OPERATING SYSTEMS

# DEADLOCKS

## 7 Deadlocks

In a multiprogramming system, when a process requests resources if those resources are being used by other processes then the process enters a waiting state. However, if other processes are also in a waiting state, we have deadlock. As a formal definition, a set of processes is in a deadlock state if every process in the set is waiting for an event that can only be caused by some processes in the same set.

## 7 Deadlocks

Suppose both two processes want to scan a document and record it on a CD. Process A first requests permission to use the scanner and process B requests the CD recorder first. They are granted. Now A asks for the CD recorder, but the request is denied until B releases it. Unfortunately, instead of releasing the CD recorder B asks for the scanner. At this point both processes are blocked and will remain forever. This situation is called a deadlock.

## 7 Deadlocks

As well as on hardware resources, deadlocks can occur on software resources. For example, in a database system, a program may have to lock several records it is using, to avoid race conditions. If process A locks record R1 and process B locks record R2, and then each process tries to lock the other one's record, we also have a deadlock.

## 7.1 Resources

A resource can be a hardware device or a piece of information. For some resources, several identical instances may be available, such as three disk drives. When interchangeable copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can be used by only a single process at any instant of time.

## 7.1 Resources

Resources come in two types: preemptable and non-preemptable.

- A preemptable resource, like memory, is one that can be taken away from the process owning it with no ill effects.
- A non-preemptable resource, like CD recorders, is one that cannot be taken away from its current owner without causing the computation to fail.

## 7.1 Resources

7

Consider a system with 64 MB of user memory, one CD recorder, and two 64-MB processes that each want to record on CD. Process A starts to compute the values to burn. Before it has finished with the computation, it exceeds its time quantum and is swapped. But if process A has begun to burn, suddenly giving the CD recorder to process B will result in a corrupted CD.

## 7.1 Resources

8

In general, deadlocks involve non-preemptable resources. Potential deadlocks that involve preemptable resources can usually be solved by reallocating resources from one process to another. Thus our treatment will focus on non-preemptable resources.

## 7.1 Resources

9

The sequence of events required to use a resource is as requesting, using, and releasing the resource. If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

## 7.2 Principles of Deadlock

10

Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

## 7.2 Principles of Deadlock

11

In most cases, each member of the set of deadlocked processes is waiting for a resource that is owned by another deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant.

## 7.3 Conditions for Deadlock

12

Following four conditions must be present for a deadlock to occur:

1. Mutual exclusion condition
2. Hold and wait condition
3. No preemption condition
4. Circular wait condition

## 7.3 Conditions for Deadlock

**Mutual exclusion**: Each resource is either currently assigned to exactly one process or is available.

**Hold and wait**: Processes currently holding resources that were granted earlier can request new resources.

**No preemption**: Resources previously granted cannot be forcibly taken away from a process. They must be clearly released by the process holding them.

**Circular wait**: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

## 7.3 Conditions for Deadlock

If one of these conditions is absent, no deadlock is possible. In fact, each condition relates to a policy that a system can have or not have:

- Can a given resource be assigned to more than one process at once?
- Can a process hold a resource and ask for another?
- Can resources be preempted?
- Can circular waits exist?

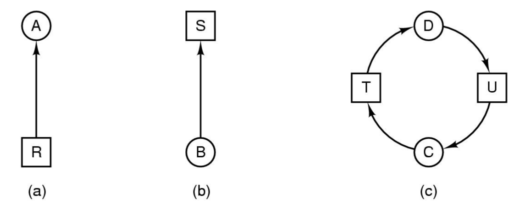
## 7.4 Deadlock Modeling

To modeling of these four conditions directed graphs can be used. The graphs have two kinds of nodes:

- Processes (circles)
- Resources (squares)

An arc from a resource node to a process node means that the resource has previously been requested by, granted to, and is currently held by that process. An arc from a process to a resource means that the process is currently blocked waiting for that resource.

## 7.4 Deadlock Modeling



(a) Holding a resource (b) Requesting a resource. (c) Deadlock.

## 7.4 Deadlock Modeling

In (a) resource R is currently assigned to process A. In (b) , process B is waiting for resource S. In (c) we see a deadlock: process C is waiting for resource T, which is currently held by process D. Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle.

## 7.4 Deadlock Modeling

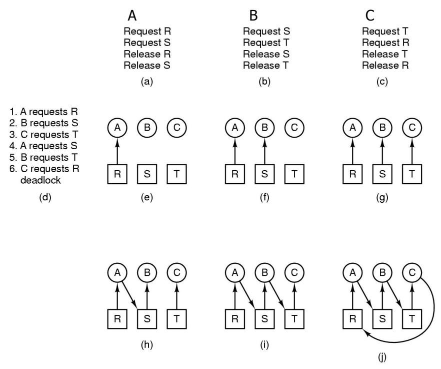
Now let us see how resource graphs can be used. Imagine that we have three processes, A, B, and C, and three resources, R, S, and T. The requests and releases of the three processes are given in (a-c). The operating system is free to run any unblocked process at any instant, so it could decide to run A until A finished all its work, then run B to completion, and finally run C.

## 7.4 Deadlock Modeling

This ordering does not lead to any deadlocks because there is no competition for resources but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin.

## 7.4 Deadlock Modeling

**An example**



## 7.4 Deadlock Modeling

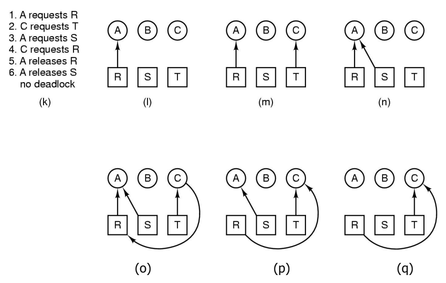
Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of (d). If these six requests are carried out in that order, the six resulting resource graphs are shown in (e)-(j). After request 4 has been made, A blocks waiting for S, as shown in (h). In the next two steps B and C also block, ultimately leading to a cycle and the deadlock of (j). From this point on, the system is frozen.

## 7.4 Deadlock Modeling

The operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request until it is safe. If the operating system knew about the impending deadlock, it could suspend B instead of granting it S. By running only A and C, we would get the requests and releases of (k) instead of (d). This sequence leads to the resource graphs of (l-q), which do not lead to deadlock.

## 7.4 Deadlock Modeling

**An example**



## 7.4 Deadlock Modeling

After step (q), process B can be granted S because A is finished and C has everything it needs. Even if B should eventually block when requesting T, no deadlock can occur. B will just wait until C is finished.

We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock.

## 7.4 Deadlock Modeling

25

Four strategies are used for dealing with deadlocks:

- Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.
- Detection and recovery. Let deadlocks occur, detect them, and take action.
- Dynamic avoidance by careful resource allocation.
- Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

## 7.5 Ignoring: The Ostrich Method

26

The simplest approach is the ostrich method: stick your head in the sand and pretend there is no problem at all.

## 7.5 Ignoring: The Ostrich Method

27

About this method, we must ask some questions:

- how often the problem is expected
- how often the system crashes for other reasons
- how serious a deadlock is

If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance.

## 7.5 Ignoring: The Ostrich Method

28

Windows and Unix systems use this method. These OSs potentially suffer from deadlocks that are not even detected, let alone automatically broken. The total number of processes in a system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a 'fork' fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

## 7.6 Detection and Recovery

29

A second technique is detection and recovery. When this technique is used, the system does not do anything except monitor the requests and releases of resources. Every time a resource is requested or released, the resource graph is updated, and a check is made to see if any cycles exist. If a cycle exists, one of the processes in the cycle is killed. If this does not break the deadlock, another process is killed, and so on until the cycle is broken.

## 7.6 Detection and Recovery

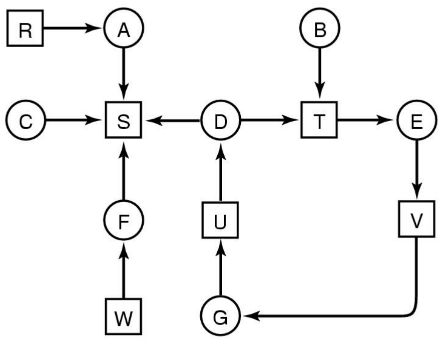
30

Consider a system with seven processes, A though G, and six resources, R through W. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process A holds R and wants S
2. Process B holds nothing but wants T
3. Process C holds nothing but wants S
4. Process D holds U and wants Sand T
5. Process E holds T and wants V
6. Process F holds Wand wants S
7. Process G holds V and wants U

## 7.6 Detection and Recovery

31


$$\begin{tikzcd}
\boxed{R} \arrow[r] & A \arrow[d] & & B \arrow[d] & \\
C \arrow[r] & \boxed{S} & D \arrow[l] \arrow[r] & \boxed{T} \arrow[r] & E \arrow[d] \\
& F \arrow[u] & \boxed{U} \arrow[u] & & \boxed{V} \arrow[dll] \\
& \boxed{W} \arrow[u] & G \arrow[u] & &
\end{tikzcd}$$

## 7.6 Detection and Recovery

32

Although it is relatively simple to pick out the deadlocked processes by eye from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. We will give a simple algorithm that inspects a graph and terminates either when it has found a cycle or when it has shown that none exists. It uses one dynamic data structure, L, a list of nodes, as well as the list of arcs. During the algorithm, arcs will be marked to indicate that they have already been inspected, to prevent repeated inspections.

## 7.6 Detection and Recovery

33

1. For each node, N in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, designate all arcs as unmarked.
3. Add current node to end of L, if the node now appears in L two times. If it does, there is a cycle, algorithm terminates.
4. From given node, see if any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this is initial node, there is no cycle, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.

## 7.6 Detection and Recovery

34

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again.

- Recovery through Preemption
- Recovery through Rollback
- Recovery through Killing Processes

## 7.6 Detection and Recovery

35

**Recovery through preemption**
- Take a resource from some other process
- Depends on nature of the resource

**Recovery through rollback**
- Checkpoint a process periodically
- Use this saved state
- Restart the process if it is found deadlocked

**Recovery through killing processes**
- Kill one of the processes in the deadlock cycle
- The other processes get its resources
- Choose process that can be rerun from the beginning

## 7.7 Deadlock Prevention

36

To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded.

| Condition | Approach |
|---|---|
| Mutual exclusion | Spool everything |
| Hold and wait | Request all resource initially |
| No preemption | Take resources away |
| Circular wait | Order resources numerically |

## 7.7 Deadlock Prevention

37

**Mutual Exclusion**

- Some resources are not sharable, but can be made sharable (printer, tape, etc.)
- Some resources can be made virtual
  - Spooling - Printer
    - Does spooling apply to all non-sharable resources?
  - Mixing - Soundcard

## 7.7 Deadlock Prevention

38

**Hold and Wait**

- Require processes to request resources before starting
  - A process never has to wait for what it needs
  - Telephone companies do this
- Solution
  - Process must give up all resources
  - Then request all immediately needed

## 7.7 Deadlock Prevention

39

**No Preemption**

- This is not a reasonable option
- Consider a process given the printer
  - Halfway through its job
  - No  forcibly take away printer

## 7.7 Deadlock Prevention

40

**Circular Wait**

- Impose an order of requests for all resources
- Method
  - Assign a unique ID to each resource
  - All resource requests must be in an ascending order of the IDs
  - Release resources in a descending order
- Prove this method has no circular wait!
- Is this generally feasible?

## 7.8 Avoidance: The Banker's algorithm

41

A way to avoid deadlocks is the banker's algorithm proposed by Dijkstra (1965). Consider a banker and some credit customers. The banker does not have enough cash to lend every customer at the same time, but he knows that not all customers will need their maximum credit immediately. He also trusts every customer to be able to repay his loan, so he knows eventually he can service all the requests. Here, customers are processes, units are disk drives, and the banker is the operating system.

## 7.8 Avoidance: The Banker's algorithm

42

Dijkstra’s (1965) the banker's algorithm

- Each customer tells banker the maximum number of resources it needs
- Customer borrows resources from banker
- Customer returns resources to banker
- Customer eventually pays back loan
- **Banker only lends resources if the system will be in a safe state after the loan**

**Safe state** - there is a lending sequence such that all customers can take out a loan

**Unsafe state** - there is a possibility of deadlock

## 7.8 Avoidance: The Banker's algorithm

|  | Has | Max |
|---|---|---|
| A | 0 | 6 |
| B | 0 | 5 |
| C | 0 | 4 |
| D | 0 | 7 |

Free: 10

(a) Safe

|  | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

Free: 2

(b) Safe

|  | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 2 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

Free: 1

(c) Unsafe

## 7.8 Avoidance: The Banker's algorithm

Each part of the figure shows a state of the system with respect to resource allocation, that is, a list of customers showing the money already loaned (disk drives already assigned) and the maximum credit available (maximum number of disk drives needed at once later). A state is safe if there exists a sequence of other states that leads to all customers getting loans up to their credit limits (all processes getting all their resources and terminating).

## 7.8 Avoidance: The Banker's algorithm

The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in (b). This state is safe because with two units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources. With four units in hand, the banker can let either D or B have the necessary units, and so on.

## 7.8 Avoidance: The Banker's algorithm

Consider what would happen if a request from B for one more unit were granted in (b). We would have situation (c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not have to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

## 7.8 Avoidance: The Banker's algorithm

The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

## 7.8 Avoidance: The Banker's algorithm

Consider we have three processes and 10 free disk space. Processes (A, B, and C) need 9, 4, and 7 disk space, respectively. First, how can we serve to processes?

|  | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 3

(a)

|  | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |

Free: 1

(b)

|  | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 2 | 7 |

Free: 5

(c)

|  | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 7 | 7 |

Free: 0

(d)

|  | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 0 | – |

Free: 7

(e)