

CHAPTER VI

DEADLOCKS

6.1 INTRODUCTION

- A **deadlock** is said to occur whenever several processes are blocked and each of these processes is waiting for a resource that can only be made available by another blocked process.

Think about deadlocks in human affairs: two friends have exchanged a few strong words and each is expecting the other to apologize first, a rebel group does not want to cease the hostilities before being recognized by the government while the government is ready to negotiate but only after the hostilities have ceased

The resources we will consider can be:

- a) **serially reusable resources** such as memory space, buffer space, disk space, tape drives and so forth; since these resources exist only in a *limited quantity*, one process may have to *wait* for another process to release the resources it is currently holding.

Think about what happens in the dining philosopher problem when all five philosophers grab their left forks at the same time: forks are serially reusable.

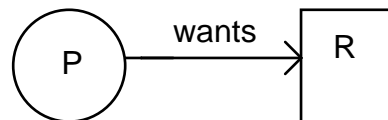
- b) **consumable resources** such as messages and signals; a process *waiting* for a message is waiting for another process to *send* it: we will say that the sending process is *creating* the resource and the receiving process *consuming* it.

Think about two processes waiting for messages from each other: since they are both blocked waiting for a message, neither of them can break the deadlock by sending a message.

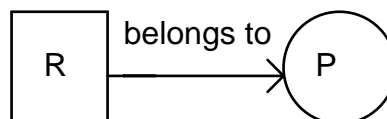
Deadlocks

- There are four possible ways to handle deadlocks:
 1. **do nothing**: many operating systems do nothing to prevent or detect deadlocks: they just ignore the problem;
 2. **deadlock prevention**: it is possible to build systems that are inherently *deadlock-free* but at a heavy cost (e.g., no client-server interactions);
 3. **deadlock avoidance**: it is often possible to detect ahead of time system *unsafe states* that *could* lead to an otherwise avoidable deadlock; hence, we should prevent the system from entering any of these unsafe states; the method only works for some kinds of serially reusable resources;
 4. **deadlock detection**: the system does not try to prevent deadlocks but tries to detect them and break them;
- Situations that may lead to deadlocks are often represented by a **resource-allocation graph** whose nodes can be either *processes* or *resources*.

An edge from a process node **P** to a resource node **R** means that the *process wants to acquire* the resource:



An edge from a resource node **R** to a process node **P** means that the *resource is currently owned* by the process:



6.2 NECESSARY CONDITIONS FOR A DEADLOCK

- There are **four necessary conditions** that must be **simultaneously in effect** for a deadlock to happen:

1. ***Mutual Exclusion:***

at least one of the processes involved in the deadlock must claim **exclusive control** of some of the resources it requires.

2. ***Hold and Wait Condition:***

processes can hold the resources that have already been allocated to them while waiting for additional resources.

3. ***No Preemption:***

once a resource has been allocated to a process, it cannot be taken away or borrowed from that process until the process is finished with it.

4. ***Circular Wait:***

there must be a circular chain of processes such that each process in the chain holds some resources that are needed by the next process in the chain.

*This last condition is the formal equivalent to what we call a **vicious circle** in everyday speech (as in I need to move to a cheaper place to save money but don't have the money to move)*

6.3 DEADLOCK PREVENTION

- Since these four conditions must be simultaneously in effect for a deadlock to happen, we can **prevent** the occurrence of deadlocks by **denying any one** of them:

Deadlocks

1. *Denying Mutual Exclusion:*

the major problem is that many resources can only be used by one process at a time; if a given process writes to a file, no other process should be allowed to access the file while it is being updated.

2. *Denying the Wait for Condition:*

this would require all processes to obtain *all* the resources they may *ever* need *before* using any one of them. There are two problems with this approach:

- a) Resources will be wasted as processes will have to acquire them before they actually need them.
- b) The method does not apply to consumable resources such as messages because it would require any process likely to receive messages from other processes to have received all of them before starting to process any one of them,

3. *Denying the No Preemption Condition:*

letting processes steal—or borrow—resources from another process can be an option but we may have to *abort* the process from which the resources have been taken away, which will result in **lost work** but could also result in the loss of some data (we can abort a compile job and restart it but we don't want to lose a whole text editor session)

4. *Preventing Circular Waits:*

we can impose a total order on all resource types and force all processes to follow that order when they acquire new resources; if a process needs more than one unit of a given resource type it should acquire all of them or none.

The method works very well for resources like CPU and memory: the scheduler prevents deadlocks by only allocating the CPU to processes that have the memory they need and are not waiting on any event.

It does not apply as well to consumable resources as it would force messages to move in only one direction and would therefore prevent processes from exchanging messages.

- The dream of building *large deadlock-free systems* is just a dream.

6.4 DEADLOCK AVOIDANCE

- Deadlock avoidance works by monitoring the way resources are allocated and does not let the system enter any state where it would not be able to satisfy the maximum resource requirements of each process in turn.
- One example of this approach is Dijkstra's **Banker's Algorithm**

6.5 DEADLOCK DETECTION

- The simplest method to detect deadlocks is the use of *timeouts*: a deadlock is likely to have occurred if several processes have failed to make any progress during a reasonable period of time.
- A more general method consists of constructing the resource allocation graph and searching for *cycles*: any cycle would indicate a deadlock.

6.8 KEY CONCEPTS

Be sure that you understand and can explain the following concepts:

all-or-nothing allocation
circular wait condition
consumable resource
deadlock prevention
hold and wait condition
serially reusable resource