

# Deadlock

Hui Chen <sup>a</sup>

<sup>a</sup>CUNY Brooklyn College

April 2, 2020

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

# Synchronization Issues

- ▶ Liveness
  - ▶ Deadlock
  - ▶ Starvation
  - ▶ Priority inversion

# Liveness

- ▶ *Liveness* refers to a set of properties that a system must satisfy to ensure processes make progress.
  - ▶ Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
  - ▶ Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
  - ▶ Indefinite waiting is an example of a liveness failure.

# Deadlock

- ▶ Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- ▶ Consider the following example,  
Let S and Q be two semaphores initialized to 1

<i>P0</i>	<i>P1</i>
1 wait(S);	1 wait(Q);
2 wait(Q);	2 wait(S);
3 ...	3 ...
4 signal(S);	4 signal(Q);
5 signal(Q);	5 signal(S);

- ▶ Consider if P0 executes wait(S) and P1 wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q)
- ▶ However, P1 is waiting until P0 execute signal(S).
- ▶ Since these signal() operations will never be executed, P0 and P1 are deadlocked.

# Starvation

- ▶ Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

# Priority Inversion

- ▶ Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - ▶ Consider the scenario with three processes P1, P2, and P3.
  - ▶ P1 has the highest priority, P2 the next highest, and P3 the lowest.
  - ▶ Assume a resource R is assigned a resource R that P1 wants. Thus, P1 must wait for P3 to finish using the resource.
  - ▶ However, P2 becomes runnable and preempts P3.
  - ▶ What has happened is that P2, a process with a lower priority than P1 has indirectly prevented P3 from gaining access to the resource.
- ▶ Solved via priority-inheritance protocol.



## Priority Inheritance Protocol

- ▶ The protocol simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource.
- ▶ Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.

# Outline

- 1 Synchronization Issues
- 2 **Deadlock and Solutions**
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

## Necessary Conditions for Deadlocks

Deadlock can arise if four conditions hold simultaneously. (the 4 necessary conditions for deadlocks)

- ▶ Mutual exclusion. Only one process at a time can use a resource
- ▶ Hold and wait. A process holding at least one resource is waiting to acquire additional resources held by other processes
- ▶ No preemption. A resource can be released only voluntarily by the process holding it, after that process has completed its task
- ▶ Circular wait. There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ ,  $\dots$ ,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# Handling Deadlocks

- ▶ Ensure that the system will never enter a deadlock state.
  - ▶ Deadlock prevention (by structurally negating one of the four required conditions)
  - ▶ Deadlock avoidance (by carefully allocating resources)
- ▶ Allow the system to enter a deadlock state and then recover
  - ▶ Deadlock detection and recovery (Let deadlocks occur, detect them, and then take action)
- ▶ Ignore the problem and pretend that deadlocks never occur in the system.
  - ▶ The Ostrich algorithm

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm**
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

Deadlocks in my system  
happen once in a blue moon and  
...



Figure: The Ostrich Algorithm

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention**
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

# Deadlock Prevention

By invalidating one of the four required conditions

- ▶ Mutual Exclusion
- ▶ Hold and wait
- ▶ No preemption
- ▶ Circular wait

But is it possible, and if possible how and at what cost?



## Invalidating Mutual Exclusion?

We introduce terms, “shareable resources” and “non-shareable resources”

- ▶ Shareable resources. Resources that allow simultaneous access, e.g., a read-only file. There isn't a mutual exclusion requirement to shareable resources.
- ▶ Non-shareable resources. Resources that do not allow simultaneous access, e.g., a printer or a mutex lock.

Cannot prevent deadlocks by denying the mutual-exclusion condition?

## Invalidating Hold-and-Wait?

That is to say, we must guarantee that whenever a process requests a resource, it does not hold any other resources. How do we achieve this?

1. Require a process to request and be allocated all its resources before it begins execution, or
2. allow a process to request resources only when the process has none allocated to it (e.g., by releasing it)

At what cost?

- ▶ Low resource utilization;
- ▶ starvation possible;
- ▶ also impractical

## Invalidating No-Preemption?

To invalidate no-preemption, we consider that the OS may do the following,

1. Check whether requested resources by process  $P_i$  are allocated to process  $P_j$  that is waiting for additional resources.
2. If so, we preempt the desired resources from  $P_j$  and allocate the resources to  $P_i$ .

Is it possible?

- ▶ Possible for resources whose state can be easily saved and restored later, such as, a database transaction
- ▶ However, not generally possible, e.g., mutex locks and semaphores.

## Invalidating Circular Wait?

Consider the following approach.

1. Impose a total ordering of all resource types by assigning each resource (i.e., mutex locks) a unique number.
2. Resources must be acquired in order based on the numbers

Does it invalidating circular wait? (Circular wait cease to happen)

- ▶ Yes, we can prove it by contradiction.

However,

- ▶ Resource ordering does not in itself prevent deadlock. Application developers must write programs that follow the ordering.
- ▶ However, establishing an ordering of all resources can be sometimes difficult.
  - ▶ Considering on a system with hundreds or even thousands of locks <sup>1</sup>.
  - ▶ What if locks can be acquired dynamically?

---

<sup>1</sup>To address this challenge, many Java developers have adopted the strategy of using the method `System.identityHashCode()` as the function for ordering lock acquisition

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph**
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

# Resource Allocation Graph

Use it to determine whether there is a circular wait condition.

- ▶ A set of vertices  $\mathbb{V}$  and a set of edges  $\mathbb{E}$ .
- ▶  $\mathbb{V}$  is partitioned into two types:
  - ▶  $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system (*drawn in ovals*)
  - ▶  $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system (*drawn in rectangles*)
- ▶  $\mathbb{E}$  is partitioned into two types:
  - ▶ Request edge. directed edge  $P_i \rightarrow R_j$ , which reads " $P_i$  requests or waits for  $R_j$ "
  - ▶ Assignment edge. directed edge  $R_j \rightarrow P_i$ , which reads " $R_j$  is assigned to or is held by  $P_i$ "

# Examples of Resource Allocation Algorithms

Let's examine a few examples ...

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm**
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads



# Banker's Algorithm

Use it to determine whether there is a circular wait condition when a resource has multiple instances.

# Data Structures

Let  $n$  = number of processes, and  $m$  = number of resources types.

- ▶ Available (or Free): Vector of length  $m$ . If  $available[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
  - ▶ Max:  $n \times m$  matrix. If  $Max[i, j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
  - ▶ Allocation (or Has):  $n \times m$  matrix. If  $Allocation[i, j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
  - ▶ Need:  $n \times m$  matrix. If  $Need[i, j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task
- $Need [i, j] = Max[i, j] - Allocation [i, j]$

# Safety Algorithm

1. Let `Work` and `Finish` be vectors of length  $m$  and  $n$ , respectively. Do the following initialization,
  - `Work = Available`
  - For  $i = 0, 1, \dots, n-1$ :
    - `Finish[i] = false`
2. Find an index  $i$  such that both
  - 2.1 `Finish[i] == false`
  - 2.2 `Need[i] ≤ Work`
 If no such  $i$  exists, go to step 4.
3. `Work = Work + Allocation[i]`  
`Finish[i] = true`  
 Go to step 2.
4. If `Finish[i] == true` for all  $i$ , then the system is in a *safe state*; otherwise, *unsafe state*.

This algorithm may require an order  $O(m \times n^2)$  operations to determine whether a state is safe.

# Examples of Running Safety Algorithm

Let's examine a few examples ...

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance**
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

# Deadlock Avoidance

Use Resource Allocation Graph or a variant of Banker's algorithm to determine if current resource allocation is in a safe state.

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery**
- 9 Events vs. Threads

# Deadlock Detection and Recovery

1. Use Resource Allocation Graph (Wait-for Graph) or a variant of Banker's algorithm to determine if there is a deadlock.
2. Recovery from the deadlock (multiple approaches)



# Matrix-based Deadlock Detection Algorithm

Using a variant of Banker's algorithm to detect whether there is a deadlock.

## Data Structures

Let  $n$  = number of processes, and  $m$  = number of resources types.

- ▶ Available. A vector of length  $m$  indicates the number of available resources of each type.
- ▶ Allocation. An  $n \times m$  matrix defines the number of resources of each type currently allocated to each thread.
- ▶ Request. An  $n \times m$  matrix indicates the current request of each thread. If  $\text{Request}[i][j]$  equals  $k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ . (Compare this with Need in the safety algorithm)

# Deadlock Detection Algorithm

1. Let `Work` and `Finish` be vectors of length  $m$  and  $n$ , respectively. Do the following initialization,
  - `Work = Available`
  - For  $i = 0, 1, \dots, n-1$ :
    - if `Allocation[i]  $\neq$  0`, then `Finish[i] = false`
    - else `Finish[i] = true`
2. Find an index  $i$  such that both
  - 2.1 `Finish[i] == false`
  - 2.2 `Request[i]  $\leq$  Work`
 If no such  $i$  exists, go to step 4.
3. `Work = Work + Allocation[i]`  
`Finish[i] = true`  
 Go to step 2.
4. If `Finish[i] == false` for some  $i$ ,  $0 \leq i < n$ , then the system is in a deadlocked state. Moreover, if `Finish[i] == false`, then process  $P_i$  is deadlocked.

# Examples of Deadlock Detection Algorithm

Let's examine a few examples ...

# Outline

- 1 Synchronization Issues
- 2 Deadlock and Solutions
  - Necessary Conditions
- 3 The Ostrich Algorithm
- 4 Deadlock Prevention
- 5 Resource Allocation Graph
- 6 Banker's Algorithm
- 7 Deadlock Avoidance
- 8 Deadlock Detection and Recovery
- 9 Events vs. Threads

# Events vs. Threads

There have been a recurrent discussion on how we should realize concurrency [2, 3, 4, 6, 5]

- ▶ Threads vs. events [2, 4, 6, 5]
- ▶ Theory vs. practice ([1, Section 9.1], [3])

## Reference I

- [1] Brian Goetz et al. *Java concurrency in practice*. Pearson Education, 2006.
- [2] Hugh C Lauer and Roger M Needham. “On the duality of operating system structures”. In: *ACM SIGOPS Operating Systems Review* 13.2 (1979), pp. 3–19.
- [3] John Ousterhout. “Why threads are a bad idea (for most purposes)”. In: *Presentation given at the 1996 Usenix Annual Technical Conference*. Vol. 5. San Diego, CA, USA. 1996.
- [4] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. “Why Events Are a Bad Idea (for High-Concurrency Servers).”. In: *HotOS*. 2003, pp. 19–24.

## Reference II

- [5] Rob Von Behren et al. “Capriccio: scalable threads for internet services”. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 268–281.
- [6] Matt Welsh, David Culler, and Eric Brewer. “SEDA: an architecture for well-conditioned, scalable internet services”. In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 230–243.