Operating Systems 2024/2025

T Class 05 – Deadlocks

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

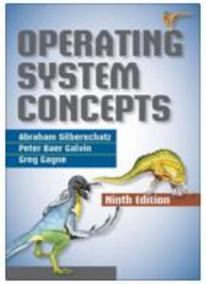
Abbreviation: OS

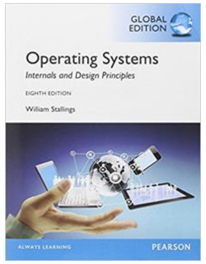
Source: Dictionary.com

Updated on
28 February 2025

Disclaimer

- This slides and notes are based on the companion material [Silberschatz13]. The original material can be found at:
  - http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/
- In some cases, material from [Stallings15] may also be used. The original material can be found at:
  - http://williamstallings.com/OS/OS5e.html
  - http://williamstallings.com/OperatingSystems/
- These slides also use materials from [McHoes2011] “Understanding Operating Systems, 6th Edition, Ann McIver McHoes and Ida M. Flynn”
- The respective copyrights belong to their owners.





Note: Some slides are also based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva (Operating Systems classes of DEI-FCTUC).

Deadlocks

Outlines

- System model
- Characterization of a deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection
- Recovery from a deadlock

System model

- In multiprogramming different processes compete for a finite number of resources
  - (e.g.: CPU, I/O devices, mutexes, semaphores)

- How to use a resource?
  - 1 - **Request** - a process must request a resource before using it
    - If the resource is not available ,the process enters in a waiting state
  - 2 - **Use**
  - 3 - **Release** - resource must be released after being used

- To complete a task, a process may need different resources at the same time

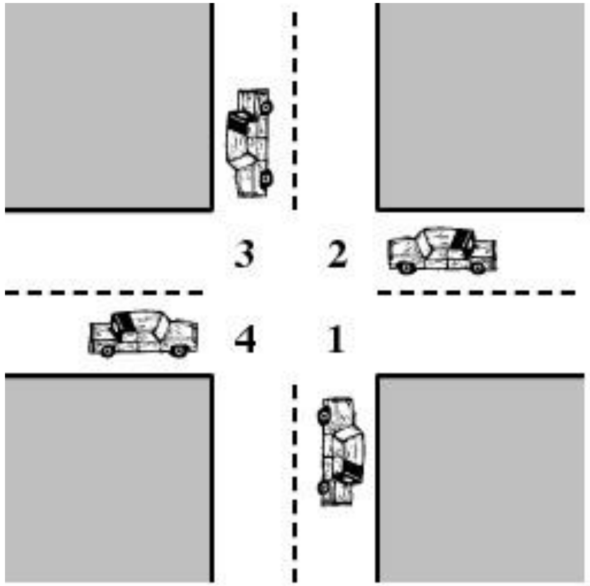
System model

- What is a deadlock?
  - Permanent blocking of a set of processes that either compete for system resources or communicate with each other.

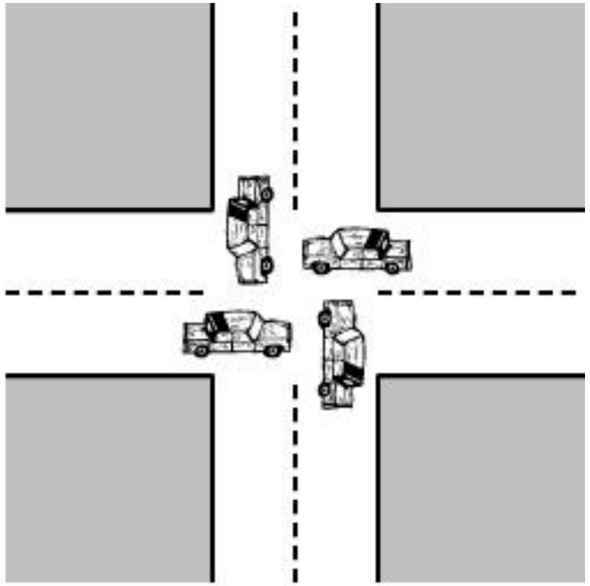
    - No efficient solution.
    - Involves conflicting needs for resources by two or more processes.
    - Typically, OSs do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs.

Deadlock characterization

- Why do deadlocks occur? A traffic example.



(a) Deadlock possible



(b) Deadlock

Deadlock characterization

- Resources and deadlocks
  - 2 resources: A and B

Process P	**Process Q**
Get A	Get B
Get B	Get A
....	....
Release A	Release B
Release B	Release A

```
Execution trace with deadlock
P – Get A
Q – Get B
P – Get B (block)
Q – Get A (block)
<DEADLOCK>
```

```
Execution trace without deadlock
P – Get A
P – Get B
Q – Get B (block)
P - …
P – Release A
P – Release B
Q – Get B (unblock)
Q – Get A
Q - …
Q – Release B
Q – Release A
```

Deadlock characterization

- Operational semantics and deadlocks
  - If `receive()` is blocking…

```
Process P

....
Receive(Q)
...
Send(Q,M1)
```

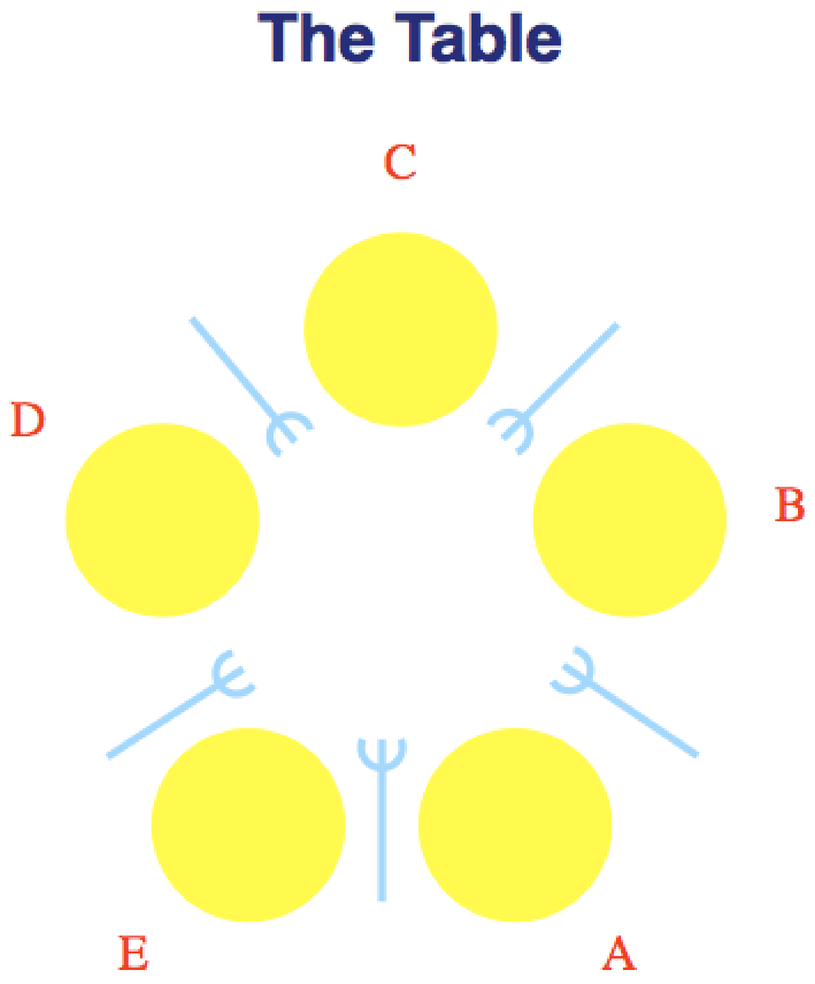
```
Process Q

....
Receive(P)
...
Send(P,M2)
```

```
Execution trace with deadlock
P – Receive(Q) <block>
Q – Receive(P) <block>
<DEADLOCK>
```

Deadlock characterization

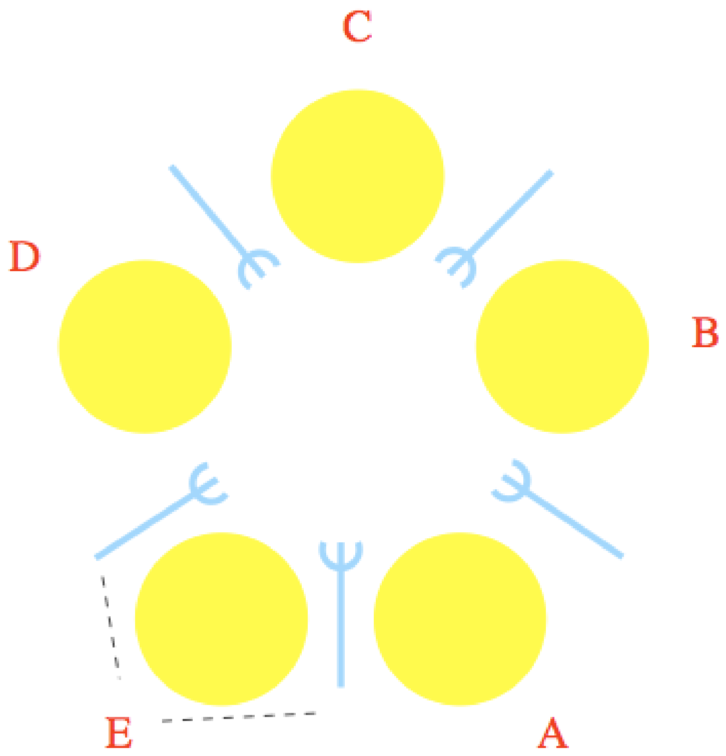
Classical Problem: DINING PHILOSOPHERS

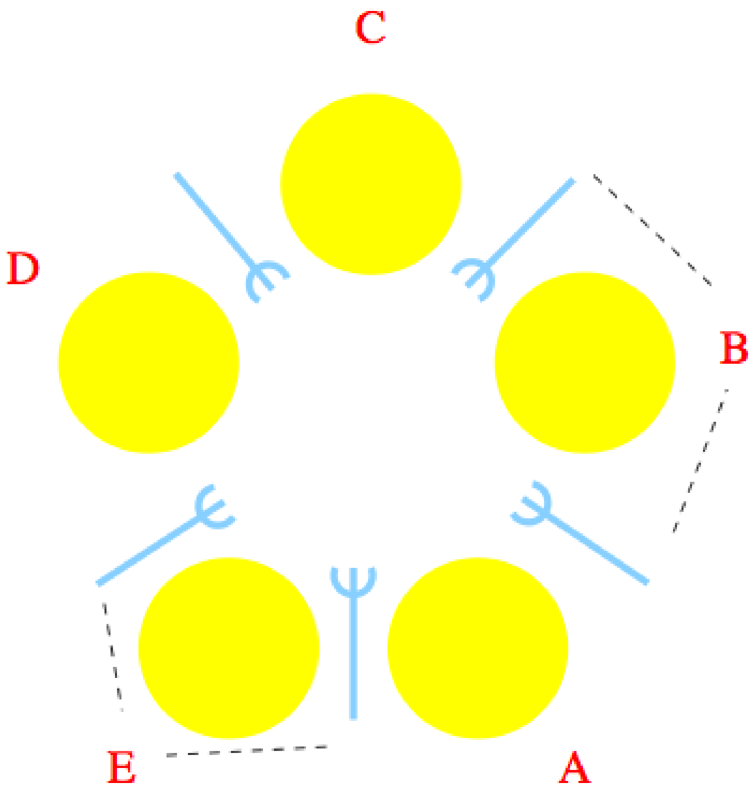
- Five philosophers sitting around a table
- Each has a plate of spaghetti
- There are forks between each pair of plates
- Philosophers need two forks to eat



Deadlock characterization

DINING PHILOSOPHERS (2)





Deadlock characterization

DINING PHILOSOPHERS (3)

- A possible solution… is it good?

Request resources

Use resources

Release resources

```
void philosopher(int i) {
    while(TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

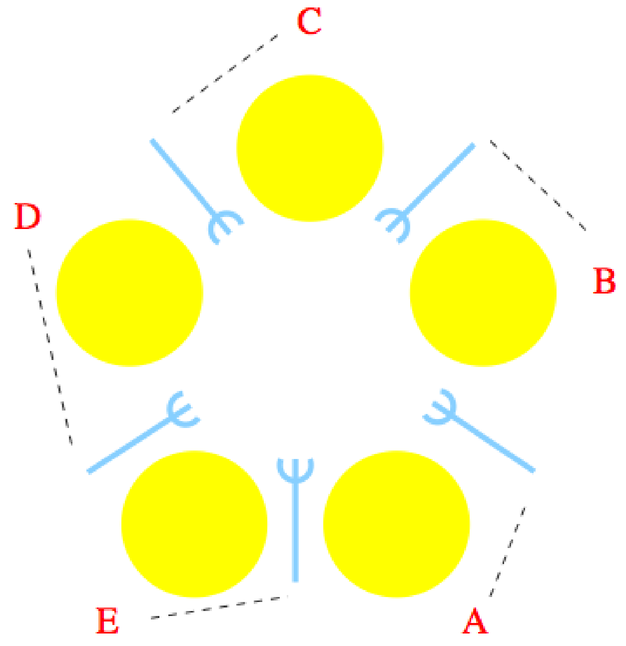
Grab 1 fork

Free 1 fork

Deadlock characterization

DINING PHILOSOPHERS (4)

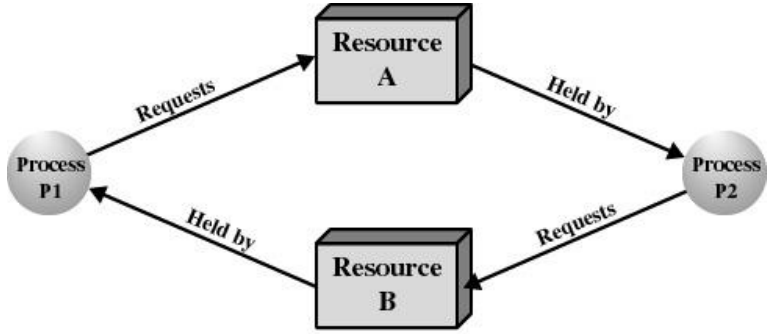
- Everyone grabs the first fork...



- **DEADLOCK...**

Deadlock characterization
Conditions for a Deadlock

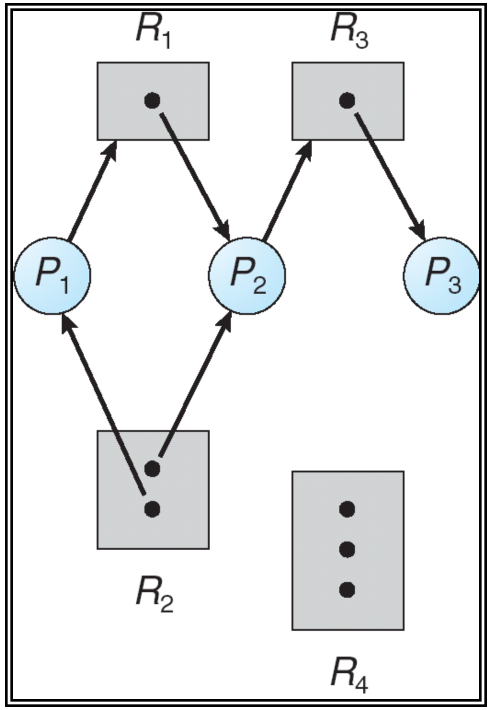
- Deadlock arises if 4 conditions exist at the same time:
  - Mutual exclusion
    - Only one process may use a resource at a time
  - Hold-and-wait
    - A process holds a resource and is waiting for additional ones - a process requests all of its required resources at one time
  - No-preemption
    - Resources cannot be preempted, i.e., can only be released voluntarily.
  - Circular wait
    - A processes is waiting for a resource held by another process that in turn is waiting for a resource held by the first



Deadlock characterization

Resource-Allocation graph

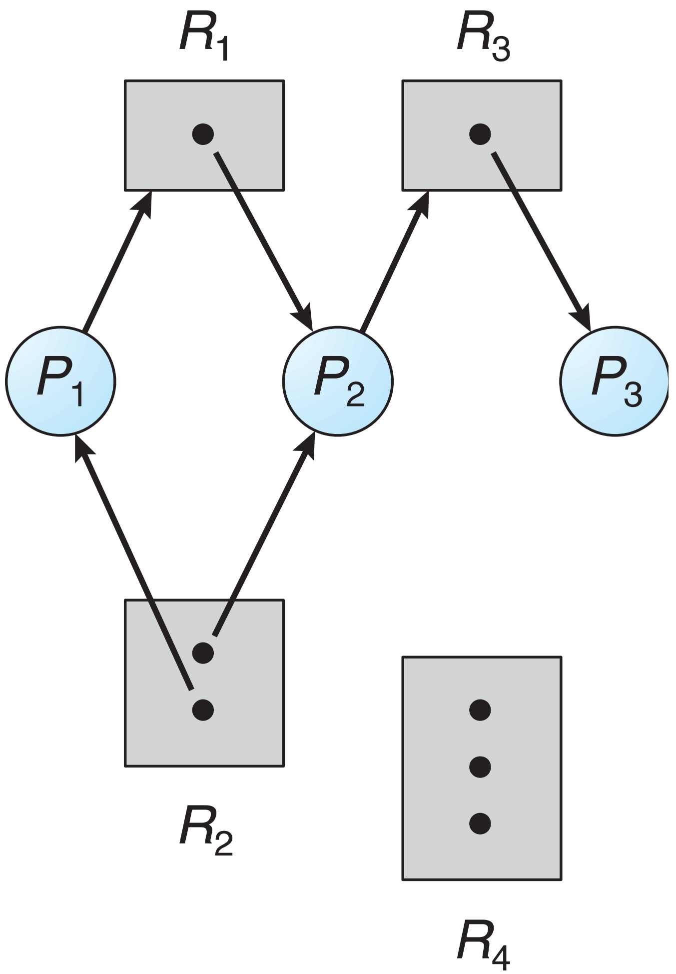
- Direct graph consisting of:
  - Vertices
    - Active processes (circles) - P
    - Resources (rectangles) - R
      - Each dot represents an instance of the resource
  - Directed edges
    - $P_x \rightarrow R_y$: Process P_x requested an instance of resource R_y
    - $R_y \rightarrow P_x$: An instance of R_y has been allocated to P_x
  - It is used to describe the resource-allocation state at a specific time



Deadlock characterization

Resource-Allocation graph (1)

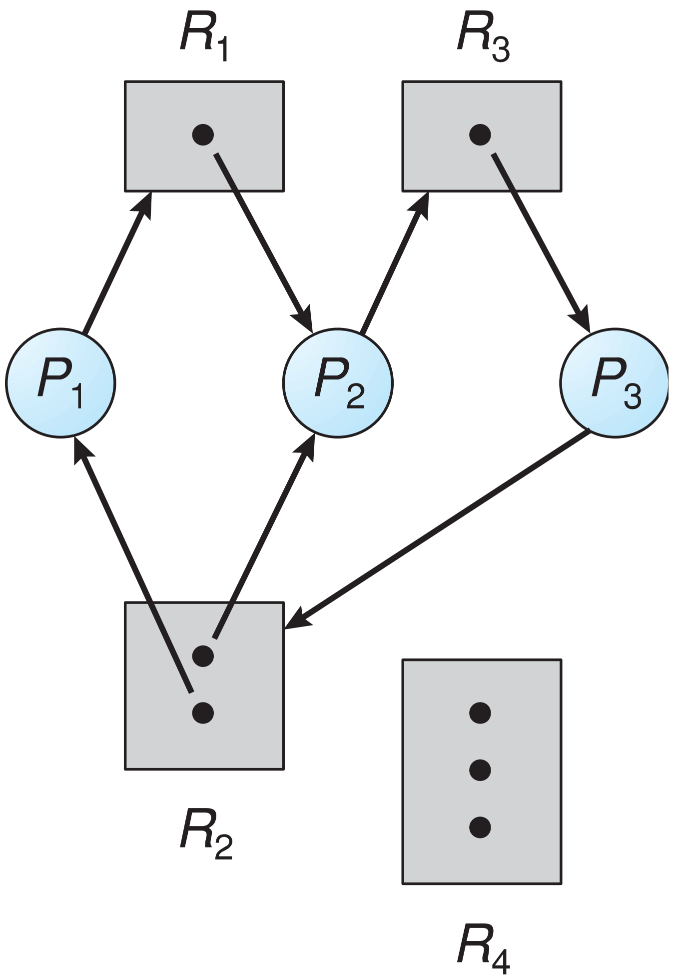
No deadlock



Deadlock characterization

Resource-Allocation graph (2)

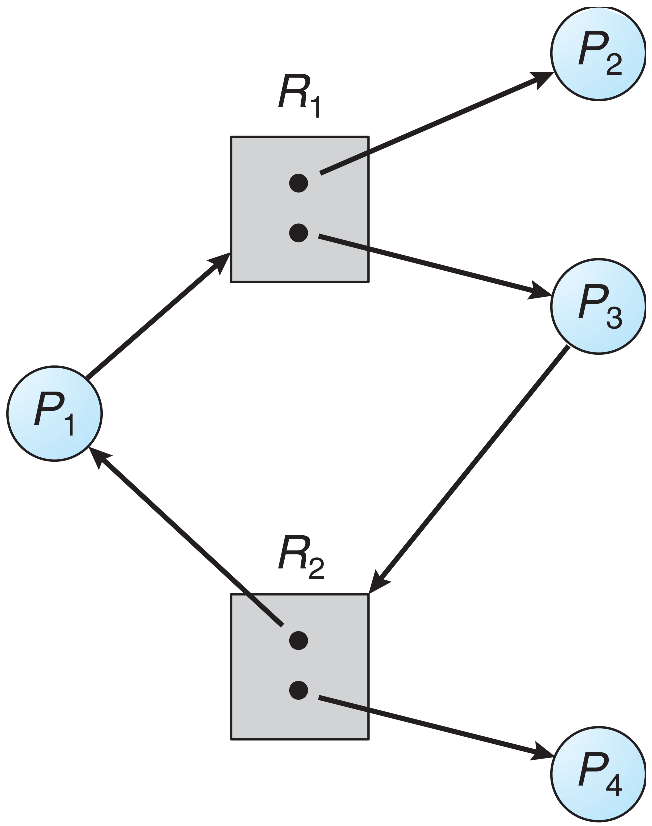
Deadlock



Deadlock characterization

Resource-Allocation graph (3)

- With cycle but no deadlock



Deadlock characterization

Resource-Allocation graph (4)

- Resource-allocation graph analysis
  - If a graph contains no cycles ⇒ no deadlock.
  - If a graph contains a cycle …
    - if only one instance per resource type, then deadlock.
    - if several instances per resource type, possibility of deadlock.

Handling deadlocks

- Options for handling deadlocks in an OS:
  - 1 - **Prevent** deadlocks to ensure that the systems never enters a deadlock state
  - 2 - **Avoid** deadlocks to ensure that the systems never enters a deadlock state
  - 3 - **Allow** a deadlock state, detect it and recover
  - 4 - **Ignore** deadlocks
    - Is up to the programmers to handle deadlocks
    - Used in Linux and Windows

Deadlock prevention

- Prevention is accomplished by ensuring that at least one of the 4 necessary conditions for a deadlock does not happen:
  - Mutual exclusion, hold-and-wait, no-preemption, circular wait
- Possible problems
  - Low device utilization
  - Reduced system throughput

Deadlock prevention (2)

- Prevent Mutual exclusion
  - Have sharable resources (e.g. read-only files)
    - However, not all resources can be sharable!

- Prevent Hold and Wait
  - Guarantee that all the resources are requested and allocated before beginning execution
    - If all resources cannot be guaranteed release the ones allocated
  - Or only request a resource after releasing the ones that are being hold
  - Both options lead to low usage of resources and may lead to starvation of a process that uses different resources

Deadlock prevention (3)

- Prevent No-preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources.
  - If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.

- Prevent Circular Wait
  - Prevented by defining a linear ordering of resource types

Deadlock prevention

Prevent a circular wait - Example

- Define a linear ordering of resources:

  R1, R2, R3,...., Rn

- If (i < j):

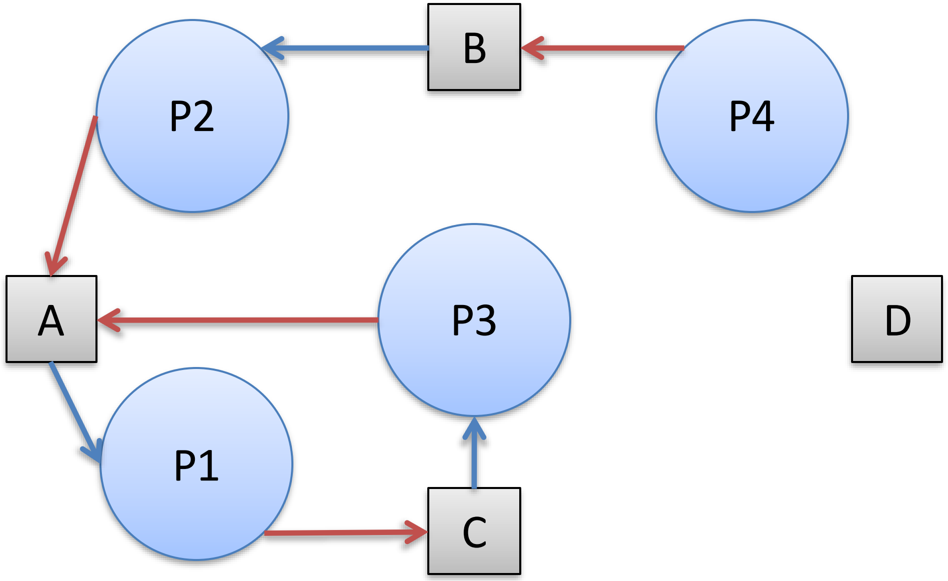
Acquire (Ri) Acquire (Rj)	**OK**
Acquire (Rj) Acquire (Ri)	**NOT OK**

Deadlock prevention

Prevent a circular wait - Example

- Without linear ordering of resources

Processo P1:	**Processo P2:**	**Processo P3:**	**Processo P4:**
Sem_wait(A);	Sem_wait(B);	Sem_wait(C);	Sem_wait(B);
Sem_wait(C);	Sem_wait(A);	Sem_wait(A);	Sem_wait(D);
....	....	....	....
Sem_signal(C);	Sem_signal(A);	Sem_signal(A);	Sem_signal(D);
Sem_signal(A);	Sem_signal(B);	Sem_signal(C);	Sem_signal(B);



Supposing that P1, P2, P3 and P4 execute in sequence and that each executes one instruction...

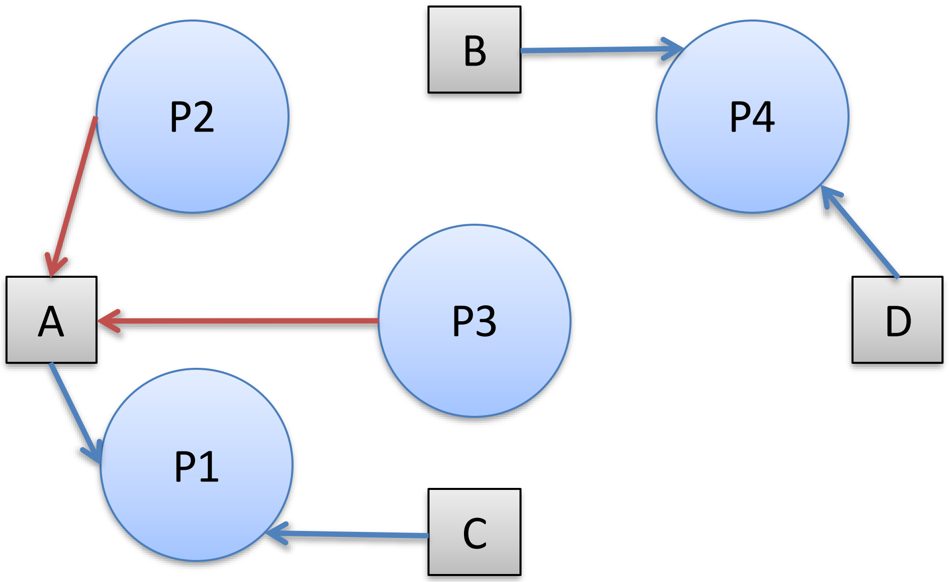
DEADLOCK

Deadlock prevention

Prevent a circular wait - Example

- With linear ordering of resources

Processo P1:	Processo P2:	Processo P3:	Processo P4:
Sem_wait(A);	Sem_wait(A);	Sem_wait(A);	Sem_wait(B);
Sem_wait(C);	Sem_wait(B);	Sem_wait(C);	Sem_wait(D);
....	....	....	....
Sem_signal(C);	Sem_signal(B);	Sem_signal(C);	Sem_signal(D);
Sem_signal(A);	Sem_signal(A);	Sem_signal(A);	Sem_signal(B);



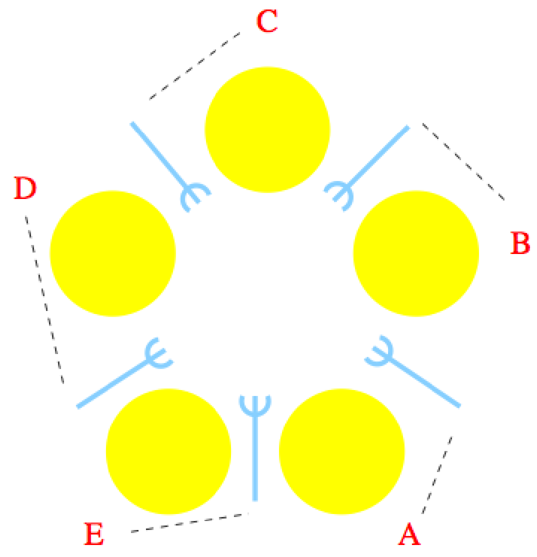
Supposing that P1, P2, P3 and P4 execute in sequence and that each executes one instruction…

After this point P1 and P4 start releasing resources…

NO DEADLOCK

Back to the philosopher's example: can it be prevented?

- Proposal 1:
  - Instead of *take_fork((i+1) % N)*, see if it is available first
  - If unavailable, **sleep** for a while and retry….
  - Still no good….
    - what if everyone tries the left fork, waits, retries simultaneously, etc.
    - Everyone can run, but no one makes progress: ***STARVATION***

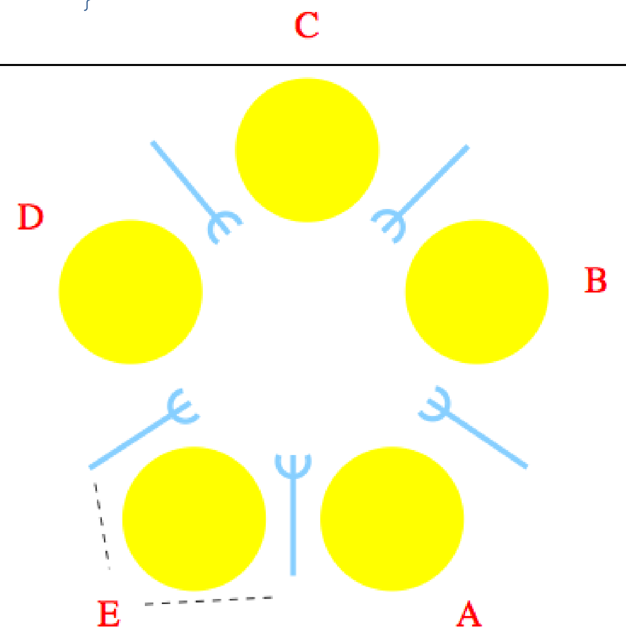


Back to the philosopher's example: can it be prevented?

- Proposal 2:
  - Before taking a fork, a philosopher locks a MUTEX
  - The philosopher can then take two forks, with no interference
  - When done eating, the forks are replaced and MUTEX is released

  - Nope…. only one philosopher can eat at a time…

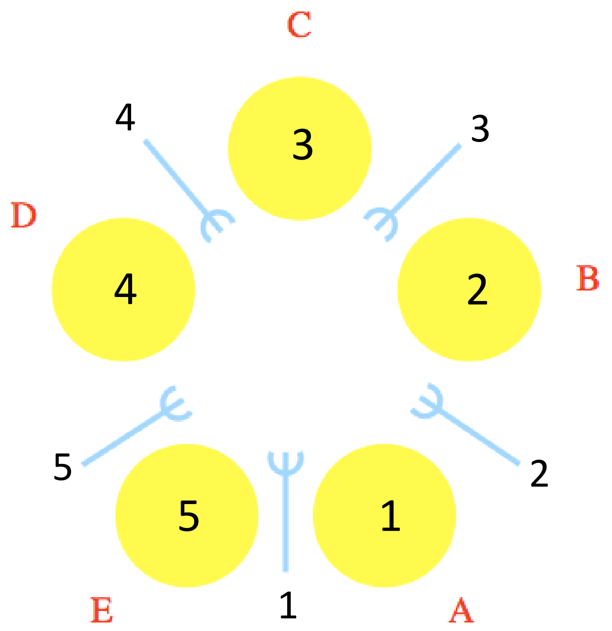
```
void philosopher(int i) {
      while(TRUE) {
            think();
            lock_mutex();
            take_fork(i);
            take_fork((i+1) % N);
            eat();
            put_fork(i);
            put_fork((i+1) % N);
            unlock_mutex();
      }
}
```



Back to the philosopher's example: can it be prevented?

- **Solution:**

```
void philosopher(int i) {
  while(TRUE) {
    think();
    if(id %2 == 0) { // even number: left, right
        take_fork(i);
        take_fork((i+1) % N);
    }
    else{   // odd number: right, left
        take_fork((i+1) % N);
        take_fork(i);
    }
    eat();
    …
```



Forks are the resources! Take them by order!

- Each even philosopher takes first the left fork
- Each odd philosopher takes first the right fork

Deadlock avoidance

- How to avoid the occurrence of deadlocks?
  - Require more information about the resources that are to be requested
  - Approaches
    - Do not start a process if its demands might lead to a deadlock.
    - Do not grant an incremental resource request to a process if this allocation might lead to a deadlock.

Deadlock avoidance

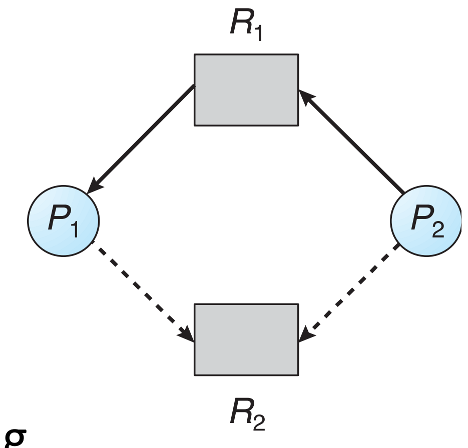
- Deadline avoidance algorithms dynamically examine the resource-allocation state to ensure that a circular-wait condition never happens
  - Is the state a safe state?

Deadlock avoidance
Safe state

- In a system with a a fixed number of processes and resources:
  - The state of the system is the current allocation of resources to process.
  - A safe sequence exists when all processes may access their needed resources without resulting in a deadlock
  - A safe state is when there is at least one safe sequence of resource allocations that does not result in deadlock.
    - All processes can run to completion
    - The system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
  - Unsafe state is a state that is not safe (!).
    - An unsafe state **may** lead to a deadlock (**does not always result in a deadlock!**)

Deadlock avoidance
Resource-allocation-graph algorithm

- **Resource-allocation-graph algorithm**
  - Algorithm used when there is only one instance of each resource type
  - The resource-allocation graph used has an additional edge – **claim edge**
    - A claim edge indicates that a process P_i may request a resource R_j in the future
    - It is depicted by a dashed line



  - All resources must be claimed before the process starts executing
  - When a process P_i requests resource R_j , the **claim edge** $P_i \rightarrow R_j$ is converted to a **request edge**
  - Requests are only granted if converting a **request edge** to an **assignment edge** does not result in a cycle in the graph

Deadlock avoidance
Banker's algorithm

- **Banker's Algorithm**
  - An algorithm that can be used in a banking system to allocate a limited amount of money to customers. The credit is given depending on the risk of the customer, so that the bank may continue to lend.
- Can be used with systems with <u>multiple instances of each resource type</u>
- A strategy of resource allocation denial
- Developed by Dijkstra

Deadlock avoidance
Banker's algorithm

- When a process enters the system, it declares the maximum number of instances of each resource type that it may need
- When resources are requested, the system must determine if their allocation leaves the system in a safe state
  - If it does -> resources are allocated
  - It it does not -> the process must wait until other process releases enough resources
- Support data structures
  - **Resource vector** – total amount of each resource in system
  - **Available vector** – amount of each resource not allocated
  - **Claim matrix** – total amount of resources that may be required by each process (stated before the process begins executing)
  - **Allocation matrix** – current allocation of resources to processes

Deadlock avoidance
Banker's algorithm

- **Determination of safe state**
  - Question: can any of the 4 processes run into completion with the available resources? I.e., is this a **safe state**?
    - What about P1?
    - What about P2?

P1 cannot run – there are no sufficient resources

	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

P2 can run into completion!

(a) Initial state

Deadlock avoidance
Banker's algorithm

- Question: can any of the 4 processes run into completion with the available resources? (cont.)
  - P2 runs to completion!
  - What about P1?

P2 completed and released all resources. Now P1 can finish!

Claim matrix C

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

Allocation matrix A

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

C - A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
6	2	3

(a) P2 runs to completion

Deadlock avoidance
Banker's algorithm

- Question: can any of the 4 processes run into completion with the available resources? (cont.)
  - P1 runs to completion!
  - What about P3?

P1 and P2 completed and released all resources. Now P3 can finish!

	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

(a) P1 runs to completion

Deadlock avoidance
Banker's algorithm

- Question: can any of the 4 processes run into completion with the available resources?
  - P3 runs to completion!

The system was on a **Safe State**

	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

P4 can also run until the end…

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(a) P3 runs to completion

Deadlock avoidance

Banker's algorithm

- **Deadlock Avoidance Strategy:**

  (To assure that the system is always in a safe state)

  - When a process makes a request for a set of resources, assume that the request is granted
  - Update the system state accordingly and then determine if the result is a **SAFE STATE**.
    - If so, grant the request.
    - If not, block the process until it is safe to grant the request.

Deadlock avoidance
Banker's algorithm

- **Determination of an unsafe state**
  - Consider the following 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	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

  - Assume that P1 makes a request: [1, 0, 1]
    - Should we give it those resources? I.e., does it lead to a safe state?

Deadlock avoidance
Banker's algorithm

- If request is granted, we will have:

	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

- **Answer: No**, because in this state each process will need at least an additional unit of R1 and there are none available!
  - Therefore, to **avoid** a **deadlock**, the request of P1 should be **denied**.
    - Maybe P1, P2, P3, P4 will not use any other R1 in the future, so the deadlock would not happen, but to avoid that possibility, the request is denied!

Deadlock detection

- If a system does not use deadlock-prevention or avoidance algorithms a deadlock may occur!
- Deadlock detection strategies grant resources whenever possible and perform an algorithm to detect if a deadlock has occurred – does not prevent the deadlock, only detects it
  - Run periodically by the OS
- An algorithm to recover from the deadlock is also needed

How to detect deadlocks?

Deadlock detection
Algorithm

- Strategy
  - Find processes whose resource requests can be satisfied
  - Grant the resources and assume the process runs to completion and releases resources
    - This is an optimistic approach since the process may later request additional resources. However, what we want to detect is if a deadlock exists at this moment!
  - Find another process and repeat

  - The algorithm does not prevent deadlocks! It only determines if they exist at the time the algorithm runs.

Deadlock detection
Algorithm

- Uses 1 vector and 2 matrices
  - Available
    - Vector that indicates the number of available resources of each type
  - Allocation
    - Matrix that has the number of resources of each type currently allocated to each process
  - Request
    - Matrix that has the current requests of each process

Deadlock detection
Algorithm

1. Mark each process that has a row in the Allocation Matrix (**A**) of all zeros.
   - **A process with no allocated resources does not participate in a deadlock**.

2. Initialize a temporary vector **W** (Work) to be equal to the Available Vector (that lists resources from **1** to **m**).

3. Find an index **i** such that process **i** is currently unmarked and the **i**th row of the Request Matrix **Q** is less than or equal to **W**.
   - That is, $\mathbf{Q_{ik} <= W_k}$ **(k=1…m)**. I.e. the process **i** can run with the available resources!
     - If no such row is found terminate the algorithm.

4. If such a row is found mark process **i** and add the corresponding row of the allocation matrix to **W**.
   - That is: set $\mathbf{W_k = W_k + A_{ik}}$ **(k=1…m)**. I.e. assume that on completion the process frees all the allocated resources.
   - Return to step 3.

Deadlock detection

Algorithm

- A **deadlock exists** if and only if there are unmarked processes at the end of the algorithm. The set of unmarked rows corresponds to the set of deadlocked processes.
  - These processes do not have sufficient resources available to fulfill their requests and cannot finish their execution

Deadlock detection
Algorithm - Example

	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

- **Mark**
- Set W = [0 0 0 0 1] (equals the Available vector)
- The request of process P3 is less than or equal to W, so **mark P3** and set:
  - W = W + [0 0 0 1 0] = [0 0 0 1 1]
    - This is the available vector after P3 finishes and releases all resources!
- No other unmarked process has a row in Q that is less than or equal to W. Therefore, terminate the algorithm.

P1 and P2 are unmarked, thus these processes are deadlocked!

Recovery from a deadlock

- Abort all deadlocked processes.
- Successively abort deadlocked processes until deadlock no longer exists.
- Successively preempt resources until deadlock no longer exists.

Deadlock Avoidance and Detection

EXERCISES

Exercise 1

- Consider a scenario where 4 processes are using 4 different system resources. The Request matrix (“Matriz de pedidos”), the Allocation matrix (“Matriz de recursos atribuídos”) and the Available resources vector (“Vector de recursos disponíveis”) are the following:

Allocation matrix A:

	R1	R2	R3	R4
P1	0	0	1	0
P2	0	0	1	1
P3	2	0	0	1
P4	0	1	2	0

Requests matrix Q:

	R1	R2	R3	R4
P1	1	0	2	0
P2	2	0	0	1
P3	1	0	1	1
P4	2	1	0	0

Available vector

R1	R2	R3	R4
2	1	0	0

Apply the deadlock detection algorithm and determine if there is a deadlock between the processes. Justify. (A binary answer is not enough, explain how the algorithm is applied).

Solution: There is a deadlock! Find out why!

Exercise 2

- Consider the following scenario where 4 processes use 4 different system resources. Consider the given claim, allocated and maximum resources matrixes given ("Matriz de pedidos","Matriz de recursos alocados no momento" and "Matriz de recursos máximos").
  - The maximum resource vector indicates the maximum number of resources of each type that exist in the system.
  - The maximum resources vector given is not the available resources vector!!

Requests matrix:

	R1	R2	R3	R4
P1	0	2	1	0
P2	0	2	0	0
P3	1	1	2	0
P4	0	0	2	1

Allocation resources matrix:

	R1	R2	R3	R4
P1	1	1	0	1
P2	0	1	0	1
P3	0	0	2	1
P4	1	0	0	0

Max resources vector:

R1	R2	R3	R4
2	3	4	4

a) Apply the deadlock detection algorithm and determine if there is a deadlock.

b) If the max resources vector was [2 4 4 4] what would be the result?

Exercise 3

- Consider a scenario with 5 processes (P1..P5) that use 4 different system resources (A,B,C,D) and the following allocation, claim and available matrices:

Claim

	A	B	C	D
P1	0	0	1	2
P2	1	7	5	0
P3	2	3	5	6
P4	0	6	5	2
P5	0	6	5	6

Allocation

	A	B	C	D
P1	0	0	1	2
P2	1	0	0	0
P3	1	3	5	4
P4	0	6	3	2
P5	0	0	1	4

Available

A	B	C	D
1	5	2	0

a) Apply the Banker's algorithm and determine if the system is in a safe state.

b) If P2 makes a new request (0,4,2,0), can the request be safely granted?

Exercise 3 – solution (summary)

- Consider a scenario with 5 processes (P1..P5) that use 4 different system resources (A,B,C,D) and the following allocation, claim and available matrices:

Claim

	A	B	C	D
P1	0	0	1	2
P2	1	7	5	0
P3	2	3	5	6
P4	0	6	5	2
P5	0	6	5	6

Allocation

	A	B	C	D
P1	0	0	1	2
P2	1	0	0	0
P3	1	3	5	4
P4	0	6	3	2
P5	0	0	1	4

Available

A	B	C	D
1	5	2	0

a) Apply the Banker's algorithm and determine if the system is in a safe state.

Is in a safe state

b) If P2 makes a new request (0,4,2,0), can the request be safely granted?

Yes

Exercise 4

- Consider a scenario with 5 processes (P1..P5) that use 4 shared different system resources (A,B,C,D) and the following allocation, claim (max resources needed by the process) and available matrices:

Allocation

	A	B	C	D
P1	0	0	1	2
P2	2	0	0	1
P3	0	1	1	2
P4	2	1	0	1
P5	0	0	2	1

Claim

	A	B	C	D
P1	1	1	1	2
P2	5	3	1	4
P3	3	1	2	3
P4	4	2	2	1
P5	1	1	3	3

Available

A	B	C	D
2	1	0	0

a) Apply the Banker's algorithm and determine if the system is in a safe state or in an unsafe state.

b) If P3 makes a new request (1,0,0,0), can the request be safely granted?

Exercise 5

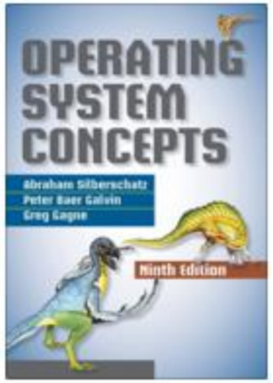
- Consider a scenario with 5 processes executing (P1..P5) and 3 different system resources (R1 to R3). The maximum number of resources provided by the system (is not the number of available ones now!) is: R1=10, R2=5, R3=7. Now, the system state is the following:

	Resources in use				Maximum resources needed		
	R1	R2	R3		R1	R2	R3
P1	0	1	0		7	5	3
P2	2	0	0		3	2	2
P3	3	0	2		9	0	2
P4	2	1	1		2	2	2
P5	0	0	2		4	3	3

a) Prove that the system is in a safe state.

b) Process P5 makes the request (R1=3, R2=3, R3=0). Which will be the answer of the resource manager, if *Banker's Algorithm* is applied? Explain the process used to obtain the answer.

References



- [Silberschatz13]
  - Chapter 7: Deadlocks

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling