Sistemas de Operação / Fundamentos de Sistemas Operativos

Deadlock

Artur Pereira <artur@ua.pt>

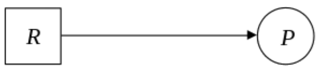
DETI / Universidade de Aveiro

Outline

Deadlock
Introduction

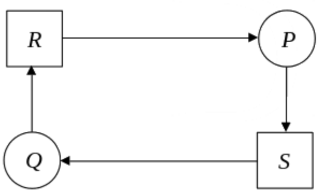
- Generically, a resource is something a process needs in order to proceed with its execution
  - physical components of the computational system (processor, memory, I/O devices, etc.)
  - common data structures defined at the operating system level (PCT, communication channels, etc,) or among processes 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 printer, a binary semaphore
- For this topic, only non-preemptable resources are relevant

Deadlock
Illustrating deadlock



process P holds resource R in its possession

process P requests resource S

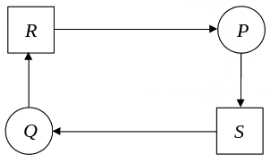


typical deadlock situation
(the simplest one)

- P needs S to proceed, which is on possession of Q
- Q needs R to proceed, which is on possession of P
- What are the conditions for the occurrence of deadlock?

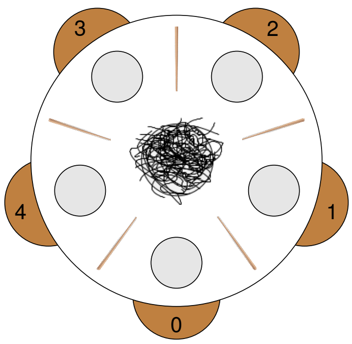
Deadlock
Necessary conditions for deadlock

- It can be proved that when deadlock occurs 4 conditions are necessarely observed:
  - mutual exclusion – only one process may use a resource at a time
    - if another process requests it, it must wait until it is released
  - hold and wait – A process holds resources in its possession while waiting for another
  - no preemption – resources are non-preemptable
    - only the process holding a resource can release it (probably after completing the task that requires 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
    - there are loops in the graph



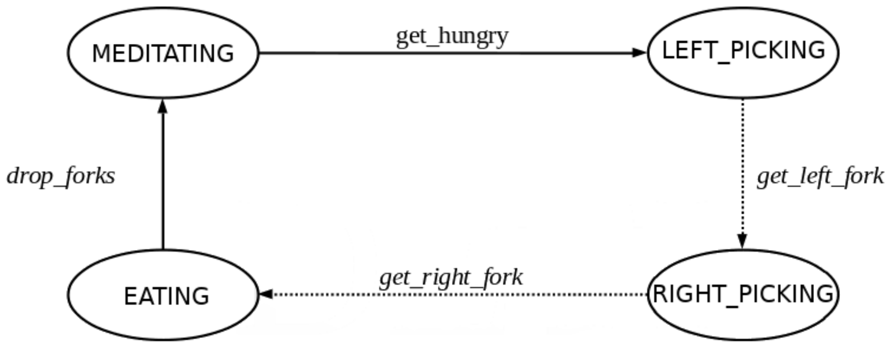
typical deadlock situation
(*the simplest one*)

Deadlock
Ilustrating with the Dining Philosophers Problem



- 5 philosophers are seated around a table, with noodles in from of them
  - To eat, every philosopher needs two chopsticks, the ones at her/his left and right sides
  - Every philosopher alternates periods in which she/he medidates with periods in which she/he eats
- Modeling every philosopher as a different process or thread and the chopsticks as resources, design a solution for the problem

Dining philosophers
Solution 1 – state diagram



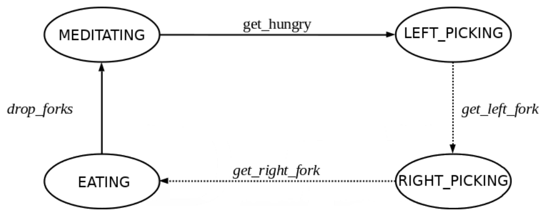
- This is a possible solution for the dining philosophers problem
  - when a philosopher gets hungry, he/she first gets the left fork and then holds it while waits for the right one
- Let's look at an implementations of this solution!

Dining philosophers
Solution 1 – code

```
enum PHILO_STATE {MEDITATING, LEFT_PICKING, RIGHT_PICKING, EATING};
enum FORK_STATE {DROPPED, TAKEN};

typedef struct TablePlace
{
    int philo_state;
    int fork_state;
    pthread_cond_t fork_available;
} TablePlace;

typedef struct Table
{
        pthread_mutex_t access;
        int nplaces;
        TablePlace place[0];
} Table;

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);
```

- Let's execute the code

Dining philosophers
Solution 1 – deadlock conditions



- This solution works some times, but can suffer from deadlock
- Let's identify the four necessary conditions
  - mutual exclusion – the forks are not sharable at the same time
  - hold and wait – each philosopher, while waiting to acquire the right fork, holds the left one
  - no preemption – only the philosophers can release the fork(s) in their possession
  - circular wait – if all philosopher acquire the left fork, there is a chain in which every philosopher waits for a fork in possession of another philosopher

Deadlock analysis
Make deadlock not occur

- From the definition

  deadlock $\Longrightarrow$
  mutual exclusion and hold and wait and
  no preemption and circular wait
- Which is equivalent to

  not mutual exclusion or not hold and wait or
  not no preemtion or not circular wait
  $\Longrightarrow$ not deadlock
- So, if at least one of the conditions can never hold, there is no possibility of deadlock
- This is called deadlock prevention
  - If is the responsibility of the concurrent application to ensure deadlock does not occur
- Or deadlock avoidance
  - If is the responsibility of a resource manager to ensure deadlock does not occur

Deadlock prevention
Denying the necessary conditions

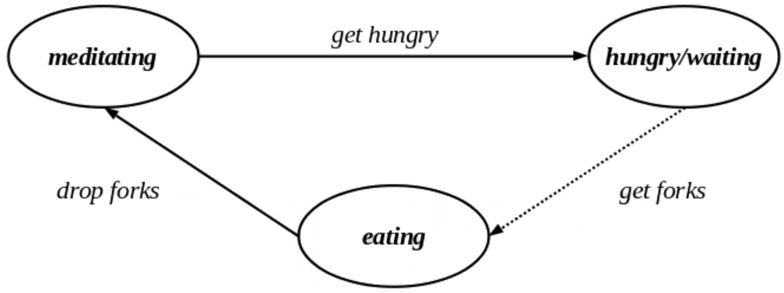
- Denying the mutual exclusion condition is only possible if resources are shareable at the same time
  - Otherwise race conditions can occur
- Denying the preemption condition is only possible if resources are preemptable
  - Which is often not the case
- Thus, in general, only the other conditions (hold-and-wait and circular wait) are used to implement deadlock prevention

- In the dining-philosopher problem, the forks are not shareable at the same time
- In the dining-philosopher problem, a fork cannot be taken away from whoever is holding it

Deadlock prevention
Denying the necessary conditions (2)

- Avoiding the hold-and-wait condition can be done if a process requests all required resources at once
  - In this solution, starvation can occur
  - Aging mechanisms are often used to solve starvation
- In the dining-philosopher problem, the two forks can be acquired at once

Dining philosophers
Solution 2 – state diagram

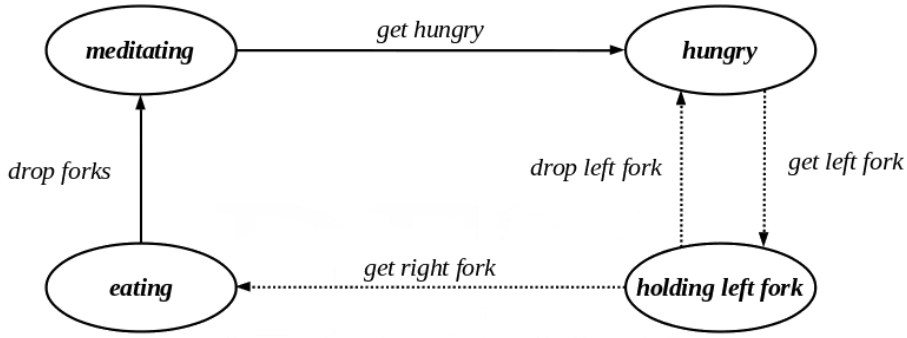


- This solution is equivalent to the one proposed by Dijkstra
- Every philosopher, when wants to eat, gets the two forks at the same time
- If they are not available, the philosopher waits in the hungry/waiting state
- Starvation can occur

Deadlock prevention
Denying the necessary conditions (3)

- Denying the hold and wait condition can also be done if a process releases the already acquired resource(s) if it fails acquiring the next one
  - Later on it can try the acquition again
- In the dining philosophers problem, a philosopher can release the left fork if she/he fails acquiring the right one
- In this kind of solutions, starvation and busy waiting may 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

Dining philosophers
Solution 3 – state diagram

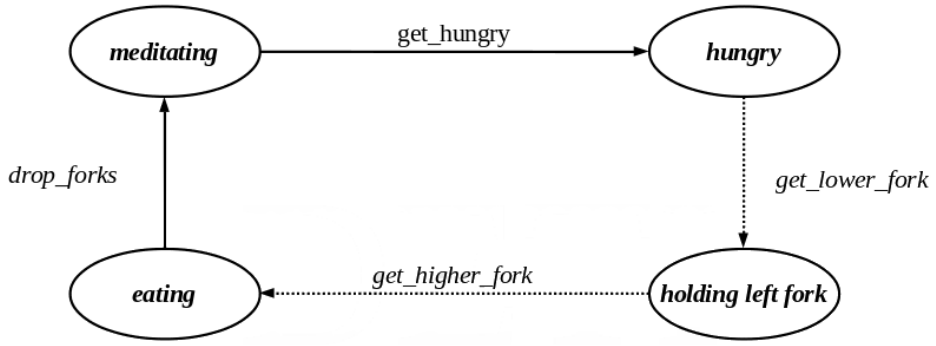


- When a philosopher gets hungry, she/he first acquire the left fork
- Then she/he tries to acquired the right one, releasing the left if she/he fails and returning to the hungry state
  - A `trydown` or `trywait` synchronization primitive is required
- busy waiting and starvation are not avoided in this solution

Deadlock prevention
Denying the necessary conditions (4)

- 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, this can be done imposing that one of the philosophers acquires first the right fork and then the left one
  - Show that this corresponds to impose that the acquisition of resources are done either in ascending or descending order!

Dining philosophers
Solution 4 – state diagram



- Philosophers are numbered from 0 to $N-1$
- Every fork is assigned an id, equal to the id of the philisopher at its left, 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 avoidance
Definition

- Deadlock avoidance is less restrictive than deadlock prevention
  - Deadlock conditions are not denied from the application side
  - There is a resources' manager that decide what to do in terms of resource allocation
  - Requires prior knowledge of the processes' maximum resource requests
    - The intervening processes have to declare at start their maximum needs in terms of resources
- Two possible approaches
  - Process Initiation Denial
    - Do not start a process if its demands might lead to deadlock
  - Resource Allocation Denial
    - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Deadlock avoidance
Process initiation denial

- The system prevents a new process to start if its termination can not be guaranteed
- Let
  - $R = (R_1, R_2, \cdots, R_n)$ be a vector of the total amount of each resource
  - P be the set of processes competing for resources
  - C_p be a vector of the total amount of each resource declared by process $p \in P$
- A new process q ($q \notin P$) is only started if

$$C_q \leq R - \sum_{p \in P} C_p$$

- It is a quite restrictive approach

Deadlock avoidance
Resource allocation denial

- A new resource is allocated to a process if and only if there is at least one sequence of future allocations that does not result in deadlock
  - In such cases, the system is said to be in a safe state
- Let
  - $R = (R_1, R_2, \cdots, R_n)$ be a vector of the total amount of each resource
  - P be the set of processes competing for resources
  - C_p be a vector of the total amount of each resource declared by process $p \in P$
  - $V = (V_1, V_2, \cdots, V_n)$ be a vector of the amount of each resource available
  - A_p be a vector of the amount of each resource already allocated to process $p \in P$
- A new request of a process q is only granted if, after it, there is a sequence $s(k)$, with $s(k) \in P$ and $k = 1, 2, \cdots, |P|$, of processes, such that

$$C_{s(k)} - A_{s(k)} = V + \sum_{m=1}^{k-1} A_{s(m)}$$

- This approach is called the banker's algorithm

Deadlock avoidance
Banker's algorithm

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0

- Consider the system state described by the table. Is it a safe state?
  - P2 may still request 2 R2, but only one is available
  - P3 may still request 4 R3, but only one is available
  - All resources that P1 can still request are available

Deadlock avoidance
Banker's algorithm (2)

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0
new request	–	–	–	–	–

- Consider the following sequence:
  - P1 requests all the resources it can still; the request is granted; then terminates
  - P2 requests all the resources it can still; the request is granted; then terminates
  - P3 requests all the resources it can still; the request is granted; then terminates

Deadlock avoidance
Banker's algorithm (3)

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0
new request	P3	0	0	2	0

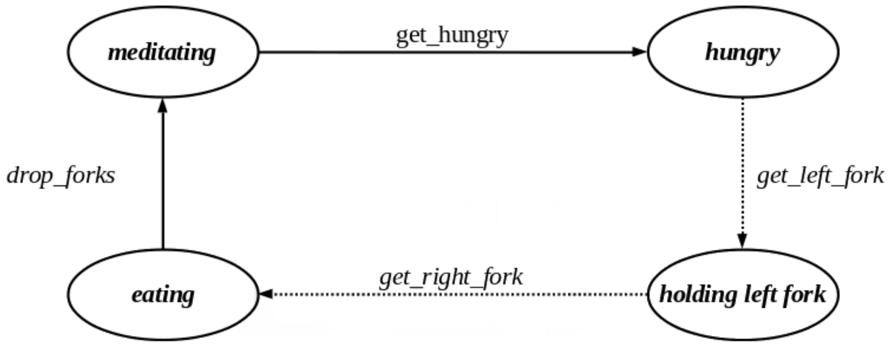
- If P3 requests 2 resources of type R3, the grant is postponed. Why?
  - Because only 1 is available

Deadlock avoidance
Banker's algorithm (4)

		R1	R2	R3	R4
total resources		6	5	7	6
available resources		3	1	1	2
resources declared	P1	3	3	2	2
	P2	1	2	3	4
	P3	1	3	5	0
resources allocated	P1	1	2	2	1
	P2	1	0	3	3
	P3	1	2	1	0
resources requestable	P1	2	1	0	1
	P2	0	2	0	1
	P3	0	1	4	0
new request	P3	0	1	0	0

- If P3 requests 1 resource of type R2, the grant is also postponed. Why?
  - Because, if the grant is given, the system transitions to an unsafe state. Show it.

Deadlock avoidance
Banker's algorithm - example



- Every philosopher first gets the left fork and then gets the right one
- However, in a specific situation the request of the left fork is postponed
  - What situation? Why?

Deadlock detection
Definition

- No deadlock-prevention or deadlock-avoidance is used
  - So, deadlock situations may occur
- The state of the system should be examined to determine whether a deadlock has occurred
- A recover from deadlock procedure should exist and be applied

- What to do?
  - In a quite naive approach, the problem can simply be ignored
  - Otherwise, the circular chain of processes and resources need to be broken

Deadlock detection
Recover procedure

- How?
  - 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 an easy to implement method

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
  - Chapter 6: Concurrency: deadlock and starvation (sections 6.1 to 6.7)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
  - Chapter 7: Deadlocks (sections 7.1 to 7.6)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
  - Chapter 6: Deadlocks (section 6.1 to 6.6)