

## 1 Overview

Multithreading allows programmers to improve application performance by executing semi-independent chunks of code concurrently. Of course, sometimes threads have to synchronize on shared resources. Figuring out how to protect shared data with appropriate synchronization is hard enough, but it turns out that there's another lurking problem with synchronization: DEADLOCK!

## 2 Deadlock Basics

Deadlock, also known as the deadly embrace, is a condition where threads cannot make progress because they are waiting for a resource held by another thread that is itself waiting for a resource. When threads deadlock, they mutually block each other and prevent any thread from making progress. In practice, deadlock is when your multi-threaded program hangs.

### 2.1 An Example

Lets say you have two threads: 1 and 2. These threads share two queues, each protected by a mutex. One is protected by mutex A and the other by mutex B. Both threads are attempting to copy elements from one queue to the other. Consider the following sequence of operations:

Thread1:copy(queue A, queue B)	Thread2:copy(queue B, queue A)
lock mutex A	...
lock mutex B	...
copy from A to B	...
unlock mutex B	lock mutex B
unlock mutex A	lock mutex A
...	copy from B to A
...	unlock mutex A
...	unlock mutex B

No problem, right. Thread 1 got to mutex A and B first, so it got to copy things over first. Both threads accomplished their task, thread 1 just got to go first. Now consider a slightly different interleaving:

Thread1:copy(queue A, queue B)	Thread2:copy(queue B, queue A)
lock mutex A	lock mutex B
lock mutex B	lock mutex A
wait on B	wait on A
...	...

Now, just because the threads executed at a slightly different rate, thread 1 blocks thread 2 and thread 2 blocks thread 1. Neither thread can make progress because each is waiting on a resource held by the other. And the other thread is incapable of releasing this resource because it is blocked on a lock request! This is deadlock. Your threaded code looked fine, and you used locks to protect shared resources, but now your code can hang!

Deadlock is a serious concern. It is a common problem in multithreaded code. Consider I/O resources exported by an operating system. Access to these resources is usually mutually exclusive (e.g. only one process can access the printer at a time). Whenever a process attempts to claim multiple resources in sequence, it runs the risk of deadlocking with another process contending for those same resources.

### 3 Deadlock, formally

**Definition:** A set of processes is deadlocked if each process in the set is blocked waiting for an event that only another process in the set can cause.

Deadlock has been heavily studied, and there are 4 necessary conditions a multi-process system must exhibit for deadlock to occur:

1. Mutually exclusive access to resources
2. Resources are non-preemptive (they can only be released by the process holding them)
3. Partial allocation is allowed (resources can be acquired on-demand and one at a time)
4. Circular waiting is allowed (threads can grab resources in any order they please)

As you can see, multithreaded programs using semaphores, locks, and message queues all fulfill these conditions. This is unfortunate, because these (particularly locks) are the major options for multi-threaded programming.

#### 3.1 Resource Allocation Diagrams

A helpful tool for detecting and illustrating deadlock is the resource allocation diagram. In these diagrams, resources/mutexes are squares and processes/threads are circles. A directed edge is drawn from a resource to a process if that process holds that resource, and a directed edge is drawn from a process to a resource if that process is requesting that resource. The following diagram represents the resource allocation diagram from the deadlock example above:

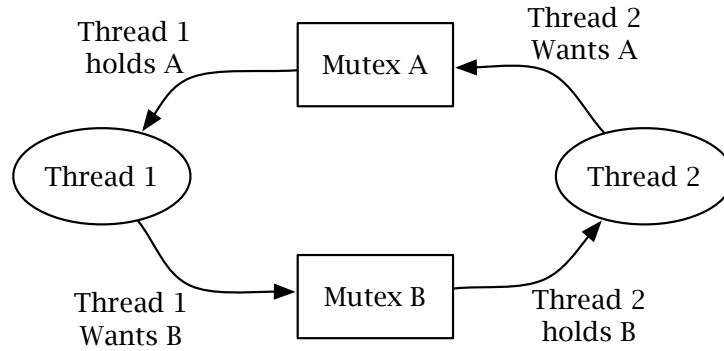


Figure 1: 2 Deadlocked Threads

As you can see, if deadlock occurs, there's a cycle in the resource allocation diagram. Of course, this is a very simple case. Resource allocation diagrams can get really hairy for applications with dozens of threads and hundreds of resources/mutexes. For example, consider this diagram:

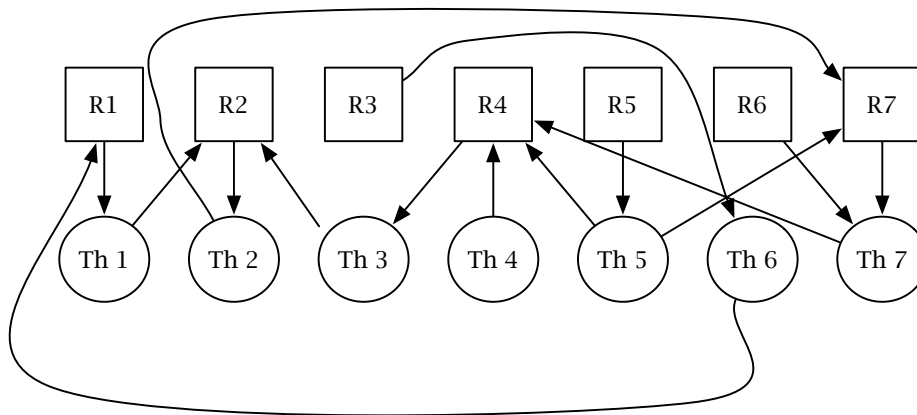


Figure 2: Find the cycle, get a cookie!

## 4 Solutions

Deadlock is a problem, so why not solve it? In the real world, the most common solution is to **Do Nothing**. That's right, most multithreaded software has incorporated no anti-deadlock mechanisms. Programmers are expected to just write completely correct software that never deadlocks. As you can imagine, this is rarely realized in practice. Among the other solutions are: detecting

deadlock, preventing deadlock, and building systems that avoid deadlock but can cope when it occurs.

## 4.1 Detection and Recovery

Deadlock detection essentially amounts to building the resource allocation graph at runtime and periodically doing cycle detection. When a cycle is detected, then the system has to employ a recovery strategy to eliminate the deadlock. Usually this means 'breaking the cycle', by which I mean that the runtime system kills one of the deadlocked threads and restarts it. This will release the locked resources and get the deadlocked threads going again. As you can imagine, there are some shortcomings to this approach:

1. Maintaining a resource graph is expensive in time AND space. Cycle detection is also time-consuming.
2. How exactly do you restart a thread?

The first problem can be "fixed" by approximating deadlock with long blocking times. If a thread has been blocked on a resource for "a while" (which is usually some arbitrary timeout specified by the runtime developers), then you assume that it's deadlocked and kill it.

The second problem is more complicated. Killing a thread isn't that difficult. Most threading systems have support for stopping a thread and de-allocating its resources. But how do you restart the thread? How can the runtime system stop a thread, then undo its changes, and then restart it from some "safe point"? There are actually ways to do this, they just don't map onto standard locking systems easily.

## 4.2 Prevention

The second strategy is to design a system that prevents deadlock from ever occurring. This can be accomplished by preventing any of the 4 necessary conditions. Let's visit those conditions again:

1. *Mutually exclusive access to resources* This is pretty much unavoidable for some resources. Many I/O devices can only process one request at a time. Data structures can often be re-engineered to be "lock free" which avoids this problem. But lock freedom is **highly** non-trivial and usually gets into low-level hardware details pretty fast.
2. *Resources are non-preemptive* If mutexes could be pre-empted, that would mean that threads could force other threads to relinquish resources and thereby break the deadlock cycle. However, preemption complicates mutex design which tends to make mutex lock/unlock operations more expensive. Additionally, the threading system must be able to decide when to pre-empt. Real deadlock detection is expensive, so that means that some approximation will be used. Because you want to catch all deadlocks, that means that the approximation will sometimes incorrectly flag threads as being deadlocked. This means that the threading system may pre-empt and restart threads unnecessarily, which is inefficient. Additionally, pre-emptive resources will require thread restarts which are complicated...

3. *Partial allocation is allowed* Partial allocation is what happens when a thread grabs mutexes one at a time. A threading system could simply require that all threads request all resources at startup. Then the system could schedule threads and allocate resources to prevent deadlock. This sounds great, but unfortunately it has some drawbacks. What if the amount of resources is unknown at startup time (for instance if user-input drives the thread)? Or what if resources are allocated and de-allocated dynamically? (as in fine-grained datastructures with single locks per list nodes or tree nodes). These problems mean that ahead of time allocation is possible only for a small subset of real programs.
4. *Circular waiting is allowed* Circular waiting occurs when threads can grab resources in any order. To prevent circular waiting, a system could rank all resources and require the threads to grab resources in order. This would solve our simple deadlock example above, if  $A < B$ , then both thread 1 and thread 2 would have to grab lock A before lock B. This solution is unfortunately inconvenient as well as inefficient. Different applications may require different lock orders, so a static ranking of resources is bad. Internal to an application it may be possible to use this scheme, however it is generally infeasible to have a compiler do this automatically.

### 4.3 Avoiding Deadlock

Deadlock avoidance is the middle path. Rather than attempting to totally prevent deadlock in the first place, the threading system makes it much less likely. This is accomplished by relaxing the prevention techniques so that they don't impact programmer convenience or efficiency too much. Deadlock may still occur, but it is now much more unlikely. This means that an avoidance system will still occasionally have to detect deadlock, or rely on programmers to write deadlock-free code.

An example of a relaxed prevention technique is **2 Phase Locking** (often called 2PL). This technique is particularly popular in databases and transactional systems. In 2PL a thread/transaction can still grab resources 1 at a time (partial allocation), but resources can only be released when all needed resources are acquired. This is why it's called two-phase locking: the first phase is when the thread grabs all the locks it needs. Then the thread processes for a bit. The second phase is when the thread releases all of its held resources (usually in the reverse order it grabbed them). In practice, the first phase is usually lock-grabbing interleaved with some computation (while the thread figures out which lock to grab next), and the second phase happens all in a big chunk at the end. 2PL does not absolutely prevent deadlock, but it establishes an order to acquiring and releasing resources that avoids many common deadlock scenarios. In DB systems, transactions are run in a 2PL fashion, but because deadlock can still happen the DB performs occasional deadlock checks.