5. Deadlocks

Sunday, September 13, 2020 3:11 PM

Deadlock concept

Definition

A deadlock is the situation when a group of processes hold and wait for resources from one another, results in no processes will ever have enough resources to complete the task.

Example: Dining philosophers problem, each one has a chopstick and waits for the other. No philosophers have enough chopsticks to start eating.

Necessary and sufficient conditions for a deadlock

A deadlock happens if and only if the 4 following conditions hold

1) Mutual Exclusion
   The resource has to be exclusive, i.e not accessible by multiple processes at a time. If a process is holding the resource, other processes have to wait.

2) Hold and Wait
   Once a resource is allocated to a process, the process will hold the resource and wait for other resources needed for its completion.

3) No preemption
   Processes do not have priority in requesting for resource allocation. A resource already allocated to a process cannot be deprived and given to another process.

4) Circular wait
   Processes waiting for a shared resource form up a cycle, therefore no process will ever have enough resources to complete the task.

Methods to deal with deadlocks

To deal with deadlocks, we may use one of the following methods

1) Detection and recovery: We accept that deadlocks may occur, and detect deadlocks when they happen, then try to recover the system from deadlocks.

2) Prevent deadlocks by strictly monitoring all resource allocation requests

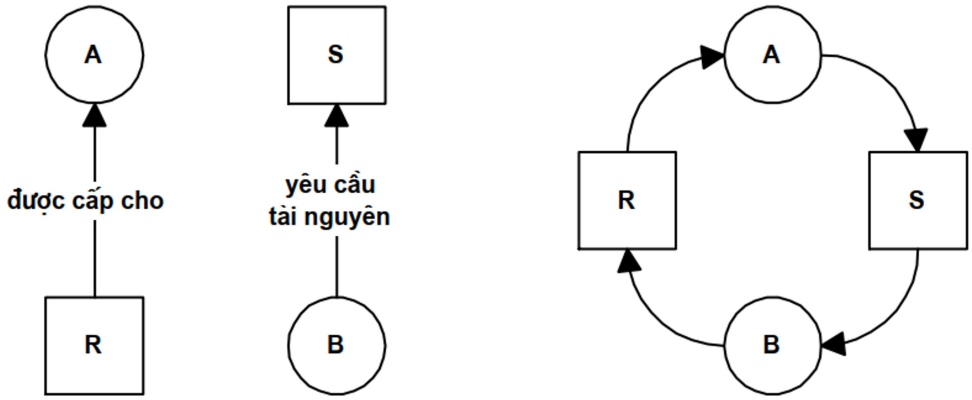
3) Prevent deadlocks by breaking one of the 4 necessary and sufficent conditions for a deadlock

4) Ostrich method: Doing nothing. This method is not bad for a normal system because
- The probability for a deadlock is low
- If it happens, the consequence is not so serious
- Cost for dealing with deadlocks is high

Operating systems such as Windows, Linux…, all use Ostrich method.

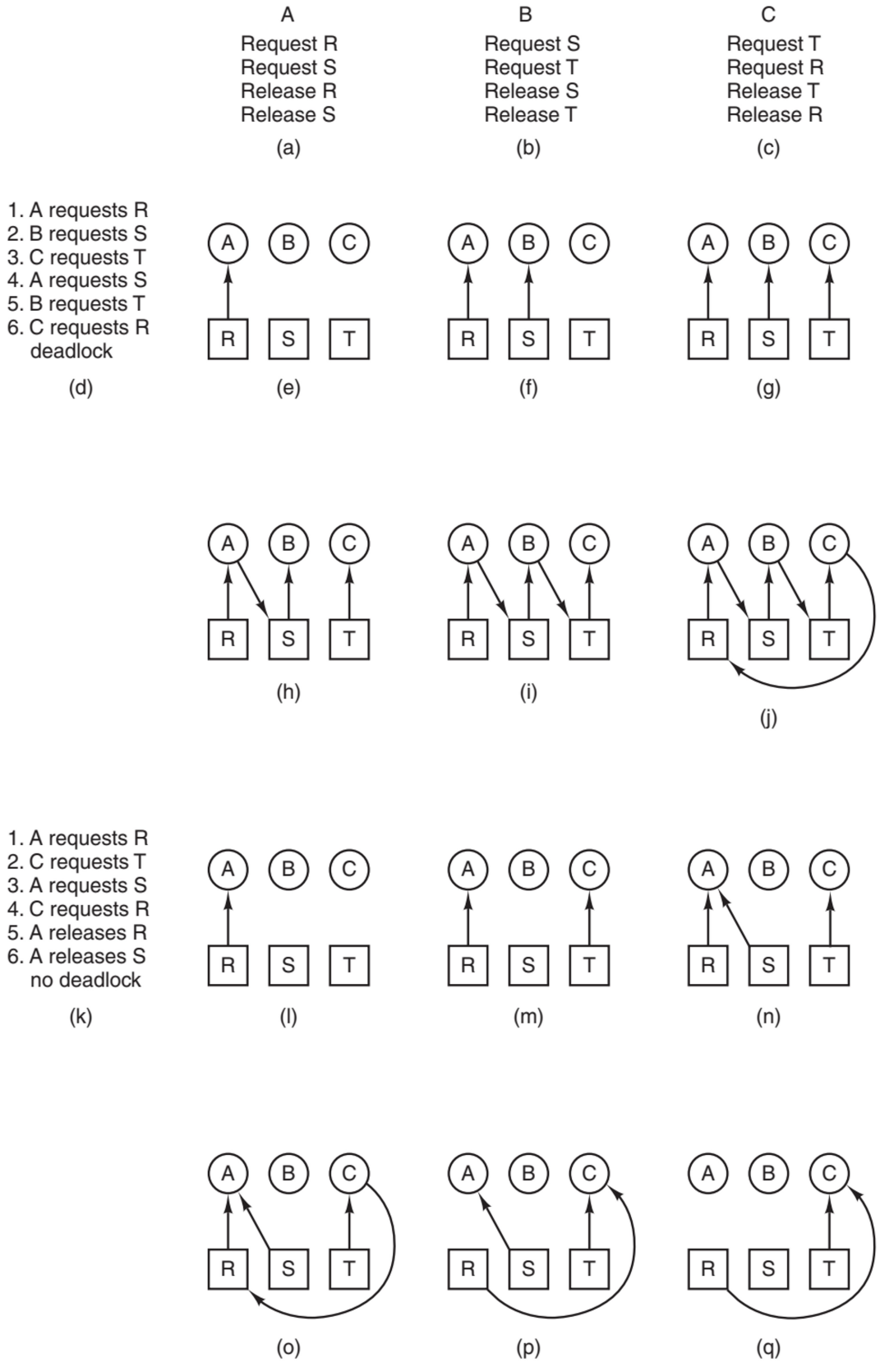
Detection and recovery

Resource allocation graph



OS maintains a resource allocation graph, with 2 types of vertices: Process and Resource. Edges represent the state of resource allocation.

Example: deadlocks could be detected through the existence of cycles in the resource allocation graph.



Cycle detection and removal

A cycle once detected can be eliminated by removing a process vertex in the cycle.
Remarks:

- The allocation graph has to be updated and a cycle detection algorithm has to be run after each resource allocation/de-allocation
- Removing a running process may cause data to lose its integrity and previous state cannot be recovered

Deadlock prevention by strictly monitoring resource allocation

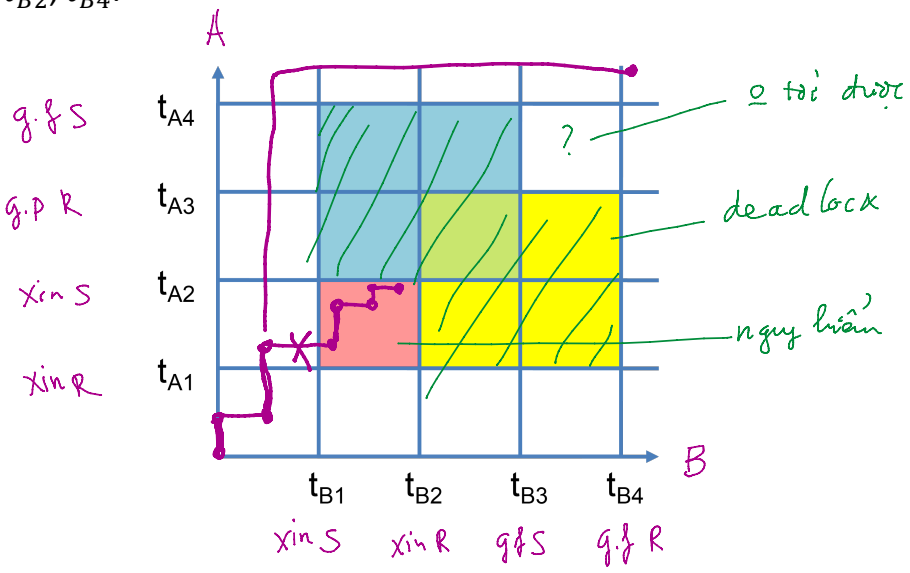
Monitoring CPU allocation

Resource allocation to a process that leads to a deadlock can be avoided by not giving CPU to that process. We use a multi-axis coordination system to represent CPU time allocation to each process. The CPU allocation graph starts at the root and goes along each axis direction. As time cannot be reversed, the graph does not go back.
Allocation/deallocation milestones on each axis will divide the space into regions of different categories:

- Safe: there exists at least a solution for CPU allocation that meets all processes' resource requests
- Dangerous: there is no way to allocate CPU time to meets all the resource requests
- Deadlock: represents the deadlock situation
- Unreachable: the CPU allocation graph cannot reach to these regions.

The job of the operating system is to "drive" the CPU allocation graph through safe regions and satisfy all processes' requests.

Illustration with 2 processes A and B. A requests and releases resource R at t_{A1}, t_{A3}; and with resource S at t_{A2}, t_{A4}. B requests and releases resource S at t_{B1}, t_{B3}; and with resource R at t_{B2}, t_{B4}.



Remarks: Intuitive but not feasible because

- It is not realistic to know the exact time of resource allocation and de-allocation before the process execution
- The number of processes are variable so the regions are also changing variably

Using banker's algorithm to monitor resource allocation for one type of resources

We model the resource allocation as a bank operations as follows
- The bank accepts only one currency (1 type of resources)
- The list of clients are fixed (no new processes)
- Each client has to inform the amount of loan in advance (number of resources requested)
- Once receiving the entire loan amount, the client will complete her task and pay back the entire loan (return the resources at completion time)

The bank state is defined as the current loans and the bank's balance, for example

	Có	Max
A	3	9
B	2	4
C	2	7

còn 3

A state is safe if there is at least a way to meet all clients' requests. Otherwise the state is unsafe and will result in a deadlock.

Banker's algorithm checks if the next state is safe, if yes, the request will be approved. Otherwise the request will be ignored.

Example: A requests for one resource. If this request is fulfilled, the bank will be in an unsafe state and result in a deadlock.

	Có	Max
A	3	9
B	2	4
C	2	7

còn 3

	Có	Max
A	4	9
B	2	4
C	2	7

còn 2

	Có	Max
A	4	9
B	4	4
C	2	7

còn 0

	Có	Max
A	4	9
B	0	-
C	2	7

còn 4

Banker's algorithm for multiple types of resources

Allocation

R_1	R_2	R_3
0	2	0
1	0	0
2	0	1
1	1	2
0	0	2

Max

R_1	R_2	R_3
7	2	3
1	2	3
5	1	1
1	1	1
4	2	1

Available

R_1	R_2	R_3
3	1	1

Assume that we have n processes and m types of resources. Let $\boldsymbol{Allocation}$ is the matrix of allocated resources, $\boldsymbol{Max}$ is the matrix of maximum resource requests from the processes. Both matrices are of size $n \times m$. $Available$ is a vector of m elements, representing for the remaining resources of each type.

For the convenience, we calculate matrix $\boldsymbol{Need} \coloneqq \boldsymbol{Max} - \boldsymbol{Allocation}$.

The bank state is defined as a tuple of $\boldsymbol{Allocation}$, $\boldsymbol{Need}$ and $Available$.

Banker's algorithm for multiple types of resources

```
count=0;
do { // loop until all processes are fulfilled or no more possible allocation
        found=FALSE;
        for (i=0; i<numProcesses; i++) {
                if ( satisfied[i]==FALSE && Need[i] <= Available) { // possible allocation
                        available += Allocation[i];  // get back the allocated resources
                        satisfied[i]=TRUE;           // mark as fulfilled
                        count++;                     // count the number of fulfilled processes
                        found=TRUE;                  // an allocation is done
                }
        }
} while (count<numProcesses && found);

if (count<numProcesses) return FALSE;   // unsafe state
```

else return TRUE; *// safe state*

Example 1: 5 processes, 3 types of resources

Initial state

Process	Allocation	Need	Satisfied
0	0 2 0	7 2 3	F
1	1 0 0	1 2 3	F
2	2 0 1	5 1 1	F
3	1 1 2	1 1 1	F
4	0 0 2	4 2 1	F

Available = (3 1 1)

Safe allocation sequence P3 -> P1 -> P2 -> P0 -> P4

Example 2: 5 proceses, 3 types of resources

Initial state

Process	Allocation	Need	Satisfied
0	0 2 0	7 2 3	F
1	1 0 0	1 2 3	F
2	2 0 1	5 1 1	F
3	1 1 2	1 1 1	F
4	0 0 2	4 2 1	F

Available = (2 1 1)

No possible allocation sequence.

Remarks: The algorithm has some limitations, the number of processes has to be fixed, the number of resources to be requested has to be known in advance. The algorithm takes (O(n^2)) time to check for a safe state.

Deadlock prevention by breaking one of the 4 necessary conditions

Breaking mutex condition

Breaking mutual exclusion condition by using spooling technique to sequentialize resource requests.
Spooling technique is often used to improve CPU usage when a high speed device

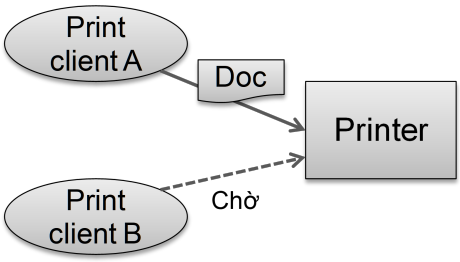
transfers data to a low speed device. Then

- Data from the high speed device is stored in a buffer
- The low speed device will process data sequentially from the buffer

Spooling technique reduces CPU idling time waiting for data synchronization between the two devices.

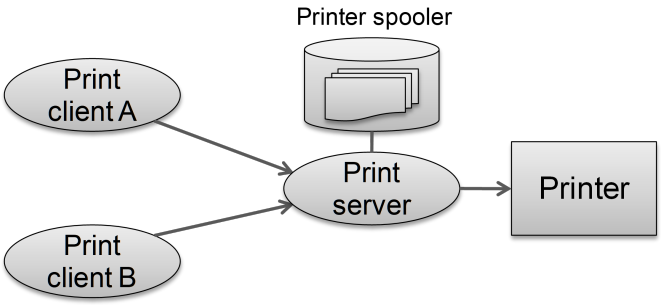
An example of a situation where spooling is not applied

- Both processes A and B want to get access to a printer
- One of the 2 processes has to wait



With spooling technique

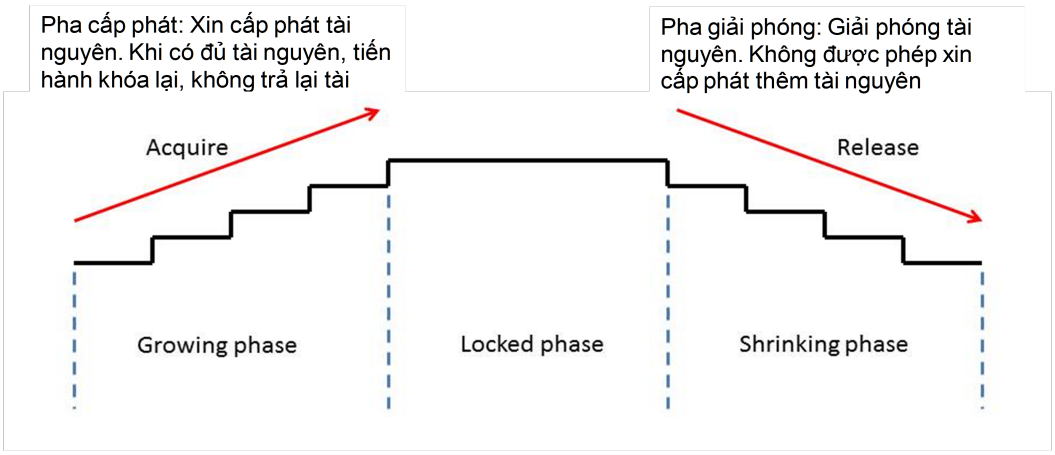
- Print server (daemon) is the only process that owns access permission to the printer
- Other processes send requests to the printer server. Requests are kept in a queue
- Printer server handles the requests from the queue using FIFO principle



Remarks: This technique is not applicable to all types of resources. Resources that requests can not be sequentialized, such as the process information table, are not applicable.

Breaking the Hold and Wait condition

Two-phase locking strategy



This strategy separates the allocation phase from the release phase.

In the 1st phase, a process sends requests for resource allocation to OS. When enough resources have been gathered, OS will lock the resources and allocate them to the process.

In the next phase, the process the resources and gradually release them. In this phase, the process is not allowed to ask for more resources.

The Hold and Wait condition does not hold because

- In the 1st phase, the process only waits without holding resources
- In the 2nd phase, the process only holds but not wait for resources

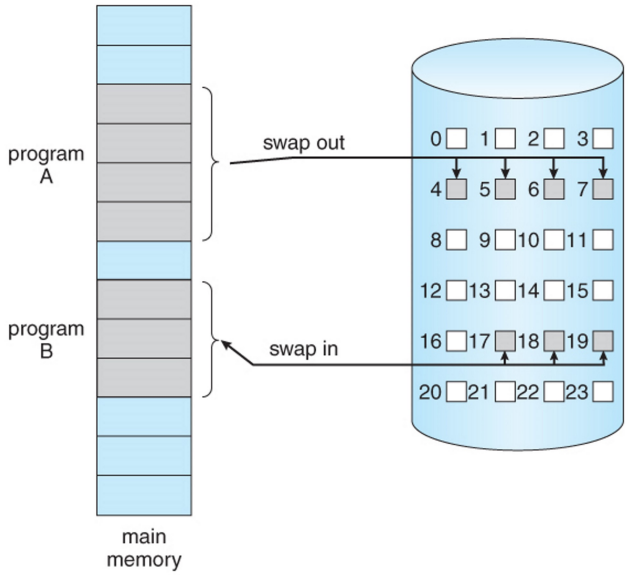
Two-phase locking strategy is often used in databases. When a client sends an SQL query, the database server will lock all necessary colums/tables for the query execution.

Remarks:

- Processes have to declare resources to be used in advance
- Wasting time of holding resources

Breaking No preemption condition

By setting priorities to processes to use the resources. When a deadlock happens, a process with higher priority can "rob" resources from another process with lower priority.



Example: The case when RAM pages are the resource. When a Page fault event occurs, pages of a process with lower priority will be swapped out to external storage and replaced by the content of another process. The priority in this case is determined by using LRU, NFU, WS-Clock algorithm etc.

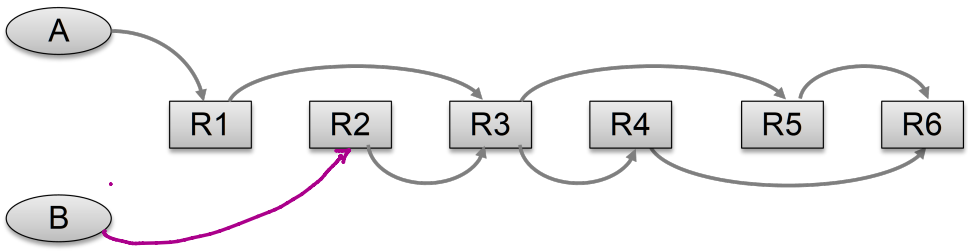
Cons: Not applicable to all types of resources. "Robbing" a resource from a running process may harm the resource data integrity.

Breaking circular condition

Processes asking for resource allocation has to follow a rule to make sure that resource allocation requests do not form a cycle.

A solution is to ennumerate all resources

- All resources have to be enumerated
- A process can only ask for a resource with a higher number



A cycle does not exist as there are no edges from a higher number device to a lower number device.

Remarks:

- Not practical for the situation where there are too many resources and the number of resources are variable, for instance records/rows in a database keep changing all the time and enumeration them all is not practical.
- Processes have to ask for resources of increasing order. This is not practical for interactive application.