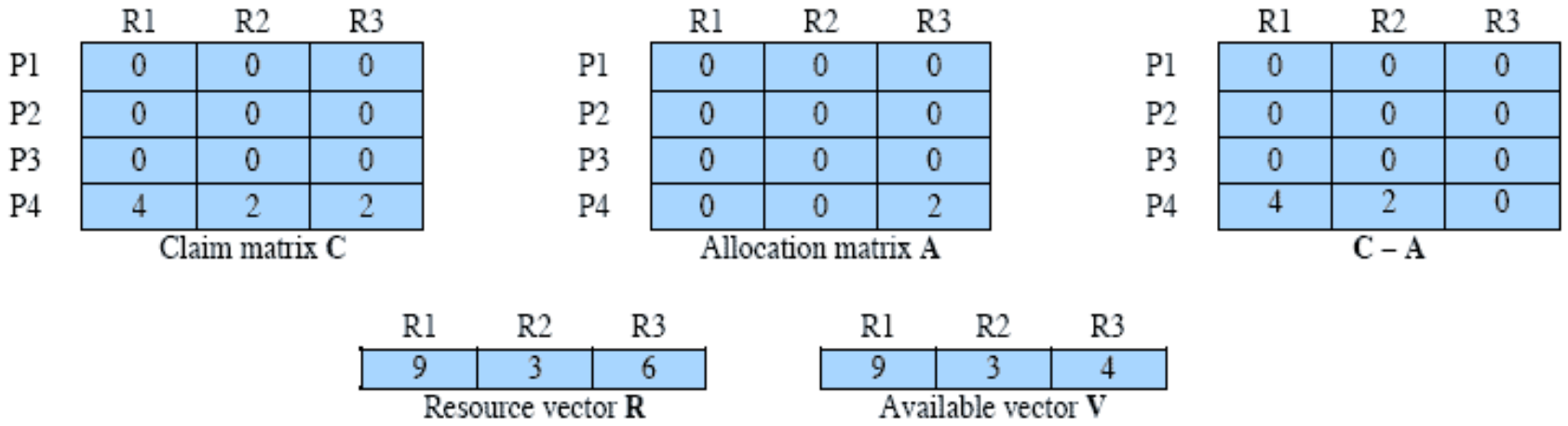
Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

P3 Runs to Completion



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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Thus, the state defined originally is a safe state

Determination of an Unsafe State

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

Deadlock Avoidance Logic

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                                /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Deadlock Avoidance Logic

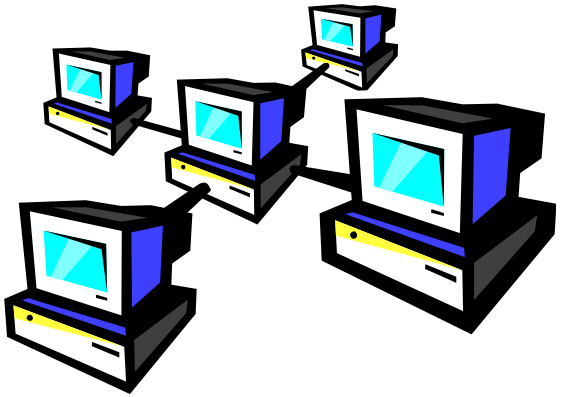
```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Deadlock Avoidance Advantages

- Process initiaton denial is hardly optimal...
- ...while resource allocation denial has a few perks!
  - It is not necessary to preempt and roll back processes, as in deadlock detection *(we will discuss detection soon)*
  - It is less restrictive than deadlock prevention



Deadlock Avoidance Restrictions

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

Deadlock Strategies

Deadlock prevention strategies are very conservative

- limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- resource requests are granted whenever possible

Deadlock Detection Algorithms

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur.

Detection performed at each resource request:

- Pros: leads to early detection, algorithm is simple
- Cons: frequent checks consume considerable CPU time

A Detection protocol

- We iteratively mark protocols that can be suspended or terminate
- If at the end all the protocols are marked we have no deadlock

	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

Allocation vector

1. P4 is marked because has no resources yet
2. P3 is marked because if we give resources it can terminate and release resources
3. Even if P3 releases resources, neither P1 nor P2 can terminate
   => Deadlock!!!

Deadline Detection - Recovery

Some approaches, in increasing order of sophistication:

- Abort all deadlocked processes (most common)
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists (requires rollback mechanism)

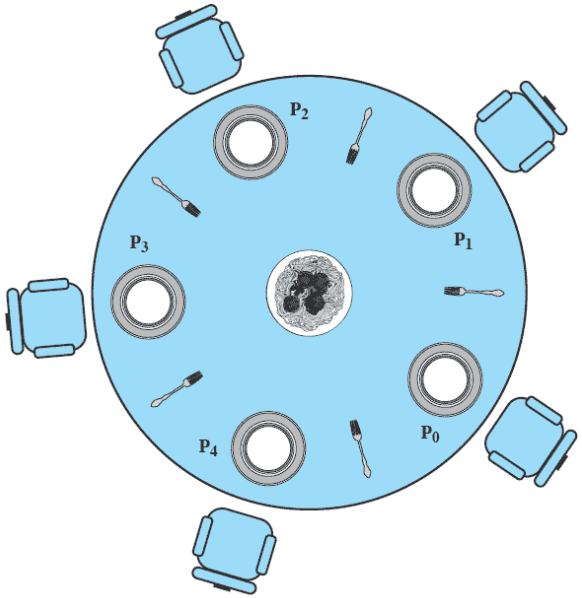
Deadlock Approaches

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Dining Philosophers Problem

No two philosophers can use the same fork at the same time (mutual exclusion)

No philosopher must starve (avoid deadlock & starvation)



Why so relevant?

- illustrates basic problems in deadlock and starvation
- standard test case for evaluating approaches to synchronization

Using N semaphores?

```
semaphore fork [5] = {1};
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
```

Try to pick the i-th fork on the left, then (i+1)-th fork on the right: the code above will easily lead to deadlock

A Correct Solution . . .

```
semaphore fork[5] = {1};
semaphore room = {4};
void philosopher (int i)
{
     while (true) {
          think();
          wait (room);
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal (fork [(i+1) mod 5]);
          signal (fork[i]);
          signal (room);
     }
}
```

Software approaches to mutual exclusion

Mutual exclusion can be implemented at software level too

Scenario: processes communicate via a central memory on a single/multi-processor machine

Assumption: mutual exclusion at the memory access level

- read/write operations for the same location are serialized by some sort of memory arbiter (the order is unknown)
- No support in OS, hardware, or programming language

Dekker algorithm

Dijkstra reported an algorithm for mutual exclusion designed by Dutch mathematical Dekker

2 processes!

We develop it by attempts

First attempt

```
/* global */
int turn = 0;

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;
while (true) {
   /*NCS*/
   while (turn != me)
        /* busy wait */ ;
   /* CS */
   turn = other;
}
```

Second attempt

```
/* global */
boolean flag[2] = {false, false};

// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
   /*NCS*/
   while (flag[other])
        /* busy wait */ ;
   flag[me] = true;
   /* CS */
   flag[me] = false;
}
```

Third attempt

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
   /*NCS*/
   flag[me] = true;
   while (flag[other])
       /* busy wait */ ;
   /* CS */
   flag[me] = false;
}
```

Fourth attempt

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
   /*NCS*/
   flag[me] = true;
   while (flag[other]) {
       flag[me] = false;
       /* delay */
       flag[me] = true;
   }
   /* CS */
   flag[me] = false;
}
```

Correct Solution: Dekker's Algorithm

```
/*global*/ turn = {0,1}
int me = 0, other = 1; // P0 (flip for P1)

while (true) {
    /*NCS*/
    flag[me] = true;
    while (flag[other]) {
        if (turn == other) {
            flag[me] = false;
            while (turn == other) /* busy wait */ ;
            flag[me] = true;
        }
    }
    /* CS */
    turn = other;
    flag[me] = false;
}
```

Dekker's Algorithm, informally

```
flag[me] = true;              I want to enter
while (flag[other]) {         If you want to enter
  if (turn==other) {          and if it's your turn
    flag[me] = false;         I don't want any more
    while (turn==other);      If it's your turn I'll wait
    flag[me] = true;          I want to enter
  }
}
/* CS */
turn = other;                 You can enter next
flag[me] = false;             I don't want any more
```

Credits: https://cs.stackexchange.com/q/12621

Dijkstra (1965)

Dijkstra generalized Dekker's algorithm to deal with N entities

He also provided definitions for mutual exclusion (ME), no deadlock (ND), and no starvation (NS) properties

=> notice that NS implies ND

```
/* global storage */
boolean interested[N] = {false, ..., false}
boolean passed[N]     = {false, ..., false}
int k = <any>         // k ∈ {0, 1, ..., N-1}

/* local info */
int i = <entity ID>   // i ∈ {0, 1, ..., N-1}
```

Dijkstra's Algorithm

```
while (true) {
   /*NCS*/
1.  interested[i] = true
2.  while (k != i) {
3.    passed[i] = false
4.    if (!interested[k]) then k = i
    }
5.  passed[i] = true
6.  for j in 1 ... N except i do
7.    if (passed[j]) then goto 2
8.  <critical section>
9. passed[i] = false; interested[i] = false
}
```

A closer look

Trying protocol

```
1.  interested[i] = true
2.  while (k != i) {
3.    passed[i] = false
4.    if (!interested[k]) then k = i
    }
5.  passed[i] = true
6.  for j in 1 ... N except i do
7.    if (passed[j]) then goto 2
```

1: process i shows interest in entering CS

2-4: k “chooses” among processes that showed interest

5: phase one passed by process i

6-7: restart if more than one process passed phase one

Example (N ≥ 4, initially k = 4)

P1	P2	P3
`intr[1] = true`		
`while (k!=1)`		
`pass[1] = false`		
	`intr[2] = true`	
	`while (k!=2)`	
	`pass[2] = false`	
		`intr[3] = true`
		`while (k!=3)`
		`pass[3] = false`
`if (!intr[4])`		
	`if (!intr[4])`	
		`if (!intr[4])`
`k = 1`		
	`k = 2`	
		`k = 3`

Entering the Critical Section

Processes P1, P2, and P3 are now ready to execute lines 5-7

- only one of them will be able to access CS
- **k=3** does not imply that P3 goes first: k is used only to solve conflicts, thus the scheduler may also let P1 or P2 in first!
- a conflict arises when two processes set pass[i]=true before the other has entered and then left the critical section

P3
`pass[3] = true`
`for j in 1..N except 3 do`
`if pass[j] goto 2` ***(skipped)***
`<critical section>`
`pass[3] = false`
`intr[3] = false`

One possible conflict-free continuation: the scheduler lets P3 run with P1 and P2 still "stopped" at line 5

P3 finished, conflict for P1 & P2?

P1	P2
`while (k!=1)`	
`pass[1] = false`	
	`while (k!=2)`
	`pass[2] = false`
`if (!intr[3])` ***(true: P3 done)***	
	`if (!intr[3])` ***(true: P3 done)***
`then k = 1`	
`while (k!=1)` ***(skipped)***	
`pass[1] = true`	
	`then k = 2`
	`while (k!=2)` ***(skipped)***
`if (pass[2]) goto 2` **(taken!!)**	
	`if (pass[1]) goto 2` **(taken!!)**

When a conflict occurs, the algorithm goes back to line 2 and clears pass[i]... except for the process that set k last! (P2 here)