32

Common Concurrency Problems

Researchers have spent a great deal of time and effort looking into concurrency bugs over many years. Much of the early work focused on **deadlock**, a topic which we've touched on in the past chapters but will now dive into deeply [C+71]. More recent work focuses on studying other types of common concurrency bugs (i.e., non-deadlock bugs). In this chapter, we take a brief look at some example concurrency problems found in real code bases, to better understand what problems to look out for. And thus our central issue for this chapter:

CRUX: HOW TO HANDLE COMMON CONCURRENCY BUGS

Concurrency bugs tend to come in a variety of common patterns. Knowing which ones to look out for is the first step to writing more robust, correct concurrent code.

32.1 What Types Of Bugs Exist?

The first, and most obvious, question is this: what types of concurrency bugs manifest in complex, concurrent programs? This question is difficult to answer in general, but fortunately, some others have done the work for us. Specifically, we rely upon a study by Lu et al. [L+08], which analyzes a number of popular concurrent applications in great detail to understand what types of bugs arise in practice.

The study focuses on four major and important open-source applications: MySQL (a popular database management system), Apache (a well-known web server), Mozilla (the famous web browser), and OpenOffice (a free version of the MS Office suite, which some people actually use). In the study, the authors examine concurrency bugs that have been found and fixed in each of these code bases, turning the developers' work into a quantitative bug analysis; understanding these results can help you understand what types of problems actually occur in mature code bases.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**

Figure 32.1 shows a summary of the bugs Lu and colleagues studied. From the figure, you can see that there were 105 total bugs, most of which were not deadlock (74); the remaining 31 were deadlock bugs. Further, you can see that the number of bugs studied from each application; while OpenOffice only had 8 total concurrency bugs, Mozilla had nearly 60.

We now dive into these different classes of bugs (non-deadlock, deadlock) a bit more deeply. For the first class of non-deadlock bugs, we use examples from the study to drive our discussion. For the second class of deadlock bugs, we discuss the long line of work that has been done in either preventing, avoiding, or handling deadlock.

32.2 Non-Deadlock Bugs

Non-deadlock bugs make up a majority of concurrency bugs, according to Lu's study. But what types of bugs are these? How do they arise? How can we fix them? We now discuss the two major types of non-deadlock bugs found by Lu et al.: **atomicity violation** bugs and **order violation** bugs.

Atomicity-Violation Bugs

The first type of problem encountered is referred to as an **atomicity violation**. Here is a simple example, found in MySQL. Before reading the explanation, try figuring out what the bug is. Do it!

```
1  Thread 1::
2  if (thd->proc_info) {
3    ...
4    fputs(thd->proc_info, ...);
5    ...
6  }
7
8  Thread 2::
9  thd->proc_info = NULL;
```

In the example, two different threads access the field `proc_info` in the structure `thd`. The first thread checks if the value is non-NULL and then prints its value; the second thread sets it to NULL. Clearly, if the first thread performs the check but then is interrupted before the call to `fputs`, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by `fputs`.

The more formal definition of an atomicity violation, according to Lu et al, is this: "The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)." In our example above, the code has an *atomicity assumption* (in Lu's words) about the check for non-NULL of `proc_info` and the usage of `proc_info` in the `fputs()` call; when the assumption is incorrect, the code will not work as desired.

Finding a fix for this type of problem is often (but not always) straightforward. Can you think of how to fix the code above?

In this solution, we simply add locks around the shared-variable references, ensuring that when either thread accesses the `proc_info` field, it has a lock held (`proc_info_lock`). Of course, any other code that accesses the structure should also acquire this lock before doing so.

```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6    ...
7    fputs(thd->proc_info, ...);
8    ...
9  }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);
```

Order-Violation Bugs

Another common type of non-deadlock bug found by Lu et al. is known as an **order violation**. Here is another simple example; once again, see if you can figure out why the code below has a bug in it.

```
1  Thread 1::
2  void init() {
3     ...
4     mThread = PR_CreateThread(mMain, ...);
5     ...
6  }
7
8  Thread 2::
9  void mMain(...) {
10     ...
11     mState = mThread->State;
12     ...
13 }
```

As you probably figured out, the code in Thread 2 seems to assume that the variable `mThread` has already been initialized (and is not NULL); however, if Thread 2 runs immediately once created, the value of `mThread` will not be set when it is accessed within `mMain()` in Thread 2, and will

likely crash with a NULL-pointer dereference. Note that we assume the value of `mThread` is initially NULL; if not, even stranger things could happen as arbitrary memory locations are accessed through the dereference in Thread 2.

The more formal definition of an order violation is this: "The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)" [L+08].

The fix to this type of bug is generally to enforce ordering. As we discussed in detail previously, using **condition variables** is an easy and robust way to add this style of synchronization into modern code bases. In the example above, we could thus rewrite the code as follows:

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit             = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // signal that the thread has been created...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20      ...
21      // wait for the thread to be initialized...
22      pthread_mutex_lock(&mtLock);
23      while (mtInit == 0)
24          pthread_cond_wait(&mtCond, &mtLock);
25      pthread_mutex_unlock(&mtLock);
26
27      mState = mThread->State;
28      ...
29 }
```

In this fixed-up code sequence, we have added a lock (`mtLock`) and corresponding condition variable (`mtCond`), as well as a state variable (`mtInit`). When the initialization code runs, it sets the state of `mtInit` to 1 and signals that it has done so. If Thread 2 had run before this point, it will be waiting for this signal and corresponding state change; if it runs later, it will check the state and see that the initialization has already occurred (i.e., `mtInit` is set to 1), and thus continue as is proper. Note that we could likely use `mThread` as the state variable itself, but do not do so for the sake of simplicity here. When ordering matters between threads, condition variables (or semaphores) can come to the rescue.

Non-Deadlock Bugs: Summary

A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations. Thus, by carefully thinking about these types of bug patterns, programmers can likely do a better job of avoiding them. Moreover, as more automated code-checking tools develop, they should likely focus on these two types of bugs as they constitute such a large fraction of non-deadlock bugs found in deployment.

Unfortunately, not all bugs are as easily fixable as the examples we looked at above. Some require a deeper understanding of what the program is doing, or a larger amount of code or data structure reorganization to fix. Read Lu et al.'s excellent (and readable) paper for more details.

32.3 Deadlock Bugs

Beyond the concurrency bugs mentioned above, a classic problem that arises in many concurrent systems with complex locking protocols is known as **deadlock**. Deadlock occurs, for example, when a thread (say Thread 1) is holding a lock (`L1`) and waiting for another one (`L2`); unfortunately, the thread (Thread 2) that holds lock `L2` is waiting for `L1` to be released. Here is a code snippet that demonstrates such a potential deadlock:

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Note that if this code runs, deadlock does not necessarily occur; rather, it may occur, if, for example, Thread 1 grabs lock `L1` and then a context switch occurs to Thread 2. At that point, Thread 2 grabs `L2`, and tries to acquire `L1`. Thus we have a deadlock, as each thread is waiting for the other and neither can run. See Figure 32.2 for a graphical depiction; the presence of a **cycle** in the graph is indicative of the deadlock.

The figure should make clear the problem. How should programmers write code so as to handle deadlock in some way?

CRUX: HOW TO DEAL WITH DEADLOCK

How should we build systems to prevent, avoid, or at least detect and recover from deadlock? Is this a real problem in systems today?

Why Do Deadlocks Occur?

As you may be thinking, simple deadlocks such as the one above seem readily avoidable. For example, if Thread 1 and 2 both made sure to grab locks in the same order, the deadlock would never arise. So why do deadlocks happen?

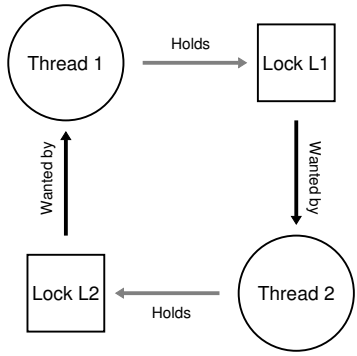


Figure 32.2: **The Deadlock Dependency Graph**

One reason is that in large code bases, complex dependencies arise between components. Take the operating system, for example. The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system. Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code.

Another reason is due to the nature of **encapsulation**. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking. As Jula et al. point out [J+08], some seemingly innocuous interfaces almost invite you to deadlock. For example, take the Java Vector class and the method `AddAll()`. This routine would be called as follows:

```
Vector v1, v2;
v1.AddAll(v2);
```

Internally, because the method needs to be multi-thread safe, locks for both the vector being added to (v1) and the parameter (v2) need to be acquired. The routine acquires said locks in some arbitrary order (say v1 then v2) in order to add the contents of v2 to v1. If some other thread calls `v2.AddAll(v1)` at nearly the same time, we have the potential for deadlock, all in a way that is quite hidden from the calling application.

Conditions for Deadlock

Four conditions need to hold for a deadlock to occur [C+71]:

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur. Thus, we first explore techniques to *prevent* deadlock; each of these strategies seeks to prevent one of the above conditions from arising and thus is one approach to handling the deadlock problem.

Prevention

Circular Wait

Probably the most practical prevention technique (and certainly one that is frequently employed) is to write your locking code such that you never induce a circular wait. The most straightforward way to do that is to provide a **total ordering** on lock acquisition. For example, if there are only two locks in the system (`L1` and `L2`), you can prevent deadlock by always acquiring `L1` before `L2`. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock.

Of course, in more complex systems, more than two locks will exist, and thus total lock ordering may be difficult to achieve (and perhaps is unnecessary anyhow). Thus, a **partial ordering** can be a useful way to structure lock acquisition so as to avoid deadlock. An excellent real example of partial lock ordering can be seen in the memory mapping code in Linux [T+94]; the comment at the top of the source code reveals ten different groups of lock acquisition orders, including simple ones such as "`i_mutex` before `i_mmap_mutex`" and more complex orders such as "`i_mmap_mutex` before `private_lock` before `swap_lock` before `mapping->tree_lock`".

As you can imagine, both total and partial ordering require careful design of locking strategies and must be constructed with great care. Further, ordering is just a convention, and a sloppy programmer can easily ignore the locking protocol and potentially cause deadlock. Finally, lock

TIP: ENFORCE LOCK ORDERING BY LOCK ADDRESS

In some cases, a function must grab two (or more) locks; thus, we know we must be careful or deadlock could arise. Imagine a function that is called as follows: `do_something(mutex_t *m1, mutex_t *m2)`. If the code always grabs `m1` before `m2` (or always `m2` before `m1`), it could deadlock, because one thread could call `do_something(L1, L2)` while another thread could call `do_something(L2, L1)`.

To avoid this particular issue, the clever programmer can use the *address* of each lock as a way of ordering lock acquisition. By acquiring locks in either high-to-low or low-to-high address order, `do_something()` can guarantee that it always acquires locks in the same order, regardless of which order they are passed in. The code would look something like this:

```
if (m1 > m2) {  // grab locks in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (it is not the same lock)
```

By using this simple technique, a programmer can ensure a simple and efficient deadlock-free implementation of multi-lock acquisition.

ordering requires a deep understanding of the code base, and how various routines are called; just one mistake could result in the "D" word[1].

Hold-and-wait

The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically. In practice, this could be achieved as follows:

```
1   pthread_mutex_lock(prevention);   // begin lock acquistion
2   pthread_mutex_lock(L1);
3   pthread_mutex_lock(L2);
4   ...
5   pthread_mutex_unlock(prevention); // end
```

By first grabbing the lock `prevention`, this code guarantees that no untimely thread switch can occur in the midst of lock acquisition and thus deadlock can once again be avoided. Of course, it requires that any time any thread grabs a lock, it first acquires the global prevention lock. For example, if another thread was trying to grab locks `L1` and `L2` in a different order, it would be OK, because it would be holding the prevention lock while doing so.

[1] Hint: "D" stands for "Deadlock".

Note that the solution is problematic for a number of reasons. As before, encapsulation works against us: when calling a routine, this approach requires us to know exactly which locks must be held and to acquire them ahead of time. This technique also is likely to decrease concurrency as all locks must be acquired early on (at once) instead of when they are truly needed.

No Preemption

Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another. Many thread libraries provide a more flexible set of interfaces to help avoid this situation. Specifically, the routine `pthread_mutex_trylock()` either grabs the lock (if it is available) and returns success or returns an error code indicating the lock is held; in the latter case, you can try again later if you want to grab that lock.

Such an interface could be used as follows to build a deadlock-free, ordering-robust lock acquisition protocol:

```
1 top:
2   pthread_mutex_lock(L1);
3   if (pthread_mutex_trylock(L2) != 0) {
4     pthread_mutex_unlock(L1);
5     goto top;
6   }
```

Note that another thread could follow the same protocol but grab the locks in the other order (`L2` then `L1`) and the program would still be deadlock free. One new problem does arise, however: **livelock**. It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. In this case, both systems are running through this code sequence over and over again (and thus it is not a deadlock), but progress is not being made, hence the name livelock. There are solutions to the livelock problem, too: for example, one could add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads.

One final point about this solution: it skirts around the hard parts of using a trylock approach. The first problem that would likely exist again arises due to encapsulation: if one of these locks is buried in some routine that is getting called, the jump back to the beginning becomes more complex to implement. If the code had acquired some resources (other than `L1`) along the way, it must make sure to carefully release them as well; for example, if after acquiring `L1`, the code had allocated some memory, it would have to release that memory upon failure to acquire `L2`, before jumping back to the top to try the entire sequence again. However, in limited circumstances (e.g., the Java vector method mentioned earlier), this type of approach could work well.

Mutual Exclusion

The final prevention technique would be to avoid the need for mutual exclusion at all. In general, we know this is difficult, because the code we wish to run does indeed have critical sections. So what can we do?

Herlihy had the idea that one could design various data structures without locks at all [H91, H93]. The idea behind these **lock-free** (and related **wait-free**) approaches here is simple: using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking.

As a simple example, let us assume we have a compare-and-swap instruction, which as you may recall is an atomic instruction provided by the hardware that does the following:

```
1  int CompareAndSwap(int *address, int expected, int new) {
2    if (*address == expected) {
3      *address = new;
4      return 1; // success
5    }
6    return 0; // failure
7  }
```

Imagine we now wanted to atomically increment a value by a certain amount. We could do it as follows:

```
1  void AtomicIncrement(int *value, int amount) {
2    do {
3      int old = *value;
4    } while (CompareAndSwap(value, old, old + amount) == 0);
5  }
```

Instead of acquiring a lock, doing the update, and then releasing it, we have instead built an approach that repeatedly tries to update the value to the new amount and uses the compare-and-swap to do so. In this manner, no lock is acquired, and no deadlock can arise (though livelock is still a possibility).

Let us consider a slightly more complex example: list insertion. Here is code that inserts at the head of a list:

```
1  void insert(int value) {
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    n->next  = head;
6    head     = n;
7  }
```

This code performs a simple insertion, but if called by multiple threads at the "same time", has a race condition (see if you can figure out why). Of course, we could solve this by surrounding this code with a lock acquire and release:

```
1  void insert(int value) {
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    pthread_mutex_lock(listlock);   // begin critical section
6    n->next  = head;
7    head     = n;
8    pthread_mutex_unlock(listlock); // end critical section
9  }
```

In this solution, we are using locks in the traditional manner[2]. Instead, let us try to perform this insertion in a lock-free manner simply using the compare-and-swap instruction. Here is one possible approach:

```
1  void insert(int value) {
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    do {
6      n->next = head;
7    } while (CompareAndSwap(&head, n->next, n) == 0);
8  }
```

The code here updates the next pointer to point to the current head, and then tries to swap the newly-created node into position as the new head of the list. However, this will fail if some other thread successfully swapped in a new head in the meanwhile, causing this thread to retry again with the new head.

Of course, building a useful list requires more than just a list insert, and not surprisingly building a list that you can insert into, delete from, and perform lookups on in a lock-free manner is non-trivial. Read the rich literature on lock-free and wait-free synchronization to learn more [H01, H91, H93].

Deadlock Avoidance via Scheduling

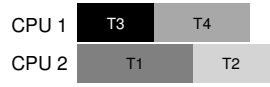
Instead of deadlock prevention, in some scenarios deadlock **avoidance** is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.

For example, assume we have two processors and four threads which must be scheduled upon them. Assume further we know that Thread 1 (T1) grabs locks `L1` and `L2` (in some order, at some point during its execution), T2 grabs `L1` and `L2` as well, T3 grabs just `L2`, and T4 grabs no locks at all. We can show these lock acquisition demands of the threads in tabular form:

[2]The astute reader might be asking why we grabbed the lock so late, instead of right when entering `insert()`; can you, astute reader, figure out why that is likely correct? What assumptions does the code make, for example, about the call to `malloc()`?

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

A smart scheduler could thus compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise. Here is one such schedule:



Note that it is OK for (T3 and T1) or (T3 and T2) to overlap. Even though T3 grabs lock `L2`, it can never cause a deadlock by running concurrently with other threads because it only grabs one lock.

Let's look at one more example. In this one, there is more contention for the same resources (again, locks `L1` and `L2`), as indicated by the following contention table:

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

In particular, threads T1, T2, and T3 all need to grab both locks `L1` and `L2` at some point during their execution. Here is a possible schedule that guarantees that no deadlock could ever occur:

As you can see, static scheduling leads to a conservative approach where T1, T2, and T3 are all run on the same processor, and thus the total time to complete the jobs is lengthened considerably. Though it may have been possible to run these tasks concurrently, the fear of deadlock prevents us from doing so, and the cost is performance.

One famous example of an approach like this is Dijkstra's Banker's Algorithm [D64], and many similar approaches have been described in the literature. Unfortunately, they are only useful in very limited environments, for example, in an embedded system where one has full knowledge of the entire set of tasks that must be run and the locks that they need. Further, such approaches can limit concurrency, as we saw in the second example above. Thus, avoidance of deadlock via scheduling is not a widely-used general-purpose solution.

Detect and Recover

One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected. For example, if an OS froze once a year, you would just reboot it and get happily (or

TIP: DON'T ALWAYS DO IT PERFECTLY (TOM WEST'S LAW)
Tom West, famous as the subject of the classic computer-industry book *Soul of a New Machine* [K81], says famously: "Not everything worth doing is worth doing well", which is a terrific engineering maxim. If a bad thing happens rarely, certainly one should not spend a great deal of effort to prevent it, particularly if the cost of the bad thing occurring is small. If, on the other hand, you are building a space shuttle, and the cost of something going wrong is the space shuttle blowing up, well, perhaps you should ignore this piece of advice.

grumpily) on with your work. If deadlocks are rare, such a non-solution is indeed quite pragmatic.

Many database systems employ deadlock detection and recovery techniques. A deadlock detector runs periodically, building a resource graph and checking it for cycles. In the event of a cycle (deadlock), the system needs to be restarted. If more intricate repair of data structures is first required, a human being may be involved to ease the process.

More detail on database concurrency, deadlock, and related issues can be found elsewhere [B+87, K87]. Read these works, or better yet, take a course on databases to learn more about this rich and interesting topic.

32.4 Summary

In this chapter, we have studied the types of bugs that occur in concurrent programs. The first type, non-deadlock bugs, are surprisingly common, but often are easier to fix. They include atomicity violations, in which a sequence of instructions that should have been executed together was not, and order violations, in which the needed order between two threads was not enforced.

We have also briefly discussed deadlock: why it occurs, and what can be done about it. The problem is as old as concurrency itself, and many hundreds of papers have been written about the topic. The best solution in practice is to be careful, develop a lock acquisition order, and thus prevent deadlock from occurring in the first place. Wait-free approaches also have promise, as some wait-free data structures are now finding their way into commonly-used libraries and critical systems, including Linux. However, their lack of generality and the complexity to develop a new wait-free data structure will likely limit the overall utility of this approach. Perhaps the best solution is to develop new concurrent programming models: in systems such as MapReduce (from Google) [GD02], programmers can describe certain types of parallel computations without any locks whatsoever. Locks are problematic by their very nature; perhaps we should seek to avoid using them unless we truly must.

References

[B+87] "Concurrency Control and Recovery in Database Systems"
Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman
Addison-Wesley, 1987
The classic text on concurrency in database management systems. As you can tell, understanding concurrency, deadlock, and other topics in the world of databases is a world unto itself. Study it and find out for yourself.

[C+71] "System Deadlocks"
E.G. Coffman, M.J. Elphick, A. Shoshani
ACM Computing Surveys, 3:2, June 1971
The classic paper outlining the conditions for deadlock and how you might go about dealing with it. There are certainly some earlier papers on this topic; see the references within this paper for details.

[D64] "Een algorithme ter voorkoming van de dodelijke omarming"
Edsger Dijkstra
Circulated privately, around 1964
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF
Indeed, not only did Dijkstra come up with a number of solutions to the deadlock problem, he was the first to note its existence, at least in written form. However, he called it the "deadly embrace", which (thankfully) did not catch on.

[GD02] "MapReduce: Simplified Data Processing on Large Clusters"
Sanjay Ghemawhat and Jeff Dean
OSDI '04, San Francisco, CA, October 2004
The MapReduce paper ushered in the era of large-scale data processing, and proposes a framework for performing such computations on clusters of generally unreliable machines.

[H01] "A Pragmatic Implementation of Non-blocking Linked-lists"
Tim Harris
International Conference on Distributed Computing (DISC), 2001
A relatively modern example of the difficulties of building something as simple as a concurrent linked list without locks.

[H91] "Wait-free Synchronization"
Maurice Herlihy
ACM TOPLAS, 13:1, January 1991
Herlihy's work pioneers the ideas behind wait-free approaches to writing concurrent programs. These approaches tend to be complex and hard, often more difficult than using locks correctly, probably limiting their success in the real world.

[H93] "A Methodology for Implementing Highly Concurrent Data Objects"
Maurice Herlihy
ACM TOPLAS, 15:5, November 1993
A nice overview of lock-free and wait-free structures. Both approaches eschew locks, but wait-free approaches are harder to realize, as they try to ensure than any operation on a concurrent structure will terminate in a finite number of steps (e.g., no unbounded looping).

[J+08] "Deadlock Immunity: Enabling Systems To Defend Against Deadlocks"
Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
OSDI '08, San Diego, CA, December 2008
An excellent recent paper on deadlocks and how to avoid getting caught in the same ones over and over again in a particular system.

[K81] "Soul of a New Machine"
Tracy Kidder, 1980
A must-read for any systems builder or engineer, detailing the early days of how a team inside Data General (DG), led by Tom West, worked to produce a "new machine." Kidder's other books are also excellent, including Mountains beyond Mountains*. Or maybe you don't agree with us, comma?*

[K87] "Deadlock Detection in Distributed Databases"
Edgar Knapp
ACM Computing Surveys, 19:4, December 1987
An excellent overview of deadlock detection in distributed database systems. Also points to a number of other related works, and thus is a good place to start your reading.

[L+08] "Learning from Mistakes — A Comprehensive Study on
Real World Concurrency Bug Characteristics"
Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou
ASPLOS '08, March 2008, Seattle, Washington
The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at Y.Y. Zhou's or Shan Lu's web pages for many more interesting papers on bugs.

[T+94] "Linux File Memory Map Code"
Linus Torvalds and many others
Available: http://lxr.free-electrons.com/source/mm/filemap.c
Thanks to Michael Walfish (NYU) for pointing out this precious example. The real world, as you can see in this file, can be a bit more complex than the simple clarity found in textbooks...

Homework

This homework lets you explore some real code that deadlocks (or avoids deadlock). The different versions of code correspond to different approaches to avoiding deadlock in a simplified `vector_add()` routine. Specifically:

- `vector-deadlock.c`: This version of `vector_add()` does not try to avoid deadlock and thus may indeed do so.
- `vector-global-order.c`: This version acquires locks in a global order to avoid deadlock.
- `vector-try-wait.c`: This version is willing to release a lock when it senses deadlock might occur.
- `vector-avoid-hold-and-wait.c`: This version uses a global lock around lock acquisition to avoid deadlock.
- `vector-nolock.c`: This version uses an atomic fetch-and-add instead of locks.

See the README for details on these programs and their common substrate.

Questions

1. First let's make sure you understand how the programs generally work, and some of the key options. Study the code in the file called `vector-deadlock.c`, as well as in `main-common.c` and related files.

   Now, run `./vector-deadlock -n 2 -l 1 -v`, which instantiates two threads (`-n 2`), each of which does one vector add (`-l 1`), and does so in verbose mode (`-v`). Make sure you understand the output. How does the output change from run to run?

2. Now add the `-d` flag, and change the number of loops (`-l`) from 1 to higher numbers. What happens? Does the code (always) deadlock?

3. How does changing the number of threads (`-n`) change the outcome of the program? Are there any values of `-n` that ensure no deadlock occurs?

4. Now examine the code in `vector-global-order.c`. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? Also, why is there a special case in this `vector_add()` routine when the source and destination vectors are the same?

5. Now run the code with the following flags: `-t -n 2 -l 100000 -d`. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?

6. What happens if you turn on the parallelism flag (`-p`)? How much would you expect performance to change when each thread is working on adding different vectors (which is what `-p` enables) versus working on the same ones?

7. Now let's study `vector-try-wait.c`. First make sure you understand the code. Is the first call to `pthread_mutex_trylock()` really needed?

   Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

8. Now let's look at `vector-avoid-hold-and-wait.c`. What is the main problem with this approach? How does its performance compare to the other versions, when running both with `-p` and without it?

9. Finally, let's look at `vector-nolock.c`. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?

10. Now compare its performance to the other versions, both when threads are working on the same two vectors (no `-p`) and when each thread is working on separate vectors (`-p`). How does this no-lock version perform?

An Introduction to DOS FAT Volume and File Structure

Mark Kampe markk@cs.ucla.edu

1. Introduction

When the first personal computers with disks became available, they were very small (a few megabytes of disk and a few dozen kilobytes of memory). A file system implementation for such machines had to impose very little overhead on disk space, and be small enough to fit in the BIOS ROM. BIOS stands for **BASIC I/O Subsystem**. Note that the first word is all upper-case. The purpose of the BIOS ROM was to provide run-time support for a BASIC interpreter (which is what Bill Gates did for a living before building DOS). DOS was never intended to provide the features and performance of real timesharing systems.

Disk and memory size have increased in the last thirty years, People now demand state-of-the-art power and functionality from their PCs. Despite the evolution that the last decades have seen, old standards die hard. Much as European train tracks maintain the same wheel spacing used by Roman chariots, most modern OSs still support DOS FAT file systems. DOS file systems are not merely around for legacy reasons. The ISO 9660 CDROM file system format is a descendent of the DOS file system.

The DOS FAT file system is worth studying because:

- It is heavily used all over the world, and is the basis for more modern file system (like 9660).
- It provides reasonable performance (large transfers and well clustered allocation) with a very simple implementation.
- It is a very successful example of "linked list" space allocation.

2. Structural Overview

All file systems include a few basic types of data structures:

- bootstrap

  code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first track of the first cylinder for the boot strap.

- volume descriptors

  information describing the size, type, and layout of the file system ... and in particular how to find the other key meta-data descriptors.

- file descriptors

  information that describes a file (ownership, protection, time of last update, etc.) and points where the actual data is stored on the disk.

- free space descriptors

  lists of blocks of (currently) unused space that can be allocated to files.

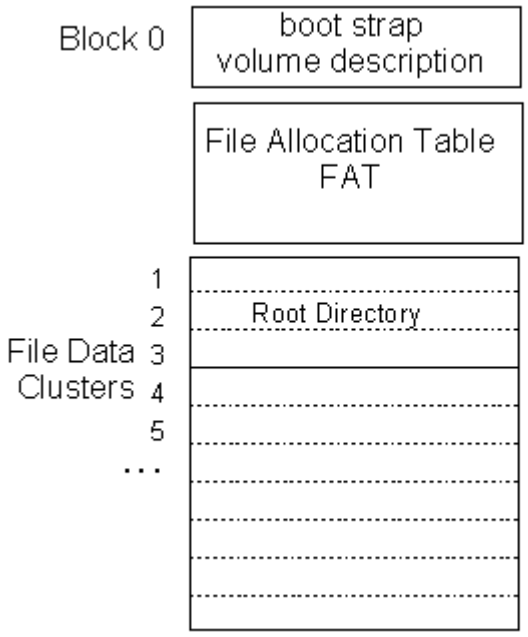
- file name descriptors

  data structures that user-chosen names with each file.

DOS FAT file systems divide the volume into fixed-sized (physical) blocks, which are grouped into larger fixed-sized (logical) block clusters.

The first block of DOS FAT volume contains the bootstrap, along with some volume description information. After this comes a much longer **File Allocation Table** (FAT from which the file system takes its name). The File Allocation Table is used, both as a free list, and to keep track of which blocks have been allocated to which files.

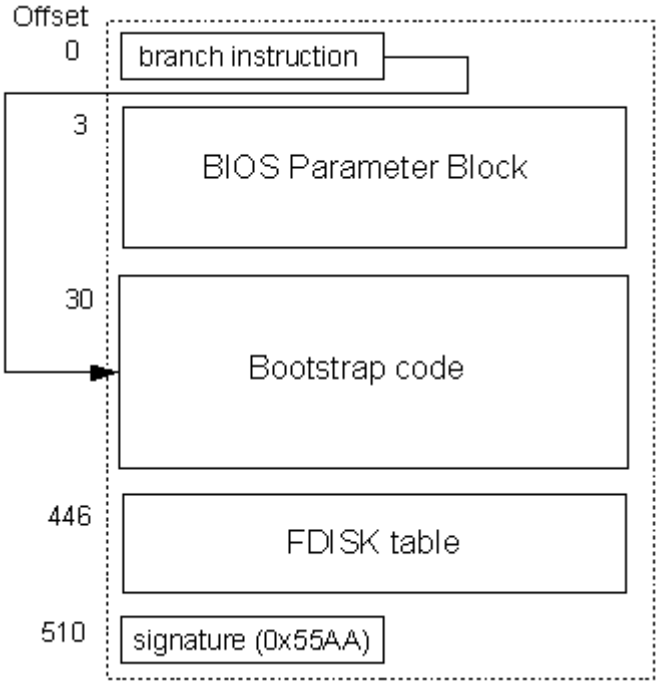
The remainder of the volume is data clusters, which can be allocated to files and directories. The first file on the volume is the root directory, the top of the tree from which all other files and directories on the volume can be reached.



3. Boot block BIOS Parameter Block and FDISK Table

Most file systems separate the first block (pure bootstrap code) from volume description information. DOS file systems often combine these into a single block. The format varies between (partitioned) hard disks and (unpartitioned) floppies, and between various releases of DOS and Windows ... but conceptually, the boot record:

- begins with a branch instruction (to the start of the real boostrap code).
- followed by a volume description (BIOS Parameter Block)
- followed by the real bootstrap code
- followed by an optional disk partitioning table
- followed by a signature (for error checking).



3.1 BIOS Parameter Block

After the first few bytes of the bootstrap comes the BIOS parameter block, which contains a brief summary of the device and file system. It describes the device geometry:

- number of bytes per (physical) sector
- number of sectors per track
- number of tracks per cylinder
- total number of sectors on the volume

It also describes the way the file system is layed out on the volume:

- number of sectors per (logical) cluster
- the number of reserved sectors (not part of file system)
- the number of Alternate File Allocation Tables
- the number of entries in the root directory

These parameters enable the OS to interpret the remainder of the file system.

3.2 FDISK Table

As disks got larger, the people at MicroSoft figured out that their customers might want to put multiple file systems on each disk. This meant they needed some way of partitioning the disk into logical sub-disks. To do this, they added a small partition table (sometimes called the FDISK table, because of the program that managed it) to the end of the boot strap block.

This FDISK table has four entries, each capable of describing one disk partition. Each entry includes

- A partition type (e.g. Primary DOS partition, UNIX partition).
- An ACTIVE indication (is this the one we boot from).
- The disk address where that partition starts and ends.
- The number of sectors contained within that partition.

Partn	Type	Active	Start (C:H:S)	End (C:H:S)	Start (logical)	Size (sectors)

1	LINUX	True	1:0:0	199:7:49	400	79,600
2	Windows NT		200:0:0	349:7:49	80,000	60,000
3	FAT 32		350:0:0	399:7:49	140,000	20,000
4	NONE					

In older versions of DOS the starting/ending addresses were specified as cylinder/sector/head. As disks got larger, this became less practical, and they moved to logical block numbers.

The addition of disk partitioning also changed the structure of the boot record. The first sector of a disk contains the Master Boot Record (MBR) which includes the FDISK table, and a bootstrap that finds the active partition, and reads in its first sector (Partition Boot Record). Most people (essentially everyone but Bill Gates :-) make their MBR bootstrap ask what system you want to boot from, and boot the active one by default after a few seconds. This gives you the opportunity to choose which OS you want to boot. Microsoft makes this decision for you ... you want to boot Windows.

The structure of the Partition Boot Record is entirely operating system and file system specific ... but for DOS FAT file system partitions, it includes a BIOS Parameter block as described above.

4. File Descriptors (directories)

In keeping with their desire for simplicity, DOS file systems combine both file description and file naming into a single file descriptor (directory entries). A DOS directory is a file (of a special type) that contains a series of fixed sized (32 byte) directory entries. Each entry describes a single file:

- an 11-byte name (8 characters of base name, plus a 3 character extension).
- a byte of attribute bits for the file, which include:
    - Is this a file, or a sub-directory.
    - Has this file changed since the last backup.
    - Is this file hidden.
    - Is this file read-only.
    - Is this a system file.
    - Does this entry describe a volume label.
- times and dates of creation and last modification, and date of last access.
- a pointer to the first logical block of the file. (This field is only 16 bits wide, and so when Microsoft introduced the FAT32 file system, they had to put the high order bits in a different part of the directory entry).
- the length (number of valid data bytes) in the file.

Name (8+3)	Attributes	Last Changed	First Cluster	Length
.	DIR	08/01/03 11:15:00	61	2,048
..	DIR	06/20/03 08:10:24	1	4,096
MARK	DIR	10/15/04 21:40:12	130	1,800
README.TXT	FILE	11/02/04 04:27:36	410	31,280

If the first character of a files name is a NULL (0x00) the directory entry is unused. The special character (0xE5) in the first character of a file name is used to indicate that a directory entry describes a deleted file. (See the section on Garbage collection below)

Note on times and dates:

DOS stores file modification times and dates as a pair of 16-bit numbers:

- 7 bits of year, 4 bits of month, 5 bits of day of month
- 5 bits of hour, 6 bits of minute, 5 bits of seconds (x2).

All file systems use dates relative to some epoch (time zero). For DOS, the epoch is midnight, New Year's Eve, January 1, 1980. A seven bit field for years means that the the DOS calendar only runs til 2107. Hopefully, nobody will still be using DOS file systems by then :-)

5. Links and Free Space (File Allocation Table)

Many file systems have very compact (e.g. bitmap) free lists, but most of them use some per-file data structure to keep track of which blocks are allocated to which file. The DOS File Allocation Table is a relatively unique design. It contains one entry for each logical block in the volume. If a block is free, this is indicated by the FAT entry. If a block is allocated to a file, the FAT entry gives the logical block number of the **next** logical block in the file.

5.1 Cluster Size and Performance

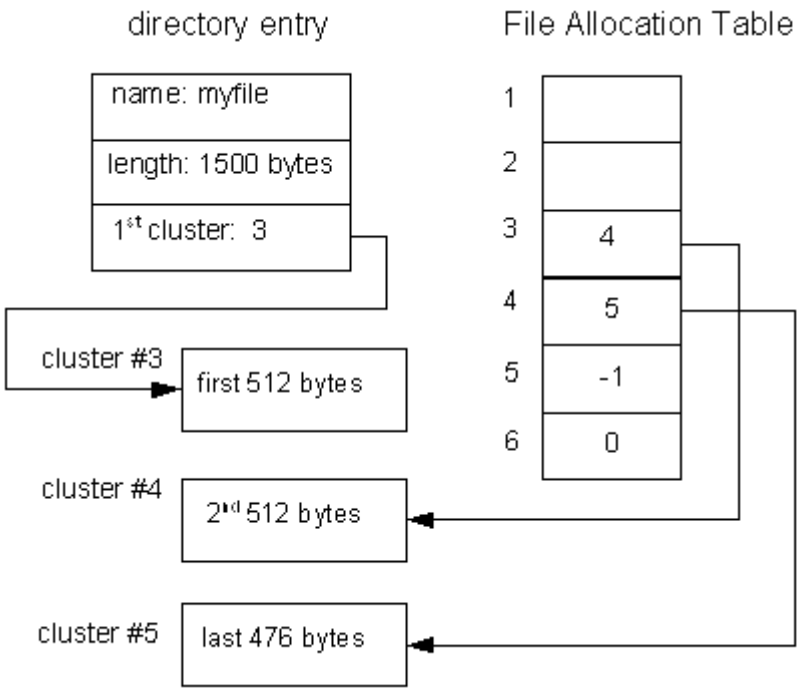
Space is allocated to files, not in (physical) blocks, but in (logical) multi-block clusters. The number of clusters per block is determined when the file system is created.

Allocating space to files in larger chunks improves I/O performance, by reducing the number of operations required to read or write a file. This comes at the cost of higher internal fragmentation (since, on average, half of the last cluster of each file is left unused). As disks have grown larger, people have become less concerned about internal fragmentation losses, and cluster sizes have increased.

The maximum number of clusters a volume can support depends on the width of the FAT entries. In the earliest FAT file systems (designed for use on floppies, and small hard drives). An 8-bit wide FAT entry would have been too small (256 * 512 = 128K bytes) to describe even the smallest floppy, but a 16-bit wide FAT entry would have been ludicrously large (8-16 Megabytes) ... so Microsoft compromised and adopted 12-bit wide FAT entries (two entries in three bytes). These were called FAT-12 file systems. As disks got larger, they created 2-byte wide (FAT-16) and 4-byte wide (FAT-32) file systems.

5.2 Next Block Pointers

A file's directory entry contains a pointer to the first cluster of that file. The File Allocation Table entry for that cluster tells us the cluster number **next** cluster in the file. When we finally get to the last cluster of the file, its FAT entry will contain a -1, indicating that there is no next block in the file.



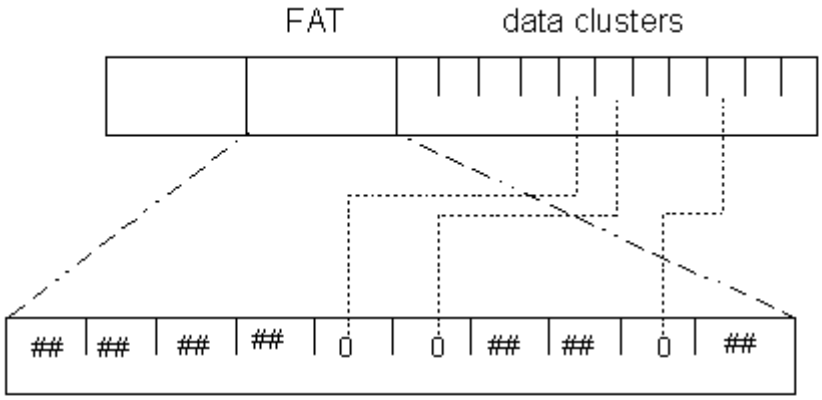
The "next block" organization of the FAT means that in order to figure out what physical cluster is the third logical block of ar file, must know the physical cluster number of the second logical block. This is not usually a problem, because almost all file access is sequential (reading the first block, and then the second, and then the third ...).

If we had to go to disk to re-read the FAT each time we needed to figure out the next block number, the file system would perform very poorly. Fortunately, the FAT is so small (e.g. 512 bytes per megabyte of file system) that the entire FAT can be kept in memory as long as a file system is in use. This means that successor block numbers can be looked up without the need to do any additional disk I/O. It is easy to imagine

5.3 Free Space

The notion of "next block" is only meaningful for clusters that are allocated to a file ... which leaves us free to use the FAT entries associated with free clusters as a free indication. Just as we reserved a value (-1) to mean **end of file** we can rerserve another value (0) to mean **this cluster is free**.

To find a free cluster, one has but to search the FAT for an entry with the value -2. If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file.



Each FAT entry corresponds to one data cluster

A FAT entry of 0 means the corresponding data cluster is not allocated to any file.

5.4 Garbage Collection

Older versions of FAT file systems did not bother to free blocks when a file was deleted. Rather, they merely crossed out the first byte of the file name in the directory entry (with the reserved value 0xE5). This had the advantage of greatly reducing the amount of I/O associated with file deletion ... but it meant that DOS file systems regularly ran out of space.

When this happened, they would initiate garbage collection. Starting from the root directory, they would find every "valid" entry. They would follow the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of all sub-directories. After completing the enumeration of all allocated clusters, they inferred that any cluster not found in some file was free, and marked them as such in the File Allocation Table.

This "feature" was probably motivated by a combination of laziness and a desire for performance. It did, however, have an advantage. Since clusters were not freed when files were deleted, they could not be reallocated until after garbage colection was performed. This meant that it might be possible to recover the contents of deleted files for quite a while. The opportunity this created was large enough to enable Peter Norton to start a very successful company.

6. Descendents of the DOS file system

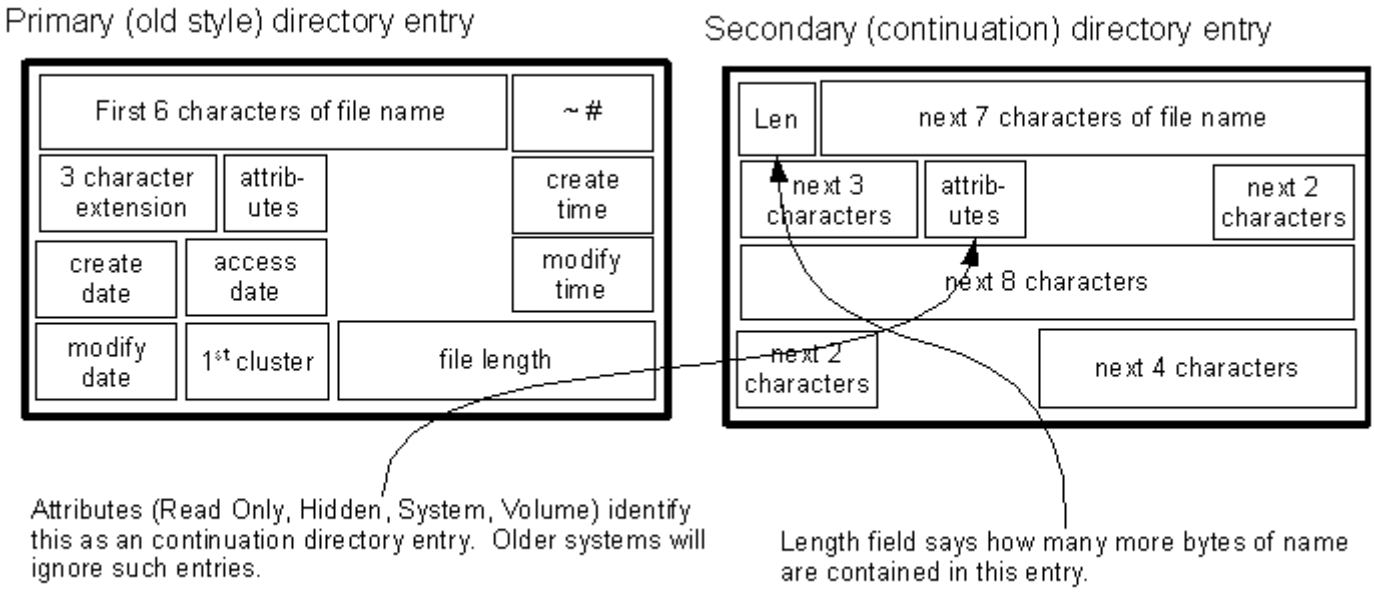
The DOS file system has evolved with time. Not only have wider (16- and 32-bit) FAT entries been used to support larger disks, but other features have been added. The last stand-alone DOS product was DOS 6.x. After this, all DOS support was under Windows, and along with the change to Windows came an enhanced version of the FAT file system called Virtual FAT (or simply VFAT).

6.1 Long File Names

Most DOS and Windows systems were used for personal productivity, and their users didn't demand much in the way of file system features. Their biggest complaints were about the 8+3 file names. Windows users demanded longer and mixed-case file names.

The 32 byte directory entries didn't have enough room to hold longer names, and changing the format of DOS directories would break hundreds or even thousands of applications. It wasn't merely a matter of importing files from old systems to new systems. DOS diskettes are commonly used to carry files between various systems ... which means that old systems still had to be able to read the new directories. They had to find a way to support longer file names without making the new files unreadable by older systems.

The solution they came up was to put the extended filenames in additional (auxiliary) directory entries. Each file would be described by an old-format directory entry, but supplementary directory entries (following the primary directory entry) could provide extenstions to the file name. To keep older systems from being confused by the new directory entries, they were tagged with a very unusual set of attributes (hidden, system, read-only, volume label). Older systems would ignore new entries, but would still be able to access new files under their 8+3 names in the old-style directory entries. New systems would recognize the new directory entries, and be able to access files by either the long or the short names.



The addition of long file names did create one problem for the old directory entries. What would you do if you had two file names that differred only after the 8th character (e.g. datafileread.c and datafilewrite.c)? If we just used the first 8 characters of the file name in the old directory entries (datafile), we would have two files with the same name, and this is illegal. For this reason, the the short file names are not merely the first eight characters of the long file names. Rather, the last two bytes of the short name were merely made unambiguous (e.g. "~1", "~2", etc).

6.2 Alternate/back-up FATs

The File Allocation Table is a very concise way of keeping track of all of the next-block pointers in the file system. If anything ever happened to the File Allocation Table, the results would be disastrous. The directory entries would tell us where the first blocks of all files were, but we would have no way of figuring out where the remainder of the data was.

Events that corrupt the File Allocation Table are extremely rare, but the consequences of such an incident are catastrophic. To protect users from such eventualities, MicroSoft added support for alternate FATs. Periodically, the primary FAT would be copied to one of the pre-reserved alternat FAT locations. Then if something bad happened to the primary FAT, it would still be possible to read most files (files created before the copy) by using the back-up FAT. This is an imperfect solution, as losing new files is bad ... but losing all files is worse.

6.3 ISO 9660

When CDs were proposed for digital storage, everyone recognized the importance of a single standard file system format. Dueling formats would raise the cost of producing new products ... and this would be a lose for everyone. To respond to this need, the International Standards Organization chartered a sub-committee to propose such a standard.

The failings of the DOS file system were widely known by this time, but, as the most widely used file system on the planet, the committee members could not ignore it. Upon examination, it became clear that the most ideomatic features of the DOS file system (the File Allocation Table) were irrelevent to a CDROM file system (which is written only once):

- We don't need to keep track of the free space on a CD ROM. We write each file contiguously, and the next file goes immidiately after the last one.
- Because files can be written contiguously, we don't need any "next block" pointers. All we need to know about a file is where its first block resides.

It was decided that ISO 9660 file systems would (like DOS file systems) have tree structured directory hierarchies, and that (like DOS) each directory entry would describe a single file (rather than having some auxiliary data structure like and I-node to do this). 9660 directory entries, like DOS directory entries, contain:

- file name (within the current directory)
- file type (e.g. file or directory)
- location of the file's first block
- number of bytes contained in the file
- time and date of creation

They did, however, learn from DOS's mistakes:

- Realizing that new information would be added to directory entries over time, they made them variable length. Each directory entry begins with a length field (giving the number of bytes in this directory entry, and thus the number of bytes until the next directory entry).
- Recognizing the need to support long file names, they also made the file name field in each entry a variable length field.
- Recognizing that, over time, people would want to associate a wide range of attributes with files, they also created a variable length extened attributes section after the file name. Much of this section has been left unused, but they defined several new attributes for files:
    - file owner
    - owning group
    - permissions
    - creation, modification, effective, and expiration times
    - record format, attributes, and length information

But, even though 9660 directory entries include much more information than DOS directory entries, it remains that 9660 volumes resemble DOS file systems much more than they resemble any other file system format. And so, the humble DOS file system is reborn in a new generation of products.

7. Summary

DOS file systems are very simple, They don't support multiple links to a file, or symbolic links, or even multi-user access control. They are also and very economical in terms of the space they take up. Free block lists, and file block pointers are combined into a single (quite compact) File Allocation Table. File descriptors are incorporated into the directory entries. And for all of these limitations, they are probably the most widely used

file system format on the planet. Despite their primitiveness, DOS file systems were used as the basis for much newer CD ROM file system designs.

What can we infer from this? That most users don't need alot of fancy features, and that the DOS file system (primitive as it may be) covers their needs pretty well.

It is also noteworthy that when Microsoft was finally forced to change the file system format to get past the 8.3 upper case file name limitations, they chose to do so with a klugy (but upwards compatible) solution using additional directory entries per file. The fact that they chose such an implementation clearly illustrates the importance of maintaining media interchangability with older systems. This too is a problem that all (successful) file systems will eventually face.

8. References

DOS file system information

- PC Guide's Overview of DOS FAT file systems. (this is a pointer to the long filename article ... but the entire library is nothing short of excellent).
- Free BSD sources, PCFS implementation, BIOS Parameter block format, and (Open Solaris) FDISK table format.
- Free BSD sources, PCFS implementation, Directory Entry format

9660 file system information

- WIkipedia Introduction to ISO 9660 file systems
- Free BSD sources, ISOFS implementation, ISO 9660 data structures.

Cluster Concepts

by Mark Kampe

Cluster

There are many different types of clusters. Common types include:

- load sharing clusters, which divide work among the members.
- high availability clusters, where back-up nodes take over when primary nodes fail.
- information sharing clusters, which ensure the dissemination of information throughout a network.

There are so many different goals and architectures that Greg Phister (in "In Search of Clusters") came to the conclusion that it is very difficult to even define the term. About the only thing we can say for certain is that a cluster is a networked connection of nodes, all of whom agree that they are part of a cluster.

Membership

If a cluster is defined as a networked connection of nodes who consider themselves to be participants in the cluster, then obviously "membership" is a key concept.

We can distinguish two types of membership:

- potential, eligible or designated members
- active or currently participating members

This distinction is important because only active members can communicate with one-another. Thus it is that the term "membership" is most commonly used to describe only the currently participating members.

It is very important to know who the current (active) cluster members are. We may, for instance, be required to make sure that each of them has been informed of some operation before we are allowed to perform it. Cluster membership often comes with responsibilities (e.g. a guarantee to respond to certain requests within a certain period if time). Thus it is vital that we know when nodes enter and leave the cluster.

In most clusters, a node has to be explicitly configured or provisioned into the cluster ... so that the set of potential members is well known, and perhaps even closed to new members. There are, however, some types of clusters where any node is welcooe to join at any time.

Node Redundancy

In a clustered system, work is divided among the active members. To reduce distributed synchronization, it is common to partition the work (e.g. desigate each server responsible for a certain subset (e.g. a file system, a range of keys, etc) of requests, and route all requests to their designated owner). In such systems, we can talk about *primaries* (the designated owners) and *secondaries* (nodes who are prepared to take over for a primary if he fails).

There are three fundamentally different approaches to take to high availability:

- Active/Stand-By
  The system is divided into *active* and *stand-by* nodes. The incoming requests are partitioned among the active nodes. The stand-by nodes are idle until an active node fails, at which point a stand-by node takes over his work.

- Active/Active
  The incoming requests are partitioned among all of the available nodes. If one node fails, his work will be redistributed among the survivors.

An active/active architecture achieves better resource utilization, and so may be more cost-effective. But when a failure occurrs, the load on the surviving nodes is increased and so performance may suffer. An active/stand-by architecture normally has idle capacity, but may not suffer any performance degradataion after a failure.

We can also look at how quickly a successor is able to take over a failed node's responsibilities. In some architectures, all operations are mirrored to the secondaries, enabling them to very quickly assume the primary role. Such secondaries are called *hot standbys*. In other systems, the secondary waits to be notified of the primary's failure, after which it opens and reads the failed primary's last check-point or journal. The former approach results in more network and CPU load, but much faster fail-overs. The latter approach consumes fewer resources during normal operation, but may take much longer to resume service after a primary has failed.

Heart Beat

Ideally nodes will announce the fact that they are joining the cluster, or are about to leave it. This is not always the case:

1. a system may crash.
2. the clustering applications may crash.
3. a node may become so busy that the clustering applications cannot run.
4. a network interface or link may fail.

Since we cannot be sure that member will notify the other members before he leaves the cluster, we need a means of detecting a member who has dropped unexpectedly out of the cluster. The most common techique is called a "heart beat". A heart beat is a message, regularly exchanged between cluster members. These may be special messages, or they may be piggy-backed on other traffic. If too much time passes without our having received a heart-beat from some node, we will assume that node has failed, and is no longer an active cluster member.

The failure of a node may (in some clusters) have serious consequences (e.g. the freeing of all resources allocated to that node, and the aborting of all in progress transactions from that node). To prevent "false alarms", many systems perform heart-beats over multiple channels, or have a back-up link with which they attempt to confirm a failure before reporting a node to be dead.

Cluster Master and Election

It is often convenient to elect or designate one node to be the cluster master:

- Coming to a mutual agreement between multiple nodes can be a complex process (e.g. Three Phase Commits). If one node is designated a cluster master, that node can serve as a central point of syncrhronization and/or control for operations in the cluster.
- Rather than requiring all nodes to heart-beat one-another, it is more economical to simply have all nodes heart-beat with the cluster master. He will detect the failure of any other node, and all nodes will detect his failure.

The election of a cluster master may, itself, be a complex process ... but having performed that process may eliminate the need for any further negotiations. There are numerous well established election/concensus algorithms. One of the best known is Leslie Lamport's Paxos algorithm.

Split Brain

A pathological network failure might divide a cluster into multiple sub-clusters, which cannot communicate with one-another. Such an event is sometimes referred to as a "partitioning" of the network. If the cluster manages critical resources (e.g. a database or nuclear warhead), it is possible that the independent sub-clusters will all continue operating and make independent but incompatible decisions. Such a condition is called "split-brain" (as if two halves of our brain were working independently and at cross-purposes).

There are two standard approaches to preventing "split-brain":

a. quorum
b. voting devices

Quorum

If there are N potential members in a cluster, we can build in a rule that says a cluster cannot form with fewer than (N/2)+1 members. This acomplishes two purposes:

- It makes it impossible for any partitioning to result in two viable sub-clusters (because N nodes cannot be divided into two groups that both contain at least (N/2)+1 nodes).
- It ensures that any decision made (and persisted) by this quorum will be remembered by any future quorum (because any group of (N/2)+1 nodes will have at least one member in common with every other group of (N/2)+1 nodes that has ever existed.

The problem with using a numerical quorum is that if (N/2)+1 nodes have been damaged, it will be impossible for the surviving nodes to form a new cluster ... even if there is no split-brain.

Voting Devices

If there is a single piece of hardware in the cluster, that must be present for the cluster to function, and that can only be owned by one node, that device can be used as a voting device.

Consider, for instance, a shared disk. If that disk is absolutely required to provide service, a node that no longer has access to that disk cannot provide service (and hence is not eligible to form a cluster). But what if two nodes can both talk to the disk, but cannot communicate with one-another? They may be able to use the disk as a voting device ... e.g. by writing a recent time-stamp into a well-known block.

Some clusters include resources that can easily serve as voting devices. There are also specially built (very reliable) voting devices that exist solely for this purpose. If there is a voting device, a cluster could be formed by a single node ... because the voting device would prevent split-brain.

Fencing

What if, you were not only sufferring from schizophrenia, but the other side of your brain had actually gone rogue, and was trying to commit acts of mayhem against you and others? In some clustered systems, it must be assumed that if a node has fallen out of the cluster, he is no longer trust-worthy ... and must be "fenced-out" of the cluster. There are two common approaches to fencing:

- reservable devices
  Some devices can be told which interface to listen to, and not to listen to the other interface. This is often done with dual-ported disks. The node that has seized the quorum device will then instruct the quorum device not to accept commands from any other node.
- remote power control
  Some clustered systems come with remote power controllers, and a node that has seized control of the

cluster from a previous (apparently failed) cluster master will often power-off or reset the previous master, to ensure that he does not continue to vie for control of the cluster and its resources.

Health Monitoring and Recovery

Introduction

Suppose that a system seems to be wedged ... it has not recently made any progress. How would we determine if the system was deadlocked?

- identify all of the blocked processes.
- identify the resource on which each process is blocked.
- identify the owner of each blocking resource.
- determine whether or not the implied graph contains any loops.

How would we determine that the system might be wedged, so that we could invoke deadlock analysis? It may not be possible to identify the owners of all of the involved resources, or even all of the resources. Worse still, a process may not actually be blocked, but merely waiting for a message or event (that has, for some reason, not yet been sent). And if we did determine that a deadlock existed, what would we do? Kill a random process? This might break the circular dependency, but would the system continue to function properly after such an action? Formal deadlock detection in real systems is:

Formal deadlock detection in real systems ...

a. is difficult to perform
b. is inadequate to diagnose most hangs
c. does not tell us how to fix the problem

Fortunately there is a simpler technique, that is far more effective at detecting, diagnosing, and repairing a much wider range of problems: health monitoring and managed recovery.

Health Monitoring

We said that we could invoke deadlock detection whenever we thought that the system might not be making progress. How could we know whether or not the system was making progress? There are many ways to do this:

- by having an internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
- by having the service send periodic heart-beat messages to a health monitoring service.
- by having an external health monitoring service send periodic test requests to the service that is being monitored, and ascertain that they are being responded to correctly and in a timely fashion.

Any of these techiques could alert us of a potential deadlock, livelock, loop, or a wide range of other failures. But each of these techniques has different strengths and weaknesses:

- heart beat messages can only tell us that the node and application are still up and running. They cannot tell us if the application is actually serving requests.
- an external health monitoring service can determine whether or not the monitored application is responding to requests. But this does not mean that some other requests have not been deadlocked or otherwise wedged.
- an internal monitoring agent might be able to monitor logs or statistics to determine that the service is processing requests at a reasonable rate (and perhaps even that no requests have been waiting too long). But if the internal monitoring agent fails, it may not be able to detect and report errors.

Many systems use a combination of these methods:

- the first line of defense is an internal monitoring agent that closely watches key applications to detect failures and hangs.
- if the internal monitoring agent is responsible for sending heart-beats (or health status reports) to a central monitoring agent, a failure of the internal monitoring agent will be noticed by the central monitoring agent.
- an external test service that periodically generates test transactions provides an independent assessment that might include external factors (e.g. switches, load balancers, network connectivity) that would not be tested by the internal and central monitoring services.

Managed Recovery

Suppose that some or all of these monitoring mechanisms determine that a service has hung or failed. What can we do about it? Highly available services must be designed for restart, recovery, and fail-over:

- The software should be designed so that any process in the system can be killed and restarted at any time. When a process restarts, it should be able to reestablish communiction with the other processes and resume working with minimal disruption.
- The software should be designed to support multiple levels of restart. Examples might be:
    - warm-start ... restore the last saved state (from a database or from information obtained from other processes) and resume service where we left off.
    - cold-start ... ignore any saved state (which may be corrupted) and restart new operations from scratch.
- The software might also designed for progressively escalating scope of restarts:
    - restart only a single process, and expect it to resync with the other processes when it comes back up.
    - restart all of the software on a single node.
    - restart a group of nodes, or the entire system.

Designing software in this way gives us the opportunity to begin with minimal disruption, restarting only the process that seems to have failed. In most cases this will solve the problem, but perhaps:

- process A failed as a result of an incorrect request received from process B.
- the operation that caused process A to fail is still listed in the database, and when process A restarts, it may attempt to re-try the same operation and fail agian.
- the operation that caused process A to fail may have been mirrored to other systems, that will also experience or cause additional failures.

For all of these reasons it is desirable to have the ability to escalate to progressively more complete restarts of a progressively wider range of components.

False Reports

Ideally a problem will be found by the internal monitoring agent on the affected node, which will automatically trigger a restart of the affected software on that node. Such prompt local action has the potential to fix the problem before other nodes even notice that there was a problem.

But suppose a central monitoring service notes that it has not received a heart-beat from process A. What might this mean?

- It might mean that the node has failed.
- It might mean that the process has failed.
- It might mean that the system is loaded and the heart-beat message was delayed.
- It might mean that a network error prevented or delayed the delivery of a heart-beat message.

Declaring a process to have failed can potentially be a very expensive operation. It might cause the cancellation and retransmission of all requests that had been sent to the failed process or node. It might cause other servers to start trying to recover work-in-progress from the failed process or node. And this recovery might involve a great deal of network traffic and system activity. We don't want to start an expensive fire-drill unless we are pretty certain that a process has failed.

- the best option would be for a failing process to detect its own problem, inform its partners, and shut-down cleanly.
- if the failure is detected by a missing heart-beat, it may be wise to wait until multiple heart-beat messages have been missed before declaring the process to have failed.
- in some cases, we might want to wait for multiple other processes/nodes to complain.

But there is a trade-off here. If we do not take the time to confirm suspected failures, we may suffer unnecessary service disruptions from forcing fail-overs from healthy servers. On the other hand, if we wait too long before initiating fail-overs, we are prolonging the service outage. These so-called "mark-out thresholds" often require a great deal of tuning.

Other Managed Restarts

As we consider failure and restart, there are two other interesting cases to note:

- non-disruptive rolling upgrades ... if a system is capable of operating without some of its nodes, it is possible to achieve non-disruptive rolling software upgrades. We take nodes down, one-at-a-time, upgrade each to a new software release, and then reintegrate them into the service. There are two tricks associated with this:
  - the new software must be up-wards compatible with the old software, so that new nodes can interoperate with old ones.
  - if the rolling upgrade does not seem to be working, there needs to be an automatic *fall-back* option to return to the previous (working) release.
  -
- prophylactic reboots ... It has long been observed that many software systems become slower and more error prone the longer they run. The most common problem is memory leaks, but there are other types of bugs that can cause software systems to degrade over time. The right solution is probably to find and fix the bugs ... but many organizations seem unable to do this. One popular alternative is to automatically restart every system at a regular interval (e.g. a few hours or days).

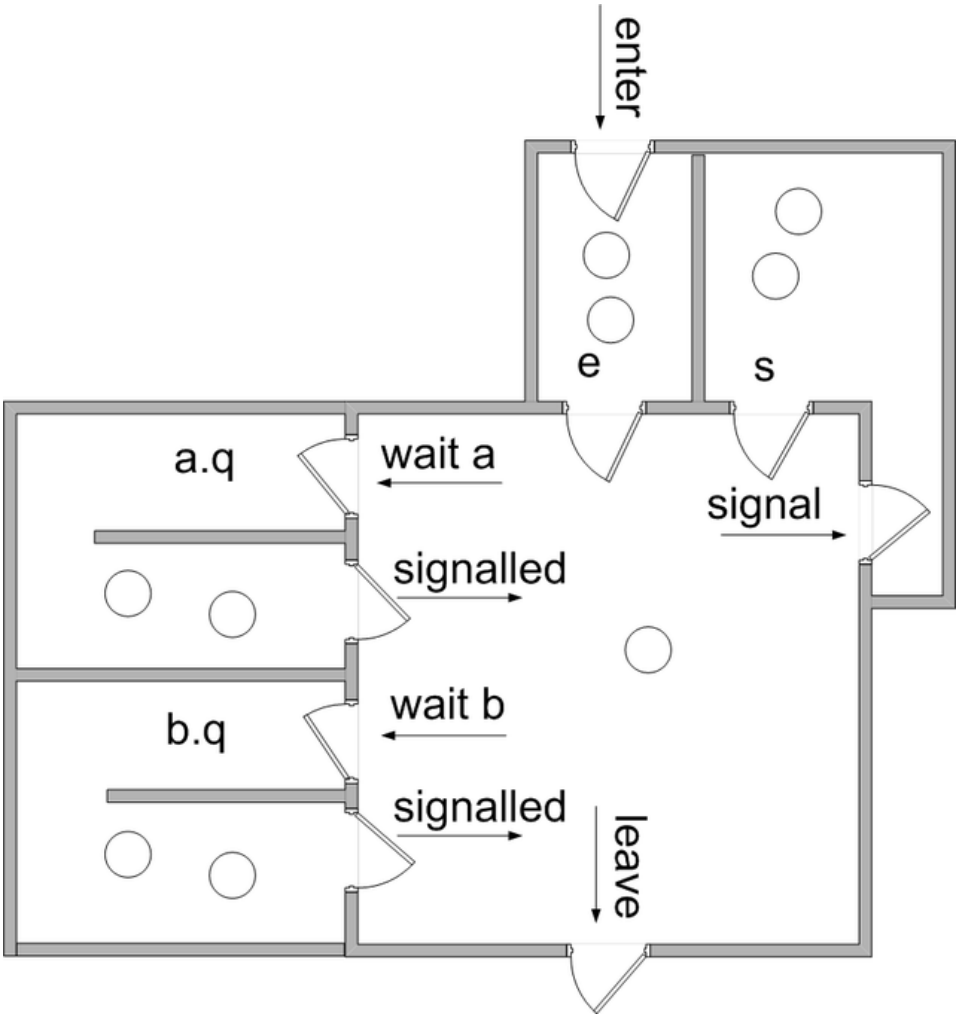
  If a system can continue operating in the face of node failures, it should be fairly simple to shut-down and restart nodes one at a time, on a regular schedule.

Monitor (synchronization)

Connected to:

Per Brinch Hansen Thread (computer science) Thread (computing)

From Wikipedia, the free encyclopedia



This article contains instructions, advice, or how-to content. The purpose of Wikipedia is to present facts, not to train. Please help improve this article either by rewriting the how-to content or by moving it to Wikiversity, Wikibooks or Wikivoyage. (January 2014)

In concurrent programming, a **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signalling other threads that their condition has been met. A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Another definition of **monitor** is a **thread-safe** class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in

time, at most one thread may be executing any of its methods. Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/class/module".

Monitors were invented by Per Brinch Hansen[1] and C. A. R. Hoare,[2] and were first implemented in Brinch Hansen's Concurrent Pascal language.[3]

Mutual exclusion

As a simple example, consider a thread-safe object for performing transactions on a bank account:

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    if balance < amount {
      return false
    } else {
      balance := balance - amount
      return true
    }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    balance := balance + amount
  }
}
```

While a thread is executing a method of a thread-safe object, it is said to *occupy* the object, by holding its mutex (lock). Thread-safe objects are implemented to enforce that *at each point in time, at most one thread may occupy the object*. The lock, which is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

Upon calling one of the methods, a thread must wait until no other thread is executing any of the thread-safe object's methods before starting execution of its method. Note that without this mutual exclusion, in the present example, two threads could cause money to be lost or gained for no reason. For example, two threads withdrawing 1000 from the account could both return true, while causing the balance to drop by only 1000, as follows: first, both threads fetch the current balance, find it greater than 1000, and subtract 1000 from it; then, both threads store the balance and return.

The syntactic sugar "monitor class" in the above example is implementing the following basic representation of the code, by wrapping each function's execution in mutexes:

```
class Account {
  private lock myLock

  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      if balance < amount {
        return false
      } else {
        balance := balance - amount
        return true
      }
    } finally {
      myLock.release()
    }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      balance := balance + amount
    } finally {
      myLock.release()
    }
  }
}
```

Condition variables

Problem statement

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition *P* holds true. A busy waiting loop

```
   while not( P ) do skip
```

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true. Other "solutions" exist such as having a loop that unlocks the monitor, waits a certain amount of time, locks the monitor and check for the condition *P*. Theoretically, it works and will not deadlock, but issues arise. It's hard to decide an appropriate amount of waiting time, too small and the thread will hog the CPU, too big and it will be apparently unresponsive. What is needed is a way to signal the thread when the condition P is true (or *could* be true).

Case study: classic bounded producer/consumer problem

A classic concurrency problem is that of the **bounded producer/consumer**, in which there is a queue or ring buffer of tasks with a maximum size, with one or more threads being "producer" threads that add tasks to the queue, and one or more other threads being "consumer" threads that take tasks out of the queue. The queue is assumed to be non–thread-safe itself, and it can be empty, full, or between empty and full. Whenever the queue is full of tasks, then we need the producer threads to block until there is room from consumer threads dequeueing tasks. On the other hand, whenever the queue is empty, then we need the consumer threads to block until more tasks are available due to producer threads adding them.

As the queue is a concurrent object shared between threads, accesses to it must be made atomic, because the queue can be put into an **inconsistent state** during the course of the queue access that should never be exposed between threads. Thus, any code that accesses the queue constitutes a **critical section** that must be synchronized by mutual exclusion. If code and processor instructions in critical sections of code that access the queue could be **interleaved** by arbitrary **context switches** between threads on the same processor or by simultaneously-running threads on multiple processors, then there is a risk of exposing inconsistent state and causing race conditions.

Incorrect without synchronization

A naïve approach is to design the code with **busy-waiting** and no synchronization, making the code subject to race conditions:

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.
        while(queue.isFull()){} // Busy-wait until the queue is non-full.
        queue.enqueue(myTask); // Add the task to the queue.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        while (queue.isEmpty()){} // Busy-wait until the queue is non-empty.
        myTask=queue.dequeue(); // Take a task off of the queue.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This code has a serious problem in that accesses to the queue can be interrupted and interleaved with other threads' accesses to the queue. The *queue.enqueue* and *queue.dequeue* methods likely have instructions to update the queue's member variables such as its size, beginning and ending positions, assignment and allocation of queue elements, etc. In addition, the *queue.isEmpty()* and *queue.isFull()* methods read this shared state as well. If producer/consumer threads are allowed to be interleaved during the calls to enqueue/dequeue, then inconsistent state of the queue can be exposed leading to race conditions. In addition, if one consumer makes the queue empty in-between another consumer's exiting the busy-wait and calling "dequeue", then the second consumer will attempt to dequeue from an empty queue leading to an error. Likewise, if a producer makes the queue full in-between another producer's exiting the busy-wait and calling "enqueue", then the second producer will attempt to add to a full queue leading to an error.

Spin-waiting

One naive approach to achieve synchronization, as alluded to above, is to use "**spin-waiting**", in which a mutex is used to protect the critical sections of code and busy-waiting is still used, with the lock being acquired and released in-between each busy-wait check.

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while(queue.isFull()){ // Busy-wait until the queue is non-full.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a consumer might take a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isFull()".
        }

        queue.enqueue(myTask); // Add the task to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
```

```
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while (queue.isEmpty()){ // Busy-wait until the queue is non-empty.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a producer might add a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isEmpty()".
        }
        myTask=queue.dequeue(); // Take a task off of the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This method assures that inconsistent state does not occur, but wastes CPU resources due to the unnecessary busy-waiting. Even if the queue is empty and producer threads have nothing to add for a long time, consumer threads are always busy-waiting unnecessarily. Likewise, even if consumers are blocked for a long time on processing their current tasks and the queue is full, producers are always busy-waiting. This is a wasteful mechanism. What is needed is a way to make producer threads block until the queue is non-full, and a way to make consumer threads block until the queue is non-empty.

(N.B.: Mutexes themselves can also be **spin-locks** which involve busy-waiting in order to get the lock, but in order to solve this problem of wasted CPU resources, we assume that *queueLock* is not a spin-lock and properly uses a blocking lock queue itself.)

Condition variables

The solution is to use **condition variables**. Conceptually a condition variable is a queue of threads, associated with a monitor, on which a thread may wait for some condition to become true. Thus each condition variable c is associated with an assertion P_c. While a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable c to indicate that assertion P_c is true in the current state.

Thus there are two main operations on condition variables:

- **`wait c, m`**, where c is a condition variable and `m` is a mutex (lock) associated with the monitor. This operation is called by a thread that needs to wait until the assertion P_c is true before proceeding. While the thread is waiting, it does not occupy the monitor. The function, and fundamental contract, of the "wait" operation, is to do the following steps:
    1. Atomically:

        a. release the mutex m,
        b. move this thread from the "ready queue" to c's "wait-queue" (a.k.a. "sleep-queue") of threads, and
        c. sleep this thread. (Context is synchronously yielded to another thread.)

    2. Once this thread is subsequently notified/signalled (see below) and resumed, then automatically re-acquire the mutex m.

    Steps 1a and 1b can occur in either order, with 1c usually occurring after them. While the thread is sleeping and in c's wait-queue, the next program counter to be executed is at step 2, in the middle of the "wait" function/subroutine. Thus, the thread sleeps and later wakes up in the middle of the "wait" operation.
    The atomicity of the operations within step 1 is important to avoid race conditions that would be caused by a preemptive thread switch in-between them. One failure mode that could occur if these were not atomic is a *missed wakeup*, in which the thread could be on c's sleep-queue and have released the mutex, but a preemptive thread switch occurred before the thread went to sleep, and another thread called a signal/notify operation (see below) on c moving the first thread back out of c's queue. As soon as the first thread in question is switched back to, its program counter will be at step 1c, and it will sleep and be unable to be woken up again, violating the invariant that it should have been on c's sleep-queue when it slept. Other race conditions depend on the ordering of steps 1a and 1b, and depend on where a context switch occurs.

- **`signal c`**, also known as **`notify c`**, is called by a thread to indicate that the assertion P_c is true. Depending on the type and implementation of the monitor, this moves one or more threads from `c`'s sleep-queue to the "ready queue" or another queues for it to be executed. It is usually considered a best practice to perform the "signal"/"notify" operation before releasing mutex m that is associated with c, but as long as the code is properly designed for concurrency and depending on the threading implementation, it is often also acceptable to release the lock before signalling. Depending on the threading implementation, the ordering of this can have scheduling-priority ramifications. (Some authors instead advocate a preference for releasing the lock before signalling.) A threading implementation should document any special constraints on this ordering.
    - **`broadcast c`**, also known as **`notifyAll c`**, is a similar operation that wakes up all threads in c's wait-queue. This empties the wait-queue. Generally, when more than one predicate condition is associated with the same condition variable, the application will require **broadcast** instead of **signal** because a thread waiting for the wrong condition might be woken up and then immediately go back to sleep without waking up a thread waiting for the correct condition that just became true. Otherwise, if the predicate condition is one-to-one with the condition variable associated with it, then **signal** may be more efficient than **broadcast**.

As a design rule, multiple condition variables can be associated with the same mutex, but not vice versa. (This is a one-to-many correspondence.) This is because the predicate P_c is the same for all threads using the monitor and must be protected with mutual exclusion from all other threads that might cause the condition to be changed or that might read it while the thread in question causes it to be changed, but there may be different threads that want to wait for a different condition on the same variable requiring the same mutex to be used. In the producer-consumer example described above, the queue must be protected by a unique mutex object, m. The "producer" threads will want to wait on a monitor using lock m and a condition variable c_{full} which blocks until the queue is non-full. The "consumer" threads will want to wait on a different monitor using the same mutex m but a different condition

variable c_{empty} which blocks until the queue is non-empty. It would (usually) never make sense to have different mutexes for the same condition variable, but this classic example shows why it often certainly makes sense to have multiple condition variables using the same mutex. A mutex used by one or more condition variables (one or more monitors) may also be shared with code that does *not* use condition variables (and which simply acquires/releases it without any wait/signal operations), if those critical sections do not happen to require waiting for a certain condition on the concurrent data.

Monitor usage

The proper basic usage of a monitor is:

```
acquire(m); // Acquire this monitor's lock.
while (!p) { // While the condition/predicate/assertion that we are waiting for is not true...
        wait(m, cv); // Wait on this monitor's lock and condition variable.
}
// ... Critical section of code goes here ...
signal(cv2); -- OR -- notifyAll(cv2); // cv2 might be the same as cv or different.
release(m); // Release this monitor's lock.
```

To be more precise, this is the same pseudocode but with more verbose comments to better explain what is going on:

```
// ... (previous code)
// About to enter the monitor.
// Acquire the advisory mutex (lock) associated with the concurrent data that is shared between threads,
// to ensure that no two threads can be preemptively interleaved or run simultaneously on different cores
// while executing in critical sections that read or write this same concurrent data.
// If another thread is holding this mutex, then this thread will be sleeped (blocked) and placed on
// m's sleep queue.  (Mutex "m" shall not be a spin-lock.)
acquire(m);
// Now, we are holding the lock and can check the condition for the first time.

// The first time we execute the while loop condition after the above "acquire", we are asking,
// "Does the condition/predicate/assertion we are waiting for happen to already be true?"

while ( ! p() ) // "p" is any expression (e.g. variable or function-call) that checks the condition
                                // and evaluates to boolean.  This itself is a critical section, so you *MUST*
                                // be holding the lock when executing this "while" loop condition!

// If this is not the first time the "while" condition is being checked, then we are asking the question,
// "Now that another thread using this monitor has notified me and woken me up and I have been
// context-switched back to, did the condition/predicate/assertion we are waiting on stay true between
// the time that I was woken up and the time that I
// re-acquired the lock inside the "wait" call in the last iteration of this loop,
// or did some other thread cause the condition to become false again in the meantime
// thus making this a spurious wakeup?

{
        // If this is the first iteration of the loop, then the answer is "no" -- the condition is not ready yet.
        // Otherwise, the answer is: the latter.  This was a spurious wakeup, some other thread occurred first
        // and caused the condition to become false again, and we must wait again.

        wait(m, cv);
                // Temporarily prevent any other thread on any core from doing operations on m or cv.
                // release(m)   // Atomically release lock "m" so other code using this concurrent data
                //                              // can operate, move this thread to cv's wait-queue so that it will be notified
                //                              // sometime when the condition becomes true, and sleep this thread.
                //                              // Re-enable other threads and cores to do operations on m and cv.
                //
                // Context switch occurs on this core.
                //
                // At some future time, the condition we are waiting for becomes true,
                // and another thread using this monitor (m, cv) does either a signal/notify
                // that happens to wake this thread up, or a notifyAll that wakes us up, meaning
                // that we have been taken out of cv's wait-queue.
                //
                // During this time, other threads may be switched to that caused the condition to become
                // false again, or the condition may toggle one or more times, or it may happen to
                // stay true.
                //
                // This thread is switched back to on some core.
                //
                // acquire(m)   // Lock "m" is re-acquired.

        // End this loop iteration and re-check the "while" loop condition to make sure the predicate is
        // still true.

}

// The condition we are waiting for is true!
// We are still holding the lock, either from before entering the monitor or from the
// last execution of "wait".

// Critical section of code goes here, which has a precondition that our predicate
// must be true.
// This code might make cv's condition false, and/or make other condition variables'
// predicates true.
```

```
// Call signal/notify or notifyAll, depending on which condition variables' predicates
// (who share mutex m) have been made true or may have been made true, and the monitor semantic type
// being used.

for (cv_x in cvs_to_notify){
        notify(cv_x); -- OR -- notifyAll(cv_x);
}
// One or more threads have been woken up but will block as soon as they try
// to acquire m.

// Release the mutex so that notified thread(s) and others can enter
// their critical sections.
release(m);
```

Solving the bounded producer/consumer problem

This section may be too technical for most readers to understand. Please help improve this section to make it understandable to non-experts, without removing the technical details. The talk page may contain suggestions. (January 2014) (Learn how and when to remove this template message)

Having introduced the usage of condition variables, let's use it to revisit and solve the classic bounded producer/consumer problem. The classic solution is to use two monitors, comprising two condition variables sharing one lock on the queue:

```
global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock;  // A mutex for the ring-buffer of tasks.  (Not a spin-lock.)
global CV queueEmptyCV; // A condition variable for consumer threads waiting for the queue to become non-empty.
                        // Its associated lock is "queueLock".
global CV queueFullCV; // A condition variable for producer threads waiting for the queue to become non-full.
                       //Its associated lock is also "queueLock".

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while(queue.isFull()){ // Check if the queue is non-full.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto queueFullCV, and sleep this thread.
            wait(queueLock, queueFullCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        queue.enqueue(myTask); // Add the task to the queue.

        // Now the queue is guaranteed to be non-empty, so signal a consumer thread
        // or all consumer threads that might be blocked waiting for the queue to be non-empty:
        signal(queueEmptyCV); -- OR -- notifyAll(queueEmptyCV);

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while (queue.isEmpty()){ // Check if the queue is non-empty.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto queueEmptyCV, and sleep this thread.
            wait(queueLock, queueEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }
        // Critical section that requires the queue to be non-empty.
        // N.B.: We are holding queueLock.
        myTask=queue.dequeue(); // Take a task off of the queue.
        // Now the queue is guaranteed to be non-full, so signal a producer thread
        // or all producer threads that might be blocked waiting for the queue to be non-full:
        signal(queueFullCV); -- OR -- notifyAll(queueFullCV);

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.

        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This ensures concurrency between the producer and consumer threads sharing the task queue, and blocks the threads that have nothing to do rather than busy-waiting as shown in the aforementioned approach using spin-locks.

A variant of this solution could use a single condition variable for both producers and consumers, perhaps named "queueFullOrEmptyCV" or "queueSizeChangedCV". In this case, more than one condition is associated with the condition variable, such that the condition variable represents a weaker condition than the conditions being checked by individual threads. The condition variable represents threads that are waiting for the queue to be non-full *and* ones waiting for it to be non-empty. However, doing this would require using *notifyAll* in all the threads using the condition variable and cannot use a regular *signal*. This is because the regular *signal* might wake up a thread of the wrong type whose condition has not yet been met, and that thread would go back to sleep without a thread of the correct type getting signalled. For example, a producer might make the queue full and wake up another producer instead of a consumer, and the woken producer would go back to sleep. In the complementary case, a consumer might make the queue empty and wake up another consumer instead of a producer, and the consumer would go back to sleep. Using *notifyAll* ensures that some thread of the right type will proceed as expected by the problem statement.

Here is the variant using only one condition variable and notifyAll:

```
global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.  (Not a spin-lock.)
global CV queueFullOrEmptyCV; // A single condition variable for when the queue is not ready for any thread
                              // -- i.e., for producer threads waiting for the queue to become non-full
                              // and consumer threads waiting for the queue to become non-empty.
                              // Its associated lock is "queueLock".
                              // Not safe to use regular "signal" because it is associated with
                              // multiple predicate conditions (assertions).

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while(queue.isFull()){ // Check if the queue is non-full.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto the CV, and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        queue.enqueue(myTask); // Add the task to the queue.

        // Now the queue is guaranteed to be non-empty, so signal all blocked threads
        // so that a consumer thread will take a task:
        notifyAll(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another producer instead).

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while (queue.isEmpty()){ // Check if the queue is non-empty.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto the CV, and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }
        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        myTask=queue.dequeue(); // Take a task off of the queue.

        // Now the queue is guaranteed to be non-full, so signal all blocked threads
        // so that a producer thread will take a task:
        notifyAll(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another consumer instead).

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.

        doStuff(myTask); // Go off and do something with the task.
    }
}
```

Synchronization primitives

Implementing mutexes and condition variables requires some kind of synchronization primitive provided by hardware support that provides atomicity. Locks and condition variables are higher-level abstractions over these synchronization primitives. On a uniprocessor, disabling and enabling interrupts is a way to implement monitors by preventing context switches during the critical sections of the locks and condition variables, but this is not enough on a multiprocessor. On a multiprocessor, usually special atomic **read-modify-write** instructions on the memory such as **test-and-set**, **compare-and-swap**, etc. are used, depending on what the ISA provides. These usually require deferring to spin-locking for the internal lock state itself, but this locking is very brief. Depending on the implementation, the atomic read-modify-write instructions may lock the bus from other cores' accesses and/or prevent re-

ordering of instructions in the CPU. Here is an example pseudocode implementation of parts of a threading system and mutexes and Mesa-style condition variables, using **test-and-set** and a first-come, first-served policy. This glosses over most of how a threading system works, but shows the parts relevant to mutexes and condition variables:

Sample Mesa-monitor implementation with Test-and-Set

This section may be too technical for most readers to understand. Please help improve this section to make it understandable to non-experts, without removing the technical details. The talk page may contain suggestions. (January 2014) (Learn how and when to remove this template message)

```
// Basic parts of threading system:
// Assume "ThreadQueue" supports random access.
public volatile ThreadQueue readyQueue; // Thread-unsafe queue of ready threads.  Elements are (Thread*).
public volatile global Thread* currentThread; // Assume this variable is per-core.  (Others are shared.)

// Implements a spin-lock on just the synchronized state of the threading system itself.
// This is used with test-and-set as the synchronization primitive.
public volatile global bool threadingSystemBusy=false;

// Context-switch interrupt service routine (ISR):
// On the current CPU core, preemptively switch to another thread.
public method contextSwitchISR(){
    if (testAndSet(threadingSystemBusy)){
        return; // Can't switch context right now.
    }

    // Ensure this interrupt can't happen again which would foul up the context switch:
    systemCall_disableInterrupts();

    // Get all of the registers of the currently-running process.
    // For Program Counter (PC), we will need the instruction location of
    // the "resume" label below.  Getting the register values is platform-dependent and may involve
    // reading the current stack frame, JMP/CALL instructions, etc.  (The details are beyond this scope.)
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.

    readyQueue.enqueue(currentThread); // Put this thread back onto the ready queue for later execution.

    Thread* otherThread=readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.

    currentThread=otherThread; // Replace the global current-thread pointer value so it is ready for the next thread.

    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread
    // (at "resume" below).  Again, the details of how this is done are beyond this scope.
    restoreRegisters(otherThread.registers);

    // *** Now running "otherThread" (which is now "currentThread")!  The original thread is now "sleeping". ***

    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.

    // Return to where otherThread left off.

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

}

// Thread sleep method:
// On current CPU core, a synchronous context switch to another thread without putting
// the current thread on the ready queue.
// Must be holding "threadingSystemBusy" and disabled interrupts so that this method
// doesn't get interrupted by the thread-switching timer which would call contextSwitchISR().
// After returning from this method, must clear "threadingSystemBusy".
public method threadSleep(){
    // Get all of the registers of the currently-running process.
    // For Program Counter (PC), we will need the instruction location of
    // the "resume" label below.  Getting the register values is platform-dependent and may involve
    // reading the current stack frame, JMP/CALL instructions, etc.  (The details are beyond this scope.)
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.

    // Unlike contextSwitchISR(), we will not place currentThread back into readyQueue.
    // Instead, it has already been placed onto a mutex's or condition variable's queue.

    Thread* otherThread=readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.

    currentThread=otherThread; // Replace the global current-thread pointer value so it is ready for the next thread.

    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread
    // (at "resume" below).  Again, the details of how this is done are beyond this scope.
    restoreRegisters(otherThread.registers);

    // *** Now running "otherThread" (which is now "currentThread")!  The original thread is now "sleeping". ***

    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.

    // Return to where otherThread left off.
```

```
}

public method wait(Mutex m, ConditionVariable c){
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    assert m.held; // (Specifically, this thread must be the one holding it.)

    m.release();
    c.waitingThreads.enqueue(currentThread);

    threadSleep();

    // Thread sleeps ... Thread gets woken up from a signal/broadcast.

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // Context switches may now occur here, making the client caller's predicate false.

    m.acquire();

}

public method signal(ConditionVariable c){

    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    if (!c.waitingThreads.isEmpty()){
        wokenThread=c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken thread is not given any priority.

}

public method broadcast(ConditionVariable c){

    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    while (!c.waitingThreads.isEmpty()){
        wokenThread=c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken threads are not given any priority.

}

class Mutex {
    protected volatile bool held=false;
```

```
    private volatile ThreadQueue blockingThreads; // Thread-unsafe queue of blocked threads.  Elements are (Thread*).

    public method acquire(){
        // Internal spin-lock while other threads on any core are accessing this object's
        // "held" and "threadQueue", or "readyQueue".
        while (testAndSet(threadingSystemBusy)){}
        // N.B.: "threadingSystemBusy" is now true.

        // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
        // the thread-switching timer on this core which would call contextSwitchISR().
        // Done outside threadSleep() for more efficiency so that this thread will be sleeped
        // right after going on the lock queue.
        systemCall_disableInterrupts();

        assert !blockingThreads.contains(currentThread);

        if (held){
            // Put "currentThread" on this lock's queue so that it will be
            // considered "sleeping" on this lock.
            // Note that "currentThread" still needs to be handled by threadSleep().
            readyQueue.remove(currentThread);
            blockingThreads.enqueue(currentThread);
            threadSleep();

            // Now we are woken up, which must be because "held" became false.
            assert !held;
            assert !blockingThreads.contains(currentThread);
        }

        held=true;

        threadingSystemBusy=false; // Must be an atomic assignment.
        systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    }

    public method release(){
        // Internal spin-lock while other threads on any core are accessing this object's
        // "held" and "threadQueue", or "readyQueue".
        while (testAndSet(threadingSystemBusy)){}
        // N.B.: "threadingSystemBusy" is now true.

        // System call to disable interrupts on this core for efficiency.
        systemCall_disableInterrupts();

        assert held; // (Release should only be performed while the lock is held.)

        held=false;

        if (!blockingThreads.isEmpty()){
            Thread* unblockedThread=blockingThreads.dequeue();
            readyQueue.enqueue(unblockedThread);
        }

        threadingSystemBusy=false; // Must be an atomic assignment.
        systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    }

}

struct ConditionVariable {

    volatile ThreadQueue waitingThreads;

}
```

Semaphore

As an example, consider a thread-safe class that implements a semaphore. There are methods to increment (V) and to decrement (P) a private integer `s`. However, the integer must never be decremented below 0; thus a thread that tries to decrement must wait until the integer is positive. We use a condition variable `sIsPositive` with an associated assertion of $P_{sIsPositive} = (s > 0)$.

```
monitor class Semaphore
{
  private int s := 0
  invariant s >= 0
  private Condition sIsPositive /* associated with s > 0 */

  public method P()
  {
    while s = 0:
      wait sIsPositive
    assert s > 0
    s := s - 1
  }
```

```
  public method V()
  {
    s := s + 1
    assert s > 0
    signal sIsPositive
  }
}
```

Implemented showing all synchronization (removing the assumption of a thread-safe class and showing the mutex):

```
class Semaphore
{
  private volatile int s := 0
  invariant s >= 0
  private ConditionVariable sIsPositive /* associated with s > 0 */
  private Mutex myLock /* Lock on "s" */

  public method P()
  {
    myLock.acquire()
    while s = 0:
      wait(myLock, sIsPositive)
    assert s > 0
    s := s - 1
    myLock.release()
  }

  public method V()
  {
    myLock.acquire()
    s := s + 1
    assert s > 0
    signal sIsPositive
    myLock.release()
  }
}
```

Monitor implemented using semaphores

Conversely, locks and condition variables can also be derived from semaphores, thus making monitors and semaphores reducible to one another:

The implementation given here is incorrect. If a thread calls wait() after signal() has been called it may be stuck indefinitely, since signal() increments the semaphore only enough times for threads already waiting.

```
public method wait(Mutex m, ConditionVariable c){

    assert m.held;

    c.internalMutex.acquire();

    c.numWaiters++;
    m.release(); // Can go before/after the neighboring lines.
    c.internalMutex.release();

    // Another thread could signal here, but that's OK because of how
    // semaphores count.  If c.sem's number becomes 1, we'll have no
    // waiting time.
    c.sem.Proberen(); // Block on the CV.
    // Woken
    m.acquire(); // Re-acquire the mutex.
}

public method signal(ConditionVariable c){

    c.internalMutex.acquire();
    if (c.numWaiters > 0){
        c.numWaiters--;
        c.sem.Verhogen(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();

}

public method broadcast(ConditionVariable c){

    c.internalMutex.acquire();
    while (c.numWaiters > 0){
        c.numWaiters--;
        c.sem.Verhogen(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();

}
```

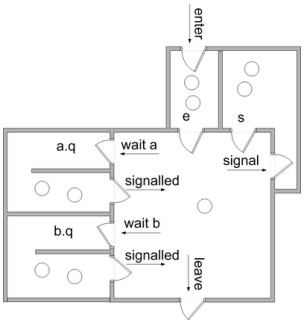
```
class Mutex {

    protected boolean held=false; // For assertions only, to make sure sem's number never goes > 1.
    protected Semaphore sem=Semaphore(1); // The number shall always be at most 1.
                                         // Not held <--> 1; held <--> 0.

    public method acquire(){

        sem.Proberen();
        assert !held;
        held=true;

    }

    public method release(){

        assert held; // Make sure we never Verhogen sem above 1.  That would be bad.
        held=false;
        sem.Verhogen();

    }

}

class ConditionVariable {

    protected int numWaiters=0; // Roughly tracks the number of waiters blocked in sem.
                                // (The semaphore's internal state is necessarily private.)
    protected Semaphore sem=Semaphore(0); // Provides the wait queue.
    protected Mutex internalMutex; // (Really another Semaphore.  Protects "numWaiters".)

}
```

When a **signal** happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. In order that at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

- *Blocking condition variables* give priority to a signaled thread.
- *Nonblocking condition variables* give priority to the signaling thread.

Blocking condition variables

The original proposals by C. A. R. Hoare and Per Brinch Hansen were for *blocking condition variables*. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition variable. Monitors using blocking condition variables are often called *Hoare-style* monitors or *signal-and-urgent-wait* monitors.



A Hoare style monitor with two condition variables `a` and `b`. After Buhr *et al*.

We assume there are two queues of threads associated with each monitor object

- `e` is the entrance queue
- `s` is a queue of threads that have signaled.

In addition we assume that for each condition variable *c*, there is a queue

- *c*`.q`, which is a queue for threads waiting on condition variable *c*

All queues are typically guaranteed to be fair and, in some implementations, may be guaranteed to be first in first out.

The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```
 enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
```

```
   else
       lock the monitor

leave the monitor:
   schedule
   return from the method

wait c :
   add this thread to c.q
   schedule
   block this thread

signal c :
   if there is a thread waiting on c.q
       select and remove one such thread t from c.q
       (t is called "the signaled thread")
       add this thread to s
       restart t
       (so t will occupy the monitor next)
       block this thread

 schedule :
   if there is a thread on s
     select and remove one thread from s and restart it
     (this thread will occupy the monitor next)
   else if there is a thread on e
     select and remove one thread from e and restart it
     (this thread will occupy the monitor next)
   else
     unlock the monitor
     (the monitor will become unoccupied)
```

The `schedule` routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor.

The resulting signaling discipline is known a *"signal and urgent wait,"* as the signaler must wait, but is given priority over threads on the entrance queue. An alternative is *"signal and wait,"* in which there is no `s` queue and signaler waits on the `e` queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

```
signal c and return :
   if there is a thread waiting on c.q
       select and remove one such thread t from c.q
       (t is called "the signaled thread")
       restart t
       (so t will occupy the monitor next)
   else
       schedule
   return from the method
```

In either case ("signal and urgent wait" or "signal and wait"), when a condition variable is signaled and there is at least one thread on waiting on the condition variable, the signaling thread hands occupancy over to the signaled thread seamlessly, so that no other thread can gain occupancy in between. If P_c is true at the start of each **signal** *c* operation, it will be true at the end of each **wait** *c* operation. This is summarized by the following contracts. In these contracts, I is the monitor's invariant.

```
enter the monitor:
   postcondition I

leave the monitor:
   precondition I

wait c :
   precondition I
   modifies the state of the monitor
   postcondition P_c and I

signal c :
   precondition P_c and I
   modifies the state of the monitor
   postcondition I

signal c and return :
   precondition P_c and I
```

In these contracts, it is assumed that I and P_c do not depend on the contents or lengths of any queues.

(When the condition variable can be queried as to the number of threads waiting on its queue, more sophisticated contracts can be given. For example, a useful pair of contracts, allowing occupancy to be passed without establishing the invariant, is

```
wait c :
   precondition I
   modifies the state of the monitor
   postcondition P_c

signal c
   precondition (not empty(c) and P_c) or (empty(c) and I)
```

```
    modifies the state of the monitor
    postcondition I
```

See Howard[4] and Buhr *et al.*,[5] for more).

It is important to note here that the assertion P_c is entirely up to the programmer; he or she simply needs to be consistent about what it is.

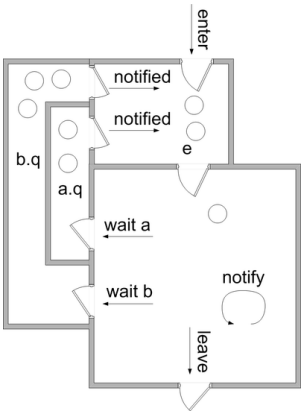
We conclude this section with an example of a thread-safe class using a blocking monitor that implements a bounded, thread-safe stack.

```
monitor class SharedStack {
  private const capacity := 10
  private int[capacity] A
  private int size := 0
  invariant 0 <= size and size <= capacity
  private BlockingCondition theStackIsNotEmpty /* associated with 0 < size and size <= capacity */
  private BlockingCondition theStackIsNotFull  /* associated with 0 <= size and size < capacity */

  public method push(int value)
  {
    if size = capacity then wait theStackIsNotFull
    assert 0 <= size and size < capacity
    A[size] := value ; size := size + 1
    assert 0 < size and size <= capacity
    signal theStackIsNotEmpty and return
  }

  public method int pop()
  {
    if size = 0 then wait theStackIsNotEmpty
    assert 0 < size and size <= capacity
    size := size - 1 ;
    assert 0 <= size and size < capacity
    signal theStackIsNotFull  and return A[size]
  }
}
```

Note that, in this example, the thread-safe stack is internally providing a mutex, which, as in the earlier producer/consumer example, is shared by both condition variables, which are checking different conditions on the same concurrent data. The only difference is that the producer/consumer example assumed a regular non-thread-safe queue and was using a standalone mutex and condition variables, without these details of the monitor abstracted away as is the case here. In this example, when the "wait" operation is called, it must somehow be supplied with the thread-safe stack's mutex, such as if the "wait" operation is an integrated part of the "monitor class". Aside from this kind of abstracted functionality, when a "raw" monitor is used, it will *always* have to include a mutex and a condition variable, with a unique mutex for each condition variable.

Nonblocking condition variables

With *nonblocking condition variables* (also called "*Mesa style*" condition variables or "*signal and continue*" condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the `e` queue. There is no need for the `s` queue.



A Mesa style monitor with two condition variables `a` and `b`

With nonblocking condition variables, the **signal** operation is often called **notify** — a terminology we will follow here. It is also common to provide a **notify all** operation that moves all threads waiting on a condition variable to the `e` queue.

The meaning of various operations are given here. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```
 enter the monitor:
    enter the method
    if the monitor is locked
      add this thread to e
      block this thread
    else
      lock the monitor
```

```
 leave the monitor:
    schedule
    return from the method

 wait c :
    add this thread to c.q
    schedule
    block this thread

 notify c :
    if there is a thread waiting on c.q
        select and remove one thread t from c.q
        (t is called "the notified thread")
        move t to e

 notify all c :
    move all threads waiting on c.q to e

  schedule :
    if there is a thread on e
      select and remove one thread from e and restart it
    else
      unlock the monitor
```

As a variation on this scheme, the notified thread may be moved to a queue called `w`, which has priority over `e`. See Howard[4] and Buhr *et al*.[5] for further discussion.

It is possible to associate an assertion P_c with each condition variable c such that P_c is sure to be true upon return from **`wait c`**. However, one must ensure that P_c is preserved from the time the **notify**ing thread gives up occupancy until the notified thread is selected to re-enter the monitor. Between these times there could be activity by other occupants. Thus it is common for P_c to simply be *true*.

For this reason, it is usually necessary to enclose each **wait** operation in a loop like this

```
  while not( P ) do wait c
```

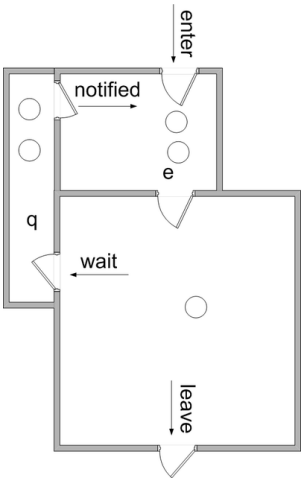
where P is some condition stronger than P_c. The operations **`notify c`** and **`notify all c`** are treated as "hints" that P may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with nonblocking monitors, one must be careful to ensure that too many notifications can not be lost.

As an example of "hinting" consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0
  private NonblockingCondition balanceMayBeBigEnough

  public method withdraw(int amount)
     precondition amount >= 0
  {
    while balance < amount do wait balanceMayBeBigEnough
    assert balance >= amount
    balance := balance - amount
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    balance := balance + amount
    notify all balanceMayBeBigEnough
  }
}
```

In this example, the condition being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to *know* that it made such a condition true. It makes sense in this case to allow each waiting thread into the monitor (one at a time) to check if its assertion is true.

Implicit condition variable monitors



A Java style monitor

In the Java language, each object may be used as a monitor. Methods requiring mutual exclusion must be explicitly marked with the **synchronized** keyword. Blocks of code may also be marked by **synchronized**.

Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a single wait queue in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notifyAll** operations apply to this queue. This approach has been adopted in other languages, for example C#.

Implicit signaling

Another approach to signaling is to omit the **signal** operation. Whenever a thread leaves the monitor (by returning or waiting) the assertions of all waiting threads are evaluated until one is found to be true. In such a system, condition variables are not needed, but the assertions must be explicitly coded. The contract for wait is

```
wait P:
   precondition I
   modifies the state of the monitor
   postcondition P and I
```

History

Brinch Hansen and Hoare developed the monitor concept in the early 1970s, based on earlier ideas of their own and of Edsger Dijkstra.[6] Brinch Hansen published the first monitor notation, adopting the class concept of Simula 67,[1] and invented a queueing mechanism.[7] Hoare refined the rules of process resumption.[2] Brinch Hansen created the first implementation of monitors, in Concurrent Pascal.[6] Hoare demonstrated their equivalence to semaphores.

Monitors (and Concurrent Pascal) were soon used to structure process synchronization in the Solo operating system.[8][9]

Programming languages that have supported monitors include

- Ada since Ada 95 (as protected objects)
- C# (and other languages that use the .NET Framework)
- C++ since C++11
- Concurrent Euclid
- Concurrent Pascal
- D
- Delphi (Delphi 2009 and above, via TObject.Monitor)
- Java (via the wait and notify methods)
- Mesa
- Modula-3
- Python (via threading.Condition object)
- Ruby
- Squeak Smalltalk
- Turing, Turing+, and Object-Oriented Turing
- μC++

A number of libraries have been written that allow monitors to be constructed in languages that do not support them natively. When library calls are used, it is up to the programmer to explicitly mark the start and end of code executed with mutual exclusion. Pthreads is one such library.

See also

- Mutual exclusion
- Communicating sequential processes - a later development of monitors by C. A. R. Hoare

The Java™ Tutorials

Trail: Essential Classes
Lesson: Concurrency
Section: Synchronization

Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through `synchronized` methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

Previous page: Memory Consistency Errors
Next page: Intrinsic Locks and Synchronization

The Java™ Tutorials

Trail: Essential Classes
Lesson: Concurrency
Section: Synchronization

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
```

```
            synchronized(lock1) {
                c1++;
            }
        }

        public void inc2() {
            synchronized(lock2) {
                c2++;
            }
        }
    }
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Previous page: Synchronized Methods
Next page: Atomic Access