



Operating Systems / Sistema de Operação

Deadlock

Artur Pereira / António Rui Borges

Deadlock

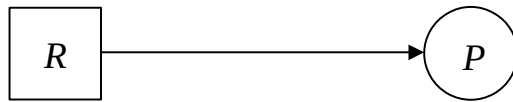
- Generically, a *resource* is something a process needs in order to proceed with its execution
 - *physical components of the computational system* (processor, memory, I/O devices, etc)
 - *common data structures* defined at the operating system level (PCT, communication channels, etc,) or among processes of a given application
- Resources can be:
 - *preemptable* – if they can be withdraw from the processes that hold them
 - ex: processor, memory regions used by a process address space
 - *non-preemptable* – if they can only be released by the processes that hold them
 - ex: a file, a shared memory region that requires exclusive access for its manipulation
- For this discussion, only non-preemptable resources are relevant

exemplo: a memoria é usada por varios processos mas nao pode ao mesmo tempo.
podem ser fisicos ou estruturas de dados

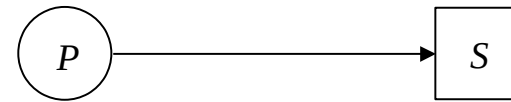
preemptable: é possivel tirar o recurso a quem o tem., por exemplo, quando esgota o time quantum.

Deadlock

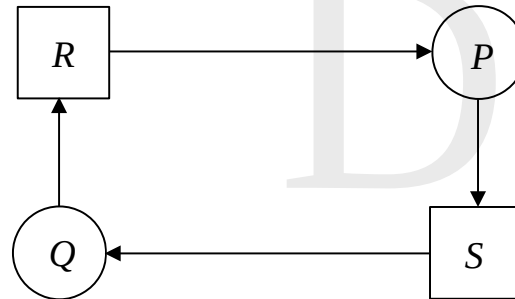
nos quadrados: recursos, nos circulares: processos



process P holds resource R in its possession



process P requests resource S



O P mantendo o r na sua posse quer o s

*typical deadlock situation
(the simplest one)*

o Q tento o s na sua posse, quer o r

- ♦ What are the conditions for the occurrence of deadlock

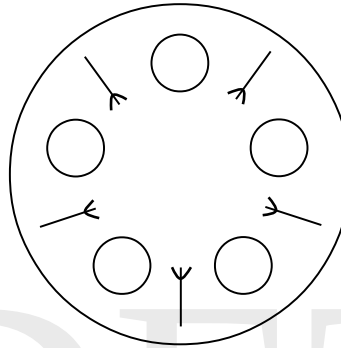
Necessary conditions for deadlock

se ha dedlock as 4 acontecem

- ♦ It can be proved that when *deadlock* occurs 4 conditions are necessarily observed:
 - ♦ *mutual exclusion* – at least one resource per process must be held in a nonsharable mode
 - ♦ if another process requests that, it must wait until it is released
 - ♦ *hold and wait* – a process must be holding at least one resource, while waiting for another that is being held by another process *tenho um. , quero outro, nao o largo.*
 - ♦ *no preemption* - resources are non-preemptable
 - ♦ only the process holding a resource can release it
 - ♦ *circular wait* - a set of waiting processes must exist such that each one is waiting for resources held by other processes in the set

quero recurso, esta na posse de outro, quero recurso, esta na posse de outro, quero ate chegar ao primeiro, again.

Necessary conditions for deadlock

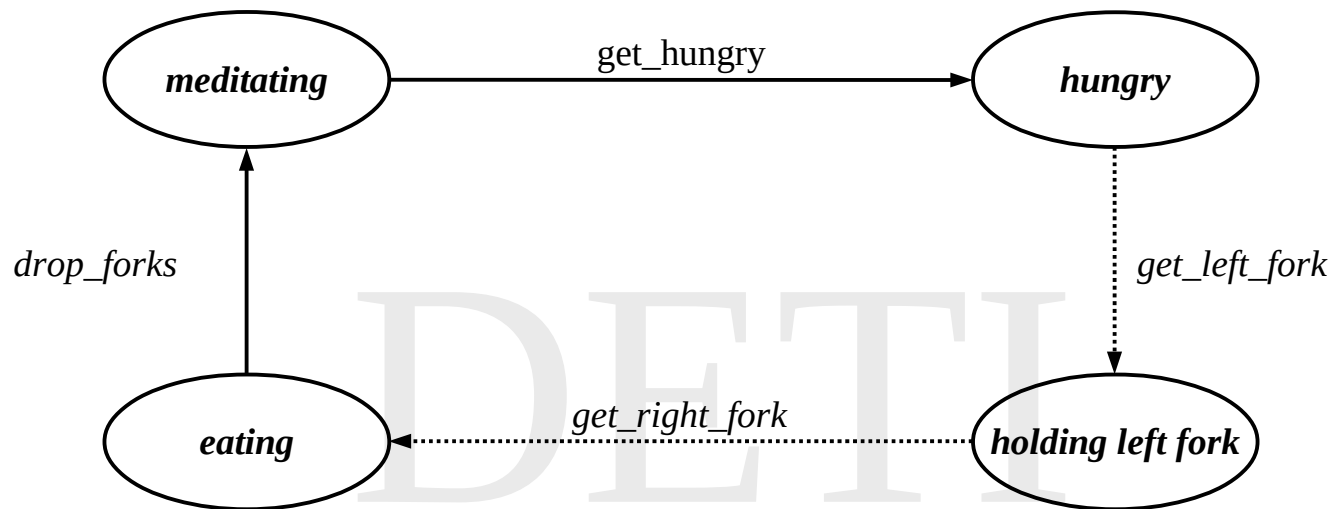


Problem statement

- 5 philosophers are seated around a table, with food in front of them
 - To eat, every philosopher needs two forks, the ones at her left and right sides
 - Every philosopher alternates periods in which she meditates with periods in which she eats
- Modelling every philosopher as a different process or thread and the forks as resources, design a solution for the problem

Necessary conditions for deadlock

cada garfo é um recurso



eu quero que esta funcao seja bloqueante, mas que nao queime tempo no processador

- This is a possible solution for the dining-philosopher problem
 - when a philosopher gets hungry, she first gets the left fork and then holds it while waits for the right one
- This solution can suffer from deadlock
 - Try to identify the four necessary conditions

meditate()

get hungry()

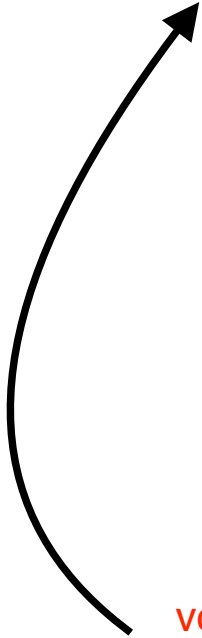
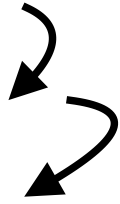
get left fork()

get right garfo()

comer()

drop garfo()

voltar ao meditate()



Necessary conditions for deadlock

```
enum PHILO_STATE {MEDITATING, HUNGRY, HOLDING, EATING};  
enum FORK_STATE {DROPPED, TAKEN};
```

```
typedef struct TablePlace  
{  
    int philo_state;  
    int fork_state;  
    cond fork_available;  
} TablePlace;
```

`malloc(size_of(table) + size_of(table_place))`

```
typedef struct Table  
{  
    mutex locker;  
    int nplaces;  
    TablePlace place[0];  
} Table;
```

- ♦ Let's look at different implementations of this solution!

```
int set_table(unsigned int n, FILE *logp);  
int get_hungry(unsigned int f);  
int get_left_fork(unsigned int f);  
int get_right_fork(unsigned int f);  
int drop_forks(unsigned int f);
```

Necessary conditions for deadlock

- ♦ This solution works some times
 - ♦ But, it can suffer from deadlock
- ♦ The four necessary conditions for the occurrence of deadlock are satisfied
 - ♦ *mutual exclusion* – the forks are not sharable
 - ♦ *hold and wait* – each philosopher while waiting to acquire the right fork holds the left one
 - ♦ *no preemption* – each philosopher keeps the forks until she finishes eating
 - ♦ *circular wait* – if every philosopher can acquire the left fork, there is a chain in which every philosopher waits for a fork in possession of another philosopher

Deadlock prevention

se A implica B entao nao B implica naoA

deadlock \Rightarrow *mutual exclusion* **and**
hold and wait **and**
no preemption **and**
circular wait

- ♦ that is equivalent to

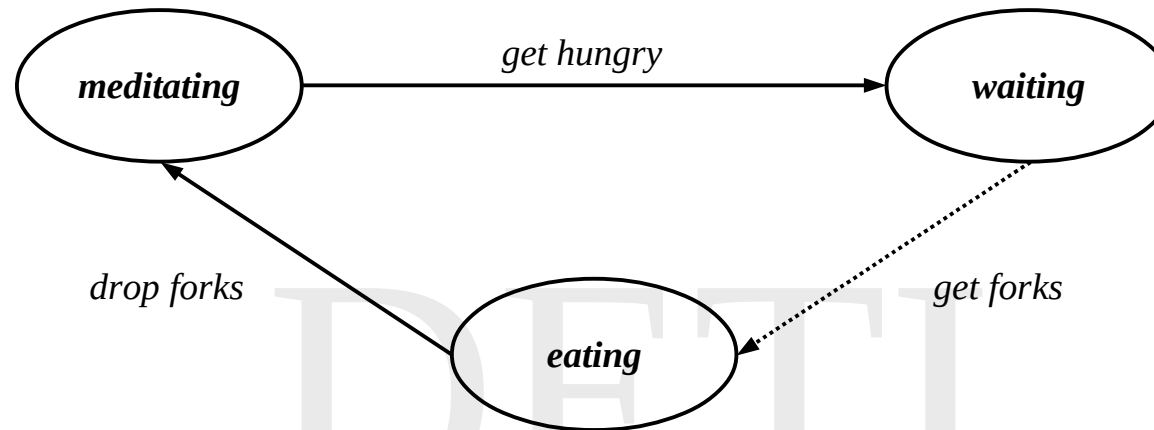
not *mutual exclusion* **or**
not *hold and wait* **or**
not *no preemption* **or**
not *circular wait* \Rightarrow **not** *deadlock*

- ♦ So, if at least one of the necessary condition does not hold, there is no deadlock
- ♦ This is called *deadlock prevention*

Deadlock prevention

- ♦ Denying the *mutual exclusion* condition is only possible if the resources are shareable
 - ♦ Otherwise *race conditions* can occur
 - ♦ In the dining-philosopher problem, the forks are not shareable
- ♦ Thus, in general, only the other conditions are used to implement deadlock prevention
- ♦ Denying the *hold-and-wait* condition can be done if a process requests all required resources at once
 - ♦ In the dining-philosopher problem, the two forks must be acquired at once
 - ♦ In this solution, *starvation* can occur
 - ♦ *Aging* mechanisms are often used to solve starvation

Deadlock prevention



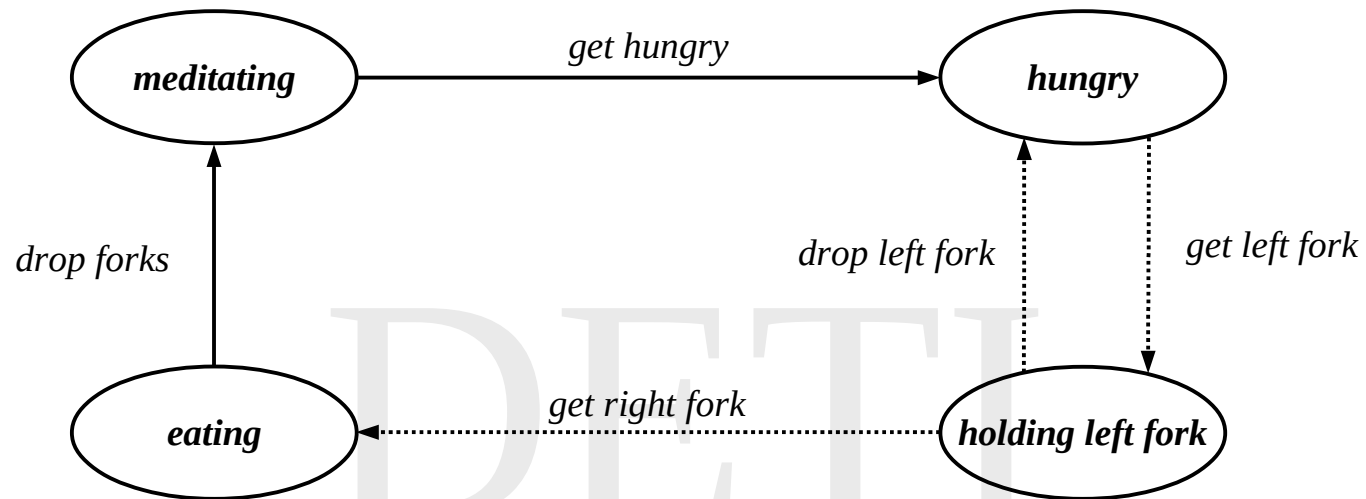
Text

- This solution is equivalent to the one proposed by Dijkstra
- Every philosopher, when want to eat acquire the two forks at the same time
- If they are not available, she waits in the waiting state
- Starvation is not avoided

Deadlock prevention

- ♦ Denying the *hold and wait* condition can also be done if a process releases the already acquired resources if it fails acquiring the next one
 - ♦ Later on she can try the acquisition again
 - ♦ In the dining-philosopher problem, a philosopher must release the left fork if she fails acquiring the right one
 - ♦ In this solution, *starvation* and *busy waiting* can occur
 - ♦ *Aging* mechanisms are often used to solve starvation
 - ♦ To avoid busy waiting, the process should block and be waked up when the resource is released

Deadlock prevention

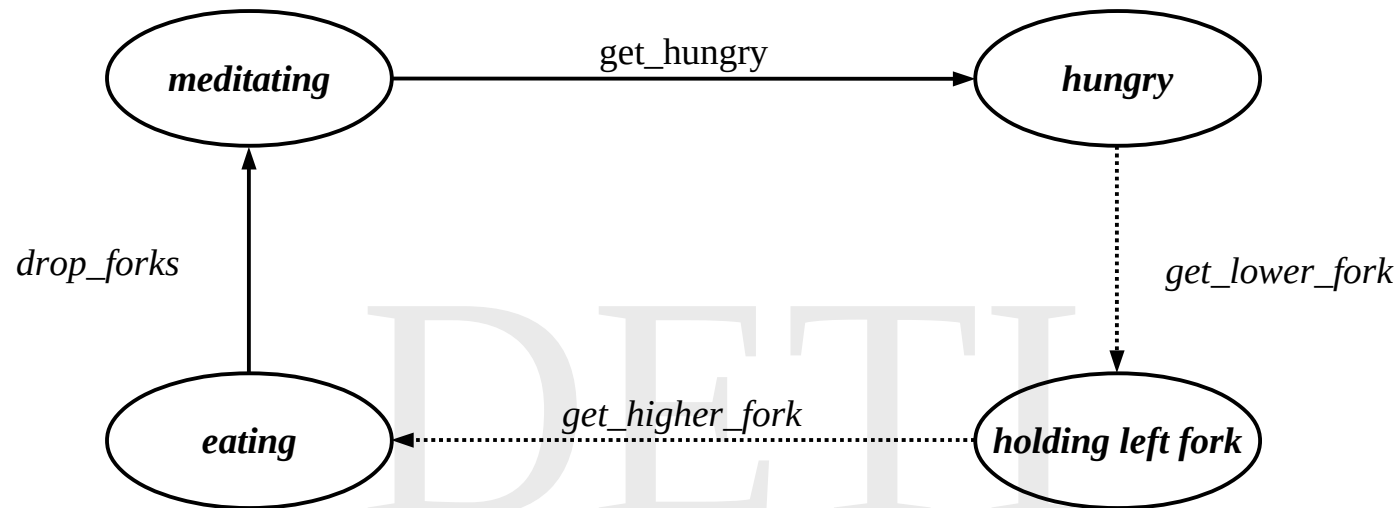


- When a philosopher gets hungry, she first acquire the left fork
- Then she tries to acquired the right one, releasing the left if she fails and returning to the hungry state
- *busy waiting* and starvation were not avoided in this solution

Deadlock prevention

- ♦ Denying the *circular wait* condition can be done assigning a different numeric id to every resource and imposing that the acquisition of resources have to be done either in ascending or descending order
 - ♦ This way the circular chain is always avoided
 - ♦ Starvation is not avoided
- ♦ In the dining-philosopher problem, one of the philosophers acquire first the right fork and then the left one

Deadlock prevention



- Philosophers are numbered from 0 to N
- Every fork is assigned an id, equal to the id of the philosopher at its right. for instance
- Every philosopher, acquires first the fork with the lower id
- This way, philosophers 0 to N-2 acquire first the left fork, while philosopher N-1 acquires first the right one

Deadlock prevention

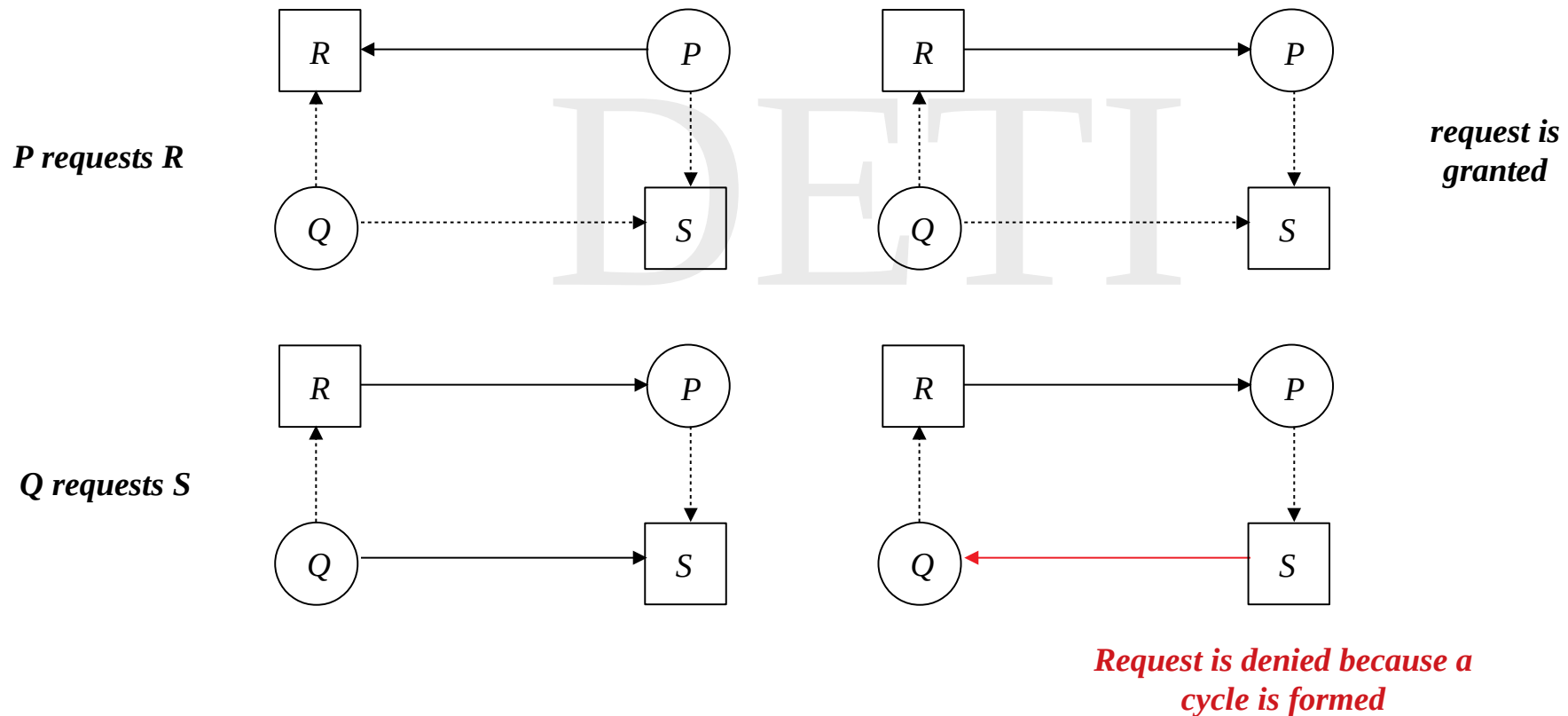
- ♦ Deadlock prevention policies are in general quite restrictive, not efficient and hard to apply in many situations
 - ♦ *denying mutual exclusion* – can only be applied to shareable resources
 - ♦ *denying hold and wait* – requires the a priori knowledge of all the necessary resources and always consider the worst case (all resources simultaneously)
 - ♦ *denying no preemption* – imposing the release and re-acquisition of resources; can introduce long delays in the processing the task
 - ♦ *denying circular wait* – can introduce a bad use of resources

Deadlock avoidance

- ♦ *Deadlock avoidance* is less restrictive than deadlock prevention
 - ♦ None of the necessary conditions is denied
 - ♦ Instead, the system is monitored continuously and a resource request is only granted if the system does not enter an unsafe state in consequence
- ♦ A state is said to be safe if there is a sequence of assignments of resources such that all intervening processes do terminate (no deadlock)
 - ♦ Otherwise it is assumed to be unsafe
- ♦ drawbacks:
 - ♦ the list of all resources must be known
 - ♦ the intervening processes have to declare at start their needs in terms of resources
- ♦ Note that unsafe does not mean deadlock
 - ♦ worst case conditions are considered

Deadlock avoidance

- In case there is a single instance per resource type:
 - Use resource allocation graph algorithm



Deadlock avoidance

- In case there are multiple instances per resource type
- Let

NTR_i – be the total no. of resources of type i (with $i = 0,1,...,N-1$)

$R_{i,j}$ – be the no. of resources of type i required by process j

(with $i = 0,1,...,N-1, j = 0,1,...,M-1$)

- The system can prevent a new process M to start if its termination can not be guaranteed

- It is only launched if
$$\forall_i: R_{i,M} \leq NTR_i - \sum_{j=0}^{M-1} R_{i,j}$$

Deadlock avoidance

- Let

NTR_i – be the total no. of resources of type i (with $i = 0, 1, \dots, N-1$)

$R_{i,j}$ – be the no. of resources of type i required by process j

(with $i = 0, 1, \dots, N-1$, $j = 0, 1, \dots, M-1$)

$A_{i,j}$ – be the no. of resources of type i assigned to process j

(with $i = 0, 1, \dots, N-1$, $j = 0, 1, \dots, M-1$)

- A new resource of type i is only assigned to a process if and only if there is a sequence $s(k) = f(i, j)$ such that

$$\forall_k \quad \forall_i : R_{i,s(k)} - A_{i,s(k)} \leq NTR_i - \sum_{j \geq k}^{M-1} A_{i,s(j)}$$

- This approach is called the **banker's algorithm**

Banker's algorithm

		A	B	C	D
	total	6	5	7	6
	free	3	1	1	2
maximum	p1	3	3	2	2
	p2	1	2	3	4
	p3	1	3	5	0
granted	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
needed	p1	2	1	0	1
	p2	0	2	0	1
	p3	0	1	4	0
new Grant	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	0	0	0

- ♦ Is this a safe state?

Banker's algorithm

		A	B	C	D
	total	6	5	7	6
	free	3	1	1	2
maximum	p1	3	3	2	2
	p2	1	2	3	4
	p3	1	3	5	0
granted	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
needed	p1	2	1	0	1
	p2	0	2	0	1
	p3	0	1	4	0
new Grant	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	0	0	0

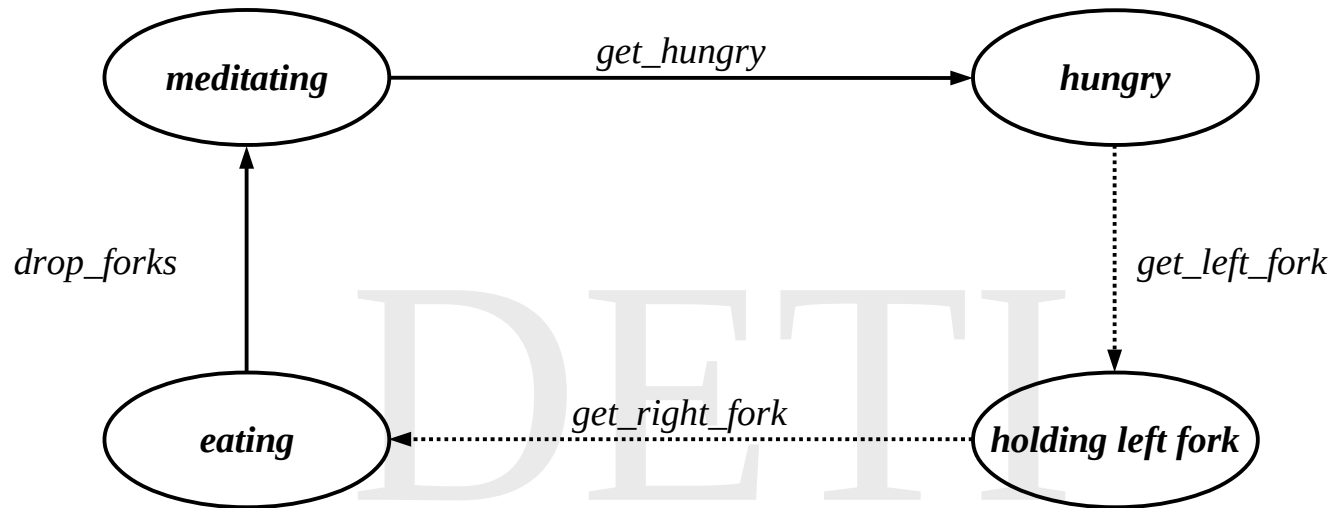
- ♦ If p3 requests 2 resources of type C, the grant has to wait
 - ♦ Because only 1 is available
- ♦ If p3 requests 1 resource of type B, the grant is postponed
 - ♦ Why?

Banker's algorithm

		A	B	C	D
	total	6	5	7	6
	free	3	0	1	2
maximum	p1	3	3	2	2
	p2	1	2	3	4
	p3	1	3	5	0
granted	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
needed	p1	2	1	0	1
	p2	0	2	0	1
	p3	0	0	4	0
new Grant	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	1	0	0

- ♦ If p3 requests 1 resource of type B, the grant is postponed
 - ♦ Because, even if there is one available, the system in case the grant is done enters an unsafe state

Banker's algorithm



- ♦ Every philosopher first gets the left fork and then gets the right one
- ♦ However, in a specific situation the request of the left fork can be denied
 - ♦ What situation? Why?

Deadlock detection

- ♦ No deadlock-prevention or deadlock-avoidance is used
 - ♦ So, deadlock situations may occur
- ♦ In these cases
 - ♦ The state of the system should be examined to determine whether a deadlock has occurred
 - ♦ In such a case, a recover procedure from deadlock should exist and be applied
- ♦ In a more naive approach
 - ♦ The problem can simply be ignored

Deadlock detection

- ♦ If deadlock has occurred, the circular chain of processes and resources need to be broken
- ♦ This can be done:
 - ♦ *release resources from a process* – if it is possible
 - ♦ The process is suspended until the resource can be returned back
 - ♦ Efficient but requires the possibility of saving the process state
 - ♦ *rollback* – if the states of execution of the different processes is periodically saved
 - ♦ A resource is released from a process, whose state of execution is rolled back to the time the resource was assigned to it
 - ♦ *kill processes* –
 - ♦ Radical but easy to implement method

Bibliography

Operating Systems Concepts, Silberschatz, Galvin, Gagne, John Wiley & Sons, 9th Ed

- Chapter 7: *Deadlocks* (sections 7.1 to 7.6)

Modern Operating Systems; Tanenbaum & Bos; Prentice-Hall International Editions, 4rd Ed

- Chapter 6: *Deadlocks* (sections 6.1 to 6.6)

Operating Systems, Stallings, Prentice-Hall International Editions, 7th Ed

- Chapter 6: *Concurrency: deadlock and starvation* (sections 6.1 to 6.7)