

## Lectures on distributed systems

# Distributed Deadlock

Paul Krzyzanowski

## Introduction

A deadlock is a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, four conditions have to be met for a deadlock to occur in a system:

1. **mutual exclusion**  
A resource can be held by at most one process.
2. **hold and wait**  
Processes that already hold resources can wait for another resource.
3. **non-preemption**  
A resource, once granted, cannot be taken away.
4. **circular wait**  
Two or more processes are waiting for resources held by one of the other processes.

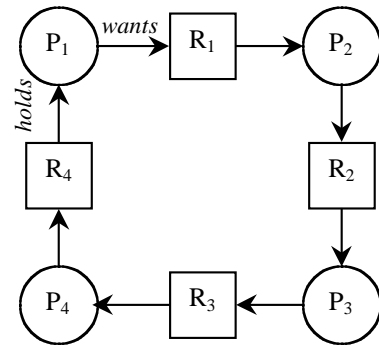


Figure 1. Deadlock

Resource allocation can be represented by directed graphs:

$P_1 \rightarrow R_1$  means that resource  $R_1$  is allocated to process  $P_1$ .

$P_1 \leftarrow R_1$  means that resource  $R_1$  is requested by process  $P_1$ .

Deadlock is present when the graph has cycles. An example is shown in Figure 1.

## Deadlocks in distributed systems

The same conditions for deadlocks in uniprocessors apply to distributed systems. Unfortunately, as in many other aspects of distributed systems, they are harder to detect, avoid, and prevent. Tannenbaum proposes four strategies for dealing with distributed deadlocks:

1. ignorance: ignore the problem (this is the most common approach).
2. detection: let deadlocks occur, detect them, and then deal with them.
3. prevention: make deadlocks impossible.
4. avoidance: choose resource allocation carefully so that deadlocks will not occur.

The last of these, deadlock avoidance through resource allocation is difficult and requires the ability to predict precisely the resources that will be needed and the times that they will be needed. This is difficult and not practical in real systems. The first of these is trivially simple. We will focus on the middle two approaches.

In a conventional system, the operating system is the component that is responsible for resource allocation and is the one to detect deadlocks. Deadlocks are resolved by killing a process. This, of

course, could create unhappiness for the owner of the process. In distributed systems employing a transaction model things are a bit different. Transactions are designed to withstand being aborted and, as such, it is perfectly reasonable to abort one or more transactions to break a deadlock. Hopefully, the transaction can be restarted later without creating another deadlock.

### Centralized deadlock detection

Centralized deadlock detection attempts to imitate the nondistributed algorithm through a central coordinator. Each machine is responsible for maintaining a resource graph for its processes and resources. A *central coordinator* maintains the resource utilization graph for the entire system. This graph is the union of the individual graphs. If this coordinator detects a cycle, it kills off one process to break the deadlock.

In the non-distributed case, all the information on resource usage lives on one system and the graph may be constructed on that system. In the distributed case, the individual subgraphs have to be propagated to a central coordinator. A message can be sent each time an arc is added or deleted. If optimization is needed, a list of added or deleted arcs can be sent periodically to reduce the overall number of messages sent.

Here is an example (from Tanenbaum).

Suppose machine *A* has a process  $P_0$ , which holds the resource  $S$  and wants resource  $R$ , which is held by  $P_1$ . The local graph on *A* is shown in Figure 2. Another machine, machine *B*, has a process  $P_2$ , which is holding resource  $T$  and wants resource  $S$ . Its local graph is shown in Figure 3. Both of these machines send their graphs to the central coordinator, which maintains the union (Figure 4).

All is well. There are no cycles and hence no deadlock. Now two events occur. Process  $P_1$  releases resource  $R$  and asks machine *B* for resource  $T$ . Two messages are sent to the coordinator:

- message 1 (from machine *A*): “releasing  $R$ ”
- message 2 (from machine *B*): “waiting for  $T$ ”

This should cause no problems (no deadlock). However, if message 2 arrives first, the coordinator would then construct the graph in Figure 5 and detect a deadlock. Such a condition is known as **false deadlock**. A way to fix this is to use Lamport’s algorithm to impose global time ordering on all machines. Alternatively, if the coordinator suspects deadlock, it can send a reliable message to every machine asking whether it has any

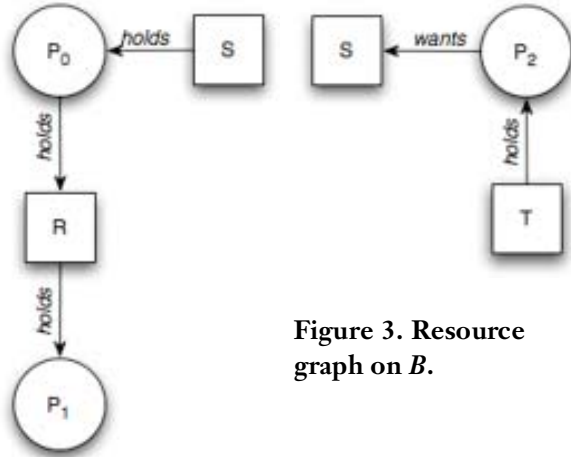


Figure 2. Resource graph on *A*.

Figure 3. Resource graph on *B*.

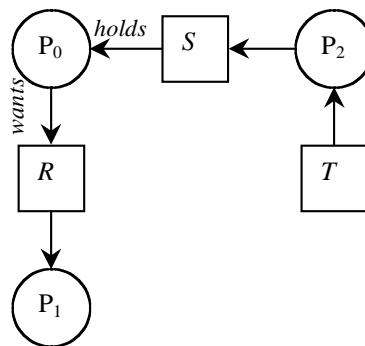
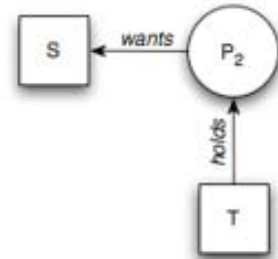


Figure 4. Resource graph on coordinator.

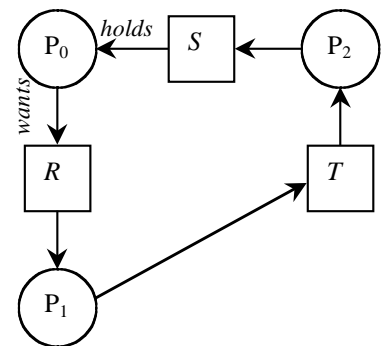


Figure 5. False deadlock coordinator.

release messages. Each machine will then respond with either a release message or a negative acknowledgement to acknowledge receipt of the message.

### Distributed deadlock detection

An algorithm for detecting deadlocks in a distributed system was proposed by Chaudy, Misra, and Haas in 1983. It allows that processes to request multiple resources at once (this speeds up the growing phase). Some processes may wait for resources (either local or remote). Cross-machine arcs make looking for cycles (detecting deadlock) hard.

The algorithm works this way: When a process has to wait for a resource, a **probe** message is sent to the process holding the resource. The probe message contains three components: the process that blocked, the process that is sending the request, and the destination. Initially, the first two components will be the same. When a process receives the probe: if the process itself is waiting on a resource, it updates the *sending* and *destination* fields of the message and forwards it to the resource holder. If it is waiting on multiple resources, a message is sent to each process holding the resources. This process continues as long as processes are waiting for resources. If the originator gets a message and sees its own process number in the *blocked* field of the message, it knows that a cycle has been taken and deadlock exists. In this case, some process (transaction) will have to die. The sender may choose to commit suicide or a ring election algorithm may be used to determine an alternate victim (e.g., youngest process, oldest process, ...).

### Distributed deadlock prevention

An alternative to detecting deadlocks is to design a system so that deadlock is impossible. One way of accomplishing this is to obtain a global timestamp for every transaction (so that no two transactions get the same timestamp). When one process is about to block waiting for a resource that another process is using, check which of the two processes has a younger timestamp and give priority to the older process.

If a younger process is using the resource, then the older process (that wants the resource) waits. If an older process is holding the resource, the younger process (that wants the resource) kills itself. This forces the resource utilization graph to be directed from older to younger processes, making cycles impossible. This algorithm is known as the **wait-die algorithm**.

An alternative method by which resource request cycles may be avoided is to have an old process preempt (kill) the younger process that holds a resource. If a younger process wants a resource that an older one is using, then it waits until the old process is done. In this case, the graph flows from young to old and cycles are again impossible. This variant is called the **wound-wait algorithm**