14. Practical Concurrency

We begin our study of concurrency by describing how to use it in practice; later, in handout 17 on formal concurrency, we shall study it more formally. First we explain where the concurrency in a system comes from, and discuss the main ways to express concurrency. Then we describe the difference between 'hard' and 'easy' concurrency[1]: the latter is done by locking shared data before you touch it, the former in subtle ways that are so error-prone that simple prudence requires correctness proofs. We give the rules for easy concurrency using locks, and discuss various issues that complicate the easy life: scheduling, locking granularity, and deadlocks.

Sources of concurrency

Before studying concurrency in detail, it seems useful to consider how you might get concurrency in your system. Obviously if you have a multiprocessor or a distributed system you will have concurrency, since in these systems there is more than one CPU executing instructions. Similarly, most hardware has separate parts that can change state simultaneously and independently. But suppose your system consists of a single CPU running a program. Then you can certainly arrange for concurrency by multiplexing that CPU among several tasks, but why would you want to do this? Since the CPU can only execute one instruction at a time, it isn't entirely obvious that there is any advantage to concurrency. Why not get one task done before moving on to the next one?

There are only two possible reasons:

1. A task might have to wait for something else to complete before it can proceed, for instance for a disk read. But this means that there is some concurrent task that is going to complete, in the example an I/O device, the disk. So we have concurrency in any system that has I/O, even when there is only one CPU.

2. Something else might have to wait for the result of one task but not for the rest of the computation, for example a human user. But this means that there is some concurrent task that is waiting, in the example the user. Again we have concurrency in any system that has I/O.

In the first case one task must wait for I/O, and we can get more work done by running another task on the CPU, rather than letting it idle during the wait. Thus the concurrency of the I/O system leads to concurrency on the CPU. If the I/O wait is explicit in the program, the programmer can know when other tasks might run; this is often called a 'non-preemptive' system, because it has sequential semantics except when the program explicitly allows concurrent activity by waiting. But if the I/O is done at some low level of abstraction, higher levels may be quite unaware of it. The most insidious example of this is I/O caused by the virtual memory system: every instruction can cause a disk read. Such a system is called 'preemptive'; for practical purposes a task can lose the CPU at any point, since it's too hard to predict which memory references might cause page faults.[2]

[1] I am indebted to Greg Nelson for this taxonomy, and for the object and set example of deadlock avoidance.
[2] Of course, if the system just waits for the page fault to complete, rather than running some other task, then the page fault is harmless.

In the second case we have a motivation for true preemption: we want some tasks to have higher priority for the CPU than others. An important special case is interrupts, discussed below.

A concurrent program is harder to write than a sequential program, since there are many more possible paths of execution and interactions among the parts of the program. The canonical example is two concurrent executions of

```
x := x + 1
```

Since this command is not atomic (either in Spec, or in C on most computers), `x` can end up with either `1` or `2`, depending on the order of execution of the expression evaluations and the assignments. The interleaved order

```
evaluate x + 1
evaluate x + 1
x := result
x := result
```

leaves `x = 1`, while doing both steps of one command before either step of the other leaves `x = 2`. This is called a *race*, because the two threads are racing each other to get `x` updated.

Since concurrent programs are harder to understand, it's best to avoid concurrency unless you really needed it for one of the reasons just discussed.[3] Unfortunately, it will very soon be the case that every PC is a multi-processor, since the only way to make productive use of all the transistors that Moore's law is giving us, without making drastic changes to the programming model, is to put several processors on each chip. It will be interesting to see what people do with all this concurrency.

One good thing about concurrency, on the other hand, is that when you write a program as a set of concurrent computations, you can defer decisions about exactly how to schedule them. More generally, concurrency can be an attractive way to decompose a large system: put different parts of it into different tasks, and carefully control their communication. We shall see some effective, if constraining, ways to do this.

We saw that in the absence of a multi-processor, the only reason for concurrency in your program is that there's something concurrent going on outside the program, usually an I/O operation. This external, or *heterogeneous* concurrency doesn't usually give rise to bugs by itself, since the concurrently running CPU and I/O device don't share any state directly and only interact by exchanging messages (though DMA I/O, when imprudently used, can make this false). We will concern ourselves from now on with *homogeneous* concurrency, where several concurrent computations are sharing the same memory.

Ways to package concurrency

In the last section we used the word 'task' informally to describe a more-or-less independent, more-or-less sequential part of a computation. Now we shall be less coy about how concurrency shows up in a system.

The most general way to describe a concurrent system is in terms of a set of atomic actions with the property that usually more than one of them can occur (is enabled); we will use this viewpoint in our later study of formal concurrency. In practice, however, we usually think in

[3] This is the main reason why threads with RPC or synchronous messages are good, and asynchronous messages are bad. The latter force you to have concurrency whenever you have communication, while the former let you put in the concurrency just where you really need it. Of course if the implementation of threads is clumsy or expensive, as it often is, that may overwhelm the inherent advantages.

terms of several 'threads' of concurrent execution. Within a single thread at most one action is enabled at a time; in general one action may be enabled from each thread, though often some of the threads are waiting or 'blocked', that is, have no enabled actions.

The most convenient way to do concurrent programming is in a system that allows each thread to be described as an execution path in an ordinary-looking program with modules, routines, commands, etc., such as Spec, C, or Java. In this scheme more than one thread can execute the code of the same procedure; threads have local state that is the local variables of the procedures they are executing. All the languages mentioned and many others allow you to program in this way.

In fault-tolerant systems there is a conceptual drawback to this thread model. If a failure can occur after each atomic command, it is hard to understand the program by following the sequential flow of control in a thread, because there are so many other paths that result from failure and recovery. In these systems it is often best to reason strictly in terms of independent atomic actions. We will see detailed examples of this when we study reliable messages, consensus, and replication. Applications programmed in a transaction system are another example of this approach: each application runs in response to some input and is a single atomic action.

The biggest drawback of this kind of 'official' thread, however, is the costs of representing the local state and call stack of each thread and of a general mechanism for scheduling the threads. There are several alternatives that reduce these costs: interrupts, control blocks, and SIMD computers. They are all based on restricting the freedom of a thread to block, that is, to yield the processor until some external condition is satisfied, for example, until there is space in a buffer or a lock is free, or a page fault has been processed.

Interrupts

An interrupt routine is not the same as a thread, because:

- It always starts at the same point.
- It cannot wait for another thread.

The reason for these restrictions is that the execution context for an interrupt routine is allocated on someone else's stack, which means that the routine must complete before the thread that it interrupted can continue to run. On the other hand, the hardware that schedules an interrupt routine is efficient and takes account of priority within certain limits. In addition, the interrupt routine doesn't pay the cost of its own stack like an ordinary thread.

It's possible to have a hybrid system in which an interrupt routine that needs to wait turns itself into an ordinary thread by copying its state. This is tricky if the wait happens in a subroutine of the main interrupt routine, since the relevant state may be spread across several stack frames. If the copying doesn't happen too often, the interrupt-thread hybrid is efficient. The main drawbacks are that the copying usually has to be done by hand, which is error-prone, and that without compiler and runtime support it's not possible to reconstruct the call stack, which means that the thread has to be structured differently from the interrupt routine.

A simpler strategy that is widely used is to limit the work in the interrupt routine to simple things that don't require waits, and to wake up a separate thread to do anything more complicated.

Control blocks and message queues

Another, related strategy is to package all the permanent state of a thread, including its program counter, in a record (usually called a 'control block') and to explicitly schedule the execution of the threads. When a thread runs, it starts at the saved program counter (usually a procedure entry point) and runs until it explicitly gives up control or 'yields'. During execution it can call procedures, but when it yields its stack must be empty so that there's no need to save it, because all the state has to be in the control block. When it yields, a reference to the control block is saved where some other thread or interrupt routine can find it and queue the thread for execution when it's ready to run, for instance after an I/O operation is complete.[4]

The advantages of this approach are similar to those of interrupts: there are no stacks to manage, and scheduling can be carefully tuned to the application. The main drawback is also similar: a thread must unwind its stack before it can wait. In particular, it cannot wait to acquire a lock at an arbitrary point in the program.

It is very common to code the I/O system of an operating system using this kind of thread. Most people who are used to this style do not realize that it is a restricted, though efficient, case of general programming with threads.

In 'active messages', a recent variant of this scheme, you break your computation down into non-blocking segments; as the end of a segment, you package the state into an 'active message' and send it to the agent that can take the next step. Incoming messages are queued until the receiver has finished processing earlier ones.[5]

There are lots of other ways to use the control block idea. In 'scheduler activations', for example, kernel operations are defined so that they always run to completion; if an operation can't do what was requested, it returns intermediate state and can be retried later.[6] In 'message queuing' systems, the record of the thread state is stored in a persistent queue whenever it moves from one module to another, and a transaction is used to take the state off one queue, do some processing, and put it back onto another queue. This means that the thread can continue execution in spite of failures in machines or communication links.[7]

SIMD or data-parallel computing

This acronym stands for 'single instruction, multiple data', and refers to processors in which several execution units all execute the same sequence of instructions on different data values. In a 'pure' SIMD machine every instruction is executed at the same time by all the processors (except that some of them might be disabled for that instruction). Each processor has its own memory, and the processors can exchange data as part of an instruction. A few such machines were built between 1970 and 1993, but they are now out of favor.[8] The same programming paradigm is still used in many scientific problems however, at a coarser grain, and is called

[4] H. Lauer and R. Needham. On the duality of operating system structures. *Second Int. Symposium on Operating Systems*, IRIA, Rocquencourt, France, Oct. 1978 (reprinted in *Operating Systems Review* **13**,*2* (April 1979), 3-19).
[5] T. von Eiken et al., Active messages: A mechanism for integrated communication and computation. *Proc. International Symposium on Computer Architecture*, May 1992, pp 256-267.
[6] T. Anderson et al., Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer systems* **10**, 1 (Feb. 1992), pp 54-79.
[7] See www.messageq.com or A. Dickman, *Designing Applications With Msmq: Message Queuing for Developers*, Addison-Wesley, 1998.
[8] The term 'SIMD' has been recycled in the Intel MMX instruction set, and similar designs from several other manufacturers, to describe something much more prosaic: doing 8 8-bit adds in parallel on a 64-bit data path.

'data-parallel' computing. In one step each processor does some computation on its private data. When all of them are done, they exchange some data and then take the next step. The action of detecting that all are done is called 'barrier synchronization'.

Streams and vectors

Another way to package concurrency is using vectors or streams of data, sequences (or sets) of data items that are all processed in a similar way. This has the advantage that a single action can launch a lot of computation; for example, to add the 1000-item vectors a_i and b_i. Streams can come from graphics or database applications as well as from scientific computing.

Some simple examples of concurrency

Here are a number of examples of concurrency. You can think of these as patterns that might apply to part of your problem

Vector and matrix computations have a lot of inherent concurrency, since the operations on distinct indices are usually independent. Adding two vectors can use as much concurrency as there are vector elements (except for the overhead of scheduling the independent additions). More generally, any scheme that *partitions* data so that at most one thread acts on a single partition makes concurrency easy. Computing the scalar product of two vectors is trickier, since the result is a single sum, but the multiplies can be done in parallel, and the sum can be computed in a tree. This is a special case of a *combining tree*, in which siblings only interact at their parent node. A much more complex example is a file system that supports map-reduce.[9]

Histograms, where you map each item in some set into a bucket, and count the number of items in each bucket. Like scalar product, this has a fully partitioned portion where you do the mapping, and a shared-data portion where you increment the counts. There's more chance for concurrency without interference because the counts for different buckets are partitioned.

Divide and conquer programs usually have lots of concurrency. Consider the Fibonacci function, for example, defined by

```
FUNC Fib(n) -> Int = RET (n < 2 => 1 [*] Fib(n-1) + Fib(n-2))
```

The two recursive calls of `Fib` can run concurrently. Unfortunately, there will be a lot of duplicated work; this can be avoided by caching the results of sub-computations, but the cache must be accessed concurrently by the otherwise independent sub-computations. Both the basic idea and the problem of duplicated work generalize to a wide range of functional programs. Whether it works for programs with state depends on how much interaction there is among the divisions.

Read-compute-write computations can run at the speed of the slowest part, rather than of the sum. This is a special case of pipelining, discussed in more detail later. Ideally the load will be balanced so that each of the three phases takes the same amount of time. This organization usually doesn't improve latency, since a given work item has to move through all three phases, but it does improve bandwidth by the amount of concurrency.

Background or speculative computations, such as garbage collection, spell checking, prefetching data, synchronizing replicas of data, ripping CDs, etc.

[9] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *OSDI 2004*.

Easy concurrency

Concurrency is easy when you program with locks. The rules are simple:

- Every shared variable must be protected by a lock. A variable is shared if it is touched by more than one thread. Alternatively, you can say that *every* variable must be protected b y a lock, and think of data that is private to a thread as being protected by an implicit lock that is always held by the thread.
- You must hold the lock for a shared variable before you touch the variable. The essential property of a lock is that two threads can't hold the same lock at the same time. This property is called 'mutual exclusion'; the abbreviation 'mutex' is another name for a lock.
- If you want an atomic operation on several shared variables that are protected by different locks, you must not release any locks until you are done. This is called 'two-phase locking', because there is a phase in which you only acquire locks and don't release any, followed by a phase in which you only release locks and don't acquire any.

Then your computation between the point that you acquire a lock and the point that you release it is equivalent to a single atomic action, and therefore you can reason about it sequentially. This atomic part of the computation is called a 'critical section'. To use this method reliably, you should annotate each shared variable with the name of the lock that protects it, and clearly bracket the regions of your program within which you hold each lock. Then it is a mechanical process to check that you hold the proper lock whenever you touch a shared variable.[10] It's also possible to check a running program for violations of this discipline.[11]

Why do locks lead to big atomic actions? Intuitively, the reason is that no other well-behaved thread can touch any shared variable while you hold its lock, because a well-behaved thread won't touch a shared variable without itself holding its lock, and only one thread can hold a lock at a time. We will make this more precise in handout 17 on formal concurrency, and give a proof of atomicity. Another way of saying this is that locking ensures that concurrent operations *commute*. Concurrency means that we aren't sure what order they will run in, but commuting says that the order doesn't matter because the result is the same in either order.

Actually locks give you a bit more atomicity than this. If a well-behaved thread acquires a sequence of locks (acquiring each one before touching the data it protects) and then releases them (not necessarily in the same order, but releasing each one after touching the data it protects), the entire computation from the first acquire to the last release is atomic. Once you have done a release, however, you can't do another acquire without losing atomicity. This is called *two-phase locking*.

The simple locks we have been describing are also called 'mutexes'; this is short for "mutual exclusion". As we shall see, more complicated kinds of locks are often useful.

Here is the spec for a mutex. It maintains mutual exclusion by allowing the mutex to be acquired only when no one already holds it. If a thread other than the current holder releases the mutex, the result is undefined. If you try to do an `Acquire` when the mutex is not free, you have to wait, since `Acquire` has no transition from that state because of the `m = nil` guard.

[10] This process is mechanized in ESC; see http://www.research.digital.com/SRC/esc/Esc.html.
[11] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* **15**, 4 (Dec 1997), pp 391-411.

```
MODULE Mutex EXPORT acq, rel =                    % Acquire and Release

VAR m: (Thread + Null) := nil
% A mutex is either nil or the thread holding the mutex.
% The variable SELF is defined to be the thread currently making a transition.

APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

END Mutex
```

The thing we care about is that only one thread can be between `acq` and `rel` at a time. It's pretty obvious from the spec that this is true as long as we never get to `HAVOC` in `rel`. We can make it explicit with an invariant:

```
INVARIANT (ALL h, h' | h in critical section ==> ~ h' in critical section)
```

Here we have treated the PC's very informally; see handout 17 for the precise details. This invariant follows from

```
INVARIANT (ALL h | h in critical section ==> m = h)
```

which in turn follows by induction on the actions of `h` that don't include `rel`, plus the fact that no thread `h'` does `m.rel` unless `m = h'`. An invariant like this on the spec is sometimes called a *property*. The model-checking proof of an implementation of `Mutex` at the end of handout 17 shows how to establish a property directly from the implementation. This is contrary to the religion of this course, which is to always do simulation proofs, but it can be effective nonetheless.

We usually need lots of mutexes, not just one, so we change `MODULE` to `CLASS` (see section 7 of handout 4, the Spec reference manual). This creates a module with a function variable in which to store the state of lots of mutexes, and a `Mutex` type with `new`, `acq`, and `rel` methods whose value indexes the variable.

If `m` is a mutex that protects the variable `x`, you use it like this:

```
m.acq; touch x; m.rel
```

That is, you touch `x` only while `m` is acquired.

You may be familiar with this style of programming from Java, where a synchronized object has an implicit lock that is automatically acquired at the start of every method and released at the end. This means that all the private fields are protected by the implicit lock. Another way to think about this style of programming is that the private fields are internal state of an isolated system. Only actions of this system can touch the fields, and only one such action runs at a time. As long as the actions don't touch any other objects, they are obviously atomic. When an action needs to touch more than one object this simple view is no longer adequate. We explore some of the complications below.

Note that Java locks differ from the ones we have specified in that they are *re-entrant*: the same thread can acquire the same lock repeatedly. In the spec above this would lead to deadlock.

The only problem with the lock-before-touching rule for easy concurrency is that it's easy to make a mistake and program a race, though tying the lock to a class instance makes mistakes less likely. Races are bad bugs to have, because they are hard to reproduce; for this reason they are often called *Heisenbugs*, as opposed to the deterministic *Bohrbugs*. There is a substantial literature on methods for statically detecting failure to follow the locking rules.

Invariants

In fact things are not so simple, since a computation seldom consists of a single atomic action. A thread should not hold a lock forever (except on private data) because that will prevent any other thread that needs to touch the data from making progress. Furthermore, it often happens that a thread can't make progress until some other thread changes the data protected by a lock. A simple example of this is a FIFO buffer, in which a consumer thread doing a `Get` on an empty buffer must wait until some other producer thread does a `Put`. In order for the producer to get access to the data, the consumer must release the lock. Atomicity does not apply to code like this that touches a shared variable `x` protected by a mutex `m`:

```
m.acq; touch x; m.rel; private computation; m.acq; touch x; m.rel
```

This code releases a lock and later re-acquires it, and therefore isn't atomic. So we need a different way to think about this situation, and here it is.

> After the `m.acq` the only thing you can assume about `x` is an invariant that holds whenever `m` is unlocked.

As usual, the invariant must be true initially. While `m` is locked you can modify `x` so that the invariant doesn't hold, but you must re-establish it before unlocking `m`. While `m` is locked, you can also poke around in `x` and discover facts that are not implied by the invariant, but you cannot assume that any of these facts are still true after you unlock `m`.

To use this methodology effectively, of course, you must *write the invariant down*.

The rule about invariants sheds some light on why the following simple locking strategy doesn't help with concurrent programming:

> Every time you touch a shared variable `x`, acquire a lock just before and release the lock just after.

The reason is that once you have released the lock, you can't assume anything about `x` except what is implied by the invariant. The whole point of holding the lock is that it allows you to know more about `x` as long as you continue to hold the lock.

Here is a more picturesque way of describing this method. To do easy concurrent programming:

> first you put your hand over some shared variables, say `x` and `y`, so that no one else can change them,
>
> then you look at them and perhaps do something with them, and
>
> finally you take your hand away.

The reason `x` and `y` can't change is that the rest of the program obeys some conventions; in particular, it acquires locks before touching shared variables. There are other, trickier conventions that can keep `x` and `y` from changing; we will see some of them later on.

This viewpoint sheds light on why fault-tolerant programming is hard: `Crash` is no respecter of conventions, and the invariant must be maintained even though a `Crash` may stop an update in mid-flight and reset all or part of the volatile state.

Scheduling: Condition variables

If a thread can't make progress until some condition is established, and therefore has to release a lock so that some other thread can establish the condition, the simplest idiom is

```
m.acq; DO ~ condition(x) involving x => m.rel; m.acq OD; touch x; m.rel
```

That is, you loop waiting for `condition(x)` to be true before touching `x`. This is called "busy waiting", because the thread keeps computing, waiting for the condition to become true. It tests `condition(x)` only with the lock held, since `condition(x)` touches `x`, and it keeps releasing the lock so that some other thread can change `x` to make `condition(x)` true.

This code is correct, but reacquiring the lock immediately makes it more difficult for another thread to get it, and going around the loop while the condition remains false wastes processor cycles. Even if you have your own processor, this isn't a good scheme because of the system-wide cost of repeatedly acquiring the lock.

The way around these problems is an optimization that replaces `m.rel; m.acq` in the box with `c.wait(m)`, where `c` is a 'condition variable'. The `c.wait(m)` releases `m` and then blocks the thread until some other thread does `c.signal`. Then it reacquires `m` and returns. If several threads are waiting, `signal` picks one or more to continue in a fair way. The variation `c.broadcast` continues all the waiting threads.

Here is the spec for condition variables. It says that the state is the set of threads waiting on the condition, and it allows for lots of `C`'s because it's a class. The `wait` method is especially interesting, since it's the first procedure we've seen in a spec that is not atomic (except for the clumsy non-atomic specs for disk and file writes, and `ObjNames`). This is because the whole point is that during the `wait` other threads have to run, access the variables protected by the mutex, and signal the condition variable. Note that `wait` takes an extra parameter, the mutex to release and reacquire.

The spec doesn't say anything about blocking or suspending the thread. The blocking happens at the semi-colon between the two atomic actions of `wait`. An implementation works by keeping a queue of blocked threads in the condition variable; `signal` takes a thread off this queue and makes it ready to run again. Of course the code must take care to make the queuing and blocking of the thread effectively atomic, so that the thread doesn't get unqueued and scheduled to run again before it has been suspended. It must also take care not to miss a `signal` that occurs between queuing `SELF` on `c` and blocking the thread. This is usually done with a 'wakeup-waiting switch', a bit in the thread state that is set by `signal` and checked atomically with blocking the thread. See `MutexImpl` and `ConditionImpl` in handout 17 for an example of how to do this implementation.

```
CLASS Condition EXPORT wait, signal, broadcast =

TYPE M = Mutex

VAR c          :  SET Thread := {}
% Each condition variable is the set of waiting threads.

PROC wait(m) =
    << c \/ := {SELF}; m.rel >>;                    % m.rel=HAVOC unless SELF IN m
    << ~ (SELF IN c) => m.acq >>

APROC signal() = <<
% Remove at least one thread from c.  In practice, usually just one.
```



```
    IF  VAR t: SET Thread | t <= c /\ t # {} => c - := t [*] SKIP FI >>

APROC broadcast() = << c := {} >>

END Condition
```

For this scheme to work, a thread that changes `x` so that the condition becomes true must do a `signal` or `broadcast`, in order to allow some waiting thread to continue. A foolproof but inefficient strategy is to have a single condition variable for `x` and to do `broadcast` whenever `x` changes at all. More complicated schemes can be more efficient, but are more likely to omit a `signal` and leave a thread waiting indefinitely. The paper by Birrell in handout 15[12] gives many examples and some good advice.

Note that you are *not* entitled to assume that the condition is true just because `wait` returns. That would be a little more efficient for the waiter, but it would be much more error prone, and it would require a tighter spec for `wait` and `signal` that is often less efficient to code. You are supposed to think of `c.wait(m)` as just an optimization of `m.rel; m.acq`. This idiom is very robust. Warning: many people don't agree with this argument, and define stronger condition variables; when reading papers on this subject, make sure you know what religion the author embraces.

More generally, after `c.wait(m)` you cannot assume anything about `x` beyond its invariant, since the `wait` unlocks `m` and then locks it again. After a `wait`, only the invariant is guaranteed to hold, not anything else that was true about `x` before the wait.

Really easy concurrency

An even easier kind of concurrency uses buffers to connect independent modules, each with its own set of variables disjoint from those of any other module. Each module consumes data from some predecessor modules and produces data for some successor modules. In the simplest case the buffers are FIFO, but they might be unordered or use some other ordering rule. A little care is needed to program the buffers' `Put` and `Get` operations, but that's all. This is often called 'pipelining'. The fancier term 'data flow' is used if the modules are connected not linearly but by a more general DAG.

A second really easy kind of concurrency is pure data parallelism, as in the example earlier of adding two vectors to get a third. Here there is no data shared among the threads, so no locking is needed. Unfortunately, pure data parallelism is rare—usually there is some shared data to spoil the purity, as in the example of scalar product. The same kind of thing happens in graphics: when Photoshop operates on a large array of pixels, it's possible that the operation is strictly per-pixel, but more often it involves a neighborhood of each pixel, so that there is some sharing.

A third really easy kind of concurrency is provided by transaction processing or TP systems, in which an application program accepts some input, reads and updates a shared database, and generates some output. The transaction mechanism makes this entire operation atomic, using techniques that we will describe later. The application programmer doesn't have to think about concurrency at all. In fact, the atomicity usually includes crash recovery, so she doesn't have to think about fault-tolerance either.

[12] Andrew Birrell, *An Introduction to Programming with C# Threads*, Research Report , Microsoft Corporation, May 2005 (handout 16).

In the pure version of TP, there is *no* state preserved outside the transaction except for the shared database. This means that the only invariants are invariants on the database; the programmer doesn't have to worry about mistakenly keeping private state that records something about the shared state after locks are released. Furthermore, it means that a transaction can run on any machine that can access the database, so the TP system can take care of launching programs and doing load balancing as well as locking and fault tolerance. How easy can it get?

A fourth way to get really easy concurrency is functional programs. If two threads operate on data that is either *immutable* or private to a thread, they can run concurrently without any complications. This happens automatically in *functional* programs, in which there is *no* mutable state but only the result of function executions. The most widely used example of a functional programming system is the SQL language for writing queries on relational databases. SQL lets you combine (join, in technical terms), project, filter, and aggregate information from tables that represent relations. SQL queries don't modify the tables, but simply produce results, so they can easily be run in parallel, and since there is a large market for large queries on large databases, much effort have been successfully invested in figuring out how to run such queries efficiently on machines with dozens or hundreds of processors.

More commonly a language is mostly functional: most of the computation is functional, but the results of functional computations are used to change the state. The grandfather of such languages is APL, which lets you write big functional computations on vectors and matrices, but then assigns the results to variables that are used in subsequent computations. Systems like Matlab and Mathematica are the modern descendants of APL.

Hard concurrency

If you don't program according to the rules for locks, then you are doing hard concurrency, and it will be hard. Why bother? There are three reasons:

> You may have to code mutexes and condition variables on top of something weaker, such as the atomic reads and writes of memory that a basic processor or file system gives you. Of course, only the low-level runtime implementer will be in this position.
>
> It may be cheaper to use weaker primitives than mutexes. If efficiency is important, hard concurrency may be worth the trouble. But you will pay for it, either in bugs or in careful proofs of correctness.
>
> It may be important to avoid waiting for a lock to be released. Even if a critical section is coded carefully so that it doesn't do too much computing, there are still ways for the lock to be held for a long time. If the thread holding the lock can fail independently (for example, if it is in a different address space or on a different machine), then the lock can be held indefinitely. If the thread can get a page fault while holding the lock, then the lock can be held for a disk access time. A concurrent algorithm that prevents one slow (or failed) thread from delaying other threads too much is called 'wait-free'.[13]

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (Jan. 1991), pp 124-149. There is a general method for implementing wait-free concurrency, given a primitive at least as strong as compare-and-swap; it is described in M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* **15,** 9 (Nov. 1993), pp 745-770. The idea is the same as optimistic concurrency control (see handout 20): do the work on a separate version of the state, and then install it atomically with compare-and-swap, which detects when someone else has gotten ahead of you.

In fact, the "put out your hand" way of looking at things applies to hard concurrency as well. The difference is that instead of preventing `x` and `y` from changing at all, you do something to ensure that some predicate `p(x, y)` will remain true. The convention that the rest of the program obeys may be quite subtle. A simple example is the careful write solution to keeping track of free space in a file system (handout 7 on formal concurrency, page 16), in which the predicate is

```
free(da) ==> ~ Reachable(da).
```

The special case of locking maintains the strong predicate `x = x0 /\ y = y0` (unless you change `x` or `y` yourself).

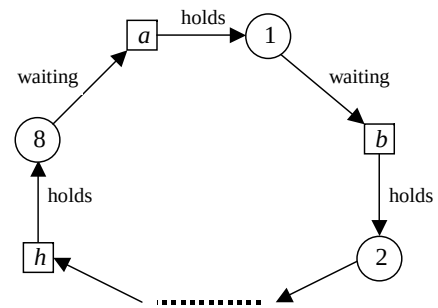
We postpone a detailed study of hard concurrency to handout 17.

Problems in easy concurrency: Deadlock

The biggest problem for easy concurrency is deadlock, in which there is a cycle of the form

> Lock *a* is held by thread 1.
> Thread 1 is waiting for lock *b*.
> Lock *b* is held by thread 2.
> ...
> Lock *h* is held by thread 8.
> Thread 8 is waiting for lock *a*.

All the locks and threads are nodes in a lock graph with the edges "lock *a* is held by thread 1", "thread 1 is waiting for lock *b*", etc.



The way to deal with this that is simplest for the application programmer is to *detect* a deadlock[14] and automatically roll back one of the threads, undoing any changes it has made and releasing its locks. Then the rolled-back thread retries; in the meantime, the others can proceed. Unfortunately, this approach is only practical when automatic rollback is possible, that is, when all the changes are done as part of a transaction. Handout 19 on sequential transactions explains how this works.

Note that from inside a module, absence of deadlock is a safety property: something bad doesn't happen. The "bad" thing is a loop of the kind just described, which is a well-defined property of certain states, indeed, one that is detected by systems that do deadlock detection. From the outside, however, you can't see the internal state, and the deadlock manifests itself as the failure of the module to make any progress.

[14] For ways of detecting deadlocks, see Gray and Reuter, pp 481-483 and A. Thomasian, Two phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems* **18**, 4 (Dec. 1993), pp. 579-625.

The main alternative to deadlock detection and rollback is to *avoid* deadlocks by defining a partial order on the locks, and abiding by a rule that you only acquire a lock if it is greater than every lock you already hold. This ensures that there can't be any cycles in the graph of threads and locks. Note that there is no requirement to release the locks in order, since a release never has to wait.

To implement this idea you

- annotate each shared variable with its protecting lock (which you are supposed to do anyway when practicing easy concurrency),
- state the partial order on the locks, and
- annotate each procedure or code block with its 'locking level' `ll`, the maximum lock that can be held when it is entered, like this: `ll <= x`.

Then you always know textually the biggest lock that can be held (by starting at the procedure entry with the annotation, and adding locks that are acquired), and can check whether an `acq` is for a bigger lock as required, or not. With a stronger annotation that tells exactly what locks are held, you can subtract those that are released as well. You also have to check when you call a procedure that the current locking level is consistent with the procedure's annotation. This check is very similar to type checking.

Having described the basic method, we look at some examples of how it works and where it runs into difficulties.

If resources are arranged in a tree and the program always traverses the tree down from root to leaves, or up from leaves to root (in the usual convention, which draws trees upside down, with the root at the top), then the tree defines a suitable lock ordering. Examples are a strictly hierarchical file system or a tree of windows. If the program sometimes goes up and sometimes goes down, there are problems; we discuss some solutions shortly. If instead of a tree we have a DAG, it still defines a suitable lock ordering.

Often, as in the file system example, this graph is actually a data structure whose links determine the accessibility of the nodes. In this situation you can choose when to release locks. If the graph is static, it's all right to release locks at any time. If you release each lock before acquiring the next one, there is no danger of deadlock regardless of the structure of the graph, because a flat ordering (everything unordered) is good enough as long as you hold at most one lock at a time. If the graph is dynamic and a node can disappear when it isn't locked, you have to hold on to one lock at least until after you have acquired the next one. This is called 'lock coupling', and a cyclic graph can cause deadlock. We will see an example of this when we study hierarchical file systems in handout 15.

Here is another common locking pattern. Consider a program that manipulates objects named by handles and maintains a set of these objects. For example, the objects might be buffers, and the set the buffers that are non-empty. One thread works on an object and sometimes needs to mess with the set, for instance when a buffer changes from empty to non-empty. Another thread processes the set and needs to mess with some of the objects, for instance to empty out the buffers at regular intervals. It's natural to have a lock `h.m` on each object and a lock `ms` on the set. How should they be ordered? We work out a solution in which the ordering of locks is every `h.m < ms`.


```
TYPE H          =  Int WITH {acq:=(\h|ot(h).m.acq),   % Handle (index in ot)
                             rel:=(\h|ot(h).m.rel),
                             y  :=(\h|ot(h).y ), empty:=...}


VAR  s          :  SET H                              % ms protects the set s
     ms         :  Mutex
     ot         :  H -> [m: Mutex, y: Any]            % Object Table.  m  protects  y,
                                                      % which is the object's data
```

Note that each piece of state that is not a mutex is annotated with the lock that protects it: `s` with `ms` and `y` with `m`. The 'object table' `ot` is fixed and therefore doesn't need a lock.

We would like to maintain the invariant "object is non-empty" = "object in set": `~ h.empty = h IN s`. This requires holding both `h.m` and `ms` when the emptiness of an object changes. Actually we maintain "`h.m` is locked `\/ (~ h.empty = h IN s)`", which is just as good. The `Fill` procedure that works on objects is very straightforward; `Add` and `Drain` are functions that compute the new state of the object in some unspecified way, leaving it non-empty and empty respectively. Note that `Fill` only acquires `ms` when it becomes non-empty, and we expect this to happen on only a small fraction of the calls.

```
PROC Fill(h, x: Any) =
% Update the object h using the data x
    h.acq;
    IF h.empty => ms.acq; s \/ := {h}; ms.rel [*] SKIP FI;
    ot(h).y := Add(h.y, x);
    h.rel
```

The `Demon` thread that works on the set is less straightforward, since the lock ordering keeps it from acquiring the locks in the order that is natural for it.

```
THREAD Demon() = DO
    ms.acq;
    IF  VAR h | h IN s  =>
            ms.rel;
            h.acq; ms.acq;                            % acquire locks in order
            IF  h IN s =>                             % is h still in s?
                s - := {h}; ot(h).y := Drain(h.y)
            [*] SKIP
            FI;
            ms.rel; h.rel
    [*] ms.rel
    FI
  OD
```

`Drain` itself does no locking, so we don't show its body.

The general idea, for parts of the program like `Demon` that can't acquire locks in the natural order, is to collect the information you need, one mutex at a time, without making any state changes. Then reacquire the locks according to the lock ordering, check that things haven't changed (or at least that your conclusions still hold), and do the updates. If it doesn't work out, retry. Version numbers can make the 'didn't change' check cheap. This scheme is closely related to optimistic concurrency control, which we discuss later in connection with concurrent transactions.

An alternative approach in the hybrid scheme allows you to make state changes while acquiring locks, but then you must undo all the changes before releasing the locks. This is called 'compensation'. It makes the main line code path simpler, and it may be more efficient on

average, but coding the compensating actions and ensuring that they are always applied can be tricky.

It's possible to use a hybrid scheme in which you keep locks as long as you can, rather than preparing to acquire a lock by always releasing any larger locks. This works if you can acquire a lower lock 'cautiously', that is, with a failure indication rather than a wait if you can't get it. If you fail in getting a lower lock, fall back to the conservative scheme of the last paragraph. This doesn't simplify the code (in fact, it makes the code more complicated), but it may be faster.

Deadlock with condition variables: Nested monitors

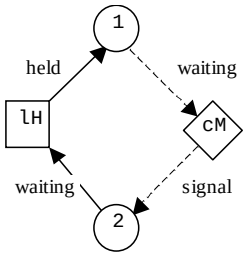
Since a thread can wait on a condition variable as well as on a lock, it's possible to have a deadlock that involves condition variables as well as locks. Usually this isn't a problem because there are many fewer conditions than locks, and the thread that signals a condition is coupled to the thread that waits on it only through the single lock that the waiting thread releases. This is fortunate, because there is no simple rule like the ordering rule for locks that can avoid this kind of deadlock. The lock ordering rule depends on the fact that a thread must be holding a lock in order to keep another thread waiting for that lock. In the case of conditions, the thread that will signal can't be distinguished in such a simple way.

The canonical example of deadlock involving conditions is known as "nested monitors". It comes up when there are two levels of abstraction, `H` and `M` (for high and medium; low would be confused with the `L` of locks), each with its own lock `lH` and `lM`. `M` has a condition variable `cM`. The code that deadlocks looks like this, if two threads `1` and `2` are using `H`, `1` needs to wait on `cM`, and `2` will signal `cM`.

```
H1: lH.lock; call M1
M1: lM.lock; cM.wait(lM)

H2: lH.lock; call M2
M2: lM.lock; cM.signal
```

This will deadlock because the `wait` in `M1` releases `lM` but not `lH`, so that `H2` can never get past `lH.lock` to reach `M2` and do the `signal`. This is not a lock-lock deadlock because it involves the condition variable `cM`, so a straightforward deadlock detector will not find it. The picture below illustrates the point.



To avoid this deadlock, don't wait on a condition with *any* locks held, unless you know that the `signal` can happen without acquiring any of these locks. The 'don't wait' is simple to check, given the annotations that the methodology requires, but the 'unless' may not be simple.

People have proposed to solve this problem by generalizing `wait` so that it takes a set of mutexes to release instead of just one. Why is this a bad idea? Aside from the problems of passing the right mutexes down from `H` to `M`, it means that any call on `M` might release `lH`. The `H` programmer

must to be careful not to depend on anything more than the `lH` invariant across any call to `M`. This style of programming is very error-prone.

Problems in easy concurrency: Scheduling

If there is a shortage of processor resources, there are various ways in which the simple easy concurrency method can go astray. In this situation we may want some threads to have priority over others, but subject to this constraint we want the processor resources allocated fairly. This means that the amount of time a task takes should be roughly proportional to the amount of work it does; in particular, we don't want short tasks to be blocked by long ones. A naïve view of fairness gives poor results, however. If all the tasks are equally important, it seems fair to give them equal shares of the CPU, but consider what happens if two tasks of the same length l start at the same time. With equal shares, both will take time $2l$ to finish. Running one to completion, however, means that it finishes in time l, while the other still takes only $2l$, for an average completion time of $1.5l$, which is clearly better.

What if all tasks are not equally important? There are various way this can happen. If there are deadlines, usually it's best to run the task with the closest deadline first. If you want to give different shares of the resource to different tasks, there's a wide variety of schemes to choose from. The simplest to understand is lottery scheduling, which gives each task lottery tickets in proportion to its share, and then chooses a ticket at random and runs that task.[15]

Starvation

If threads are competing for a resource, it's important to schedule them *fairly*. CPU time, just discussed, is not the only resource. In fact, the most common resources that need to be scheduled are locks. The specs we gave for locks and condition variables say nothing about the order in which competing threads acquire a lock when it's released. If a thread can be repeatedly passed over for `acq`, it will make no progress, and we say that it's *starved*. This is obviously bad. It's easy to avoid simple cases of starvation by keeping the waiting threads in a queue and serving the one at the head of the queue. This scheme fails if a thread needs to release and re-acquire locks to get its job done, as in some of the schemes for handling deadlock discussed above. Methods that detect a deadlock and abort one of the threads, requiring it to reacquire its locks, may never give a thread a chance to get all the locks it needs, and the `Demon` thread has the same problem. The opposite of starvation is *progress*.

A variation on starvation that is less serious but can still have crippling effects on performance is the *convoy* phenomenon, in which there is a high-traffic resource protected by a lock which is acquired frequently by many threads. The resource should be designed so that the lock only needs to be held for a short time; then the lock will normally be free and the resource won't be a bottleneck. If there is a pre-emptive scheduler, however, a thread can acquire the lock and get pre-empted. Lots of other threads will then pile up waiting for the lock. Once you get into this state it's hard to get out of it, because a thread that acquires the lock will quickly release it, but then, since it keeps the CPU, attempts to acquire the lock again and end up at the end of the queue, so that the queue never gets empty as it's supposed to and the computation is effectively serialized. The solution is to immediately give the CPU to a waiting thread when the lock is

[15] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, November 1994.

released if the queue is long; this increases scheduling overhead, but only until the queue empties.

Priority inversion

When there are priorities there can be "priority inversion". This happens when a low-priority thread `A` acquires a lock and then loses the CPU, either to a higher-priority thread or to round-robin scheduling. Now a high-priority thread `B` tries to acquire the lock and ends up waiting for `A`. Clearly the priority of `A` should be temporarily increased to that of `B` until `A` completes its critical section, so that `B` can continue. Otherwise `B` may wait for a long time while threads with priorities between `A` and `B` run, which is not what we had in mind when we set up the priority scheme. Unfortunately, many thread systems don't raise `A`'s priority in this situation.

Granularity of locks

A different issue is the 'granularity' of the locks: how much data each lock protects. A single lock is simple and cheap, but doesn't allow any concurrency. Lots of fine-grained locks allow lots of concurrency, but the program is more complicated, there's more overhead for acquiring locks, and there's more chance for deadlock (discussed earlier). For example, a file system might have a single global lock, one lock on each directory, one lock on each file, or locks only on byte ranges within a file. The goal is to have fine enough granularity that the queue of threads waiting on a mutex is empty most of the time. More locks than that don't accomplish anything.

It's possible to have an adaptive scheme in which locks start out fine-grained, but when a thread acquires too many locks they are collapsed into fewer coarser ones that cover larger sets of variables. This process is called 'escalation'. It's also possible to go the other way: a process keeps track of the exact variables it needs to lock, but takes out much coarser locks until there is contention. Then the coarse locks are 'de-escalated' to finer ones until the contention disappears.

Closely related to the choice of granularity is the question of how long locks are held. If a lock that protects a lot of data is held for a long time (for instance, across a disk reference or an interaction with the user) concurrency will obviously suffer. Such a lock should protect the minimum amount of data that is in flux during the slow operation. The concurrent buffered disk example in handout 15 illustrates this point.

On the other hand, sometimes you want to minimize the amount of communication needed for acquiring and releasing the same lock repeatedly. To do this, you hold onto the lock for longer than is necessary for correctness. Another thread that wants to acquire the lock must somehow signal the holder to release it. This scheme is commonly used in distributed coherent caches, in which the lock only needs to be held across a single read, write, or test-and-set operation, but one thread may access the same location (or cache line) many times before a different thread touches it.

Lock modes

Another way to get more concurrency at the expense of complexity is to have many lock 'modes'. A mutex has one mode, usually called 'exclusive' since 'mutex' is short for 'mutual exclusion'. A reader/writer lock has two modes, called exclusive and 'shared'. It's possible to have as many modes as there are different kinds of commuting operations. Thus all reads commute and therefore need only shared mode (reader) locks. But a write commutes with nothing and therefore needs an exclusive mode (write) lock. The commutativity of the operations

is reflected in a 'conflict relation' on the locks. For reader/writer or shared/exclusive locks this matrix is:

	None	Shared (read)	Exclusive (write)
None	OK	OK	OK
Shared (read)	OK	OK	Conflict
Exclusive (write)	OK	Conflict	Conflict

Just as different granularities bring a need for escalation, different modes bring a need for 'lock conversion', which upgrades a lock to a higher mode, for instance from shared to exclusive, or downgrades it to a lower mode.

Explicit scheduling

In simple situations, queuing for locks is an adequate way to schedule threads. When things are more complicated, however, it's necessary to program the scheduling explicitly because the simple first-come first-served queuing of a lock isn't what you want. A set of printers with different properties, for example, can be optimized across a set of jobs with different priorities, requirements for paper handling, paper sizes, color, etc. There have been many unsuccessful attempts to build general resource allocation systems to handle these problems. They fail because they are too complicated and expensive for simple cases, and not flexible enough for complicated ones. A better strategy is to program the scheduling as part of the application, using as many condition variables as necessary to queue threads that are waiting for resources. Application-specific data structures can keep track of the various resource demands and application-specific code, perhaps written on top of a library, can do the optimization.

Just as you must choose the granularity of locks, you must also choose the granularity of conditions. With just a few conditions (in the limit, only one), it's easy to figure out which one to wait on and which ones to signal. The price you pay is that a thread (or many threads) may wake up from a `wait` only to find that it has to wait again, and this is inefficient. On the other hand, with many conditions you can make useless wakeups very rare, but more care is needed to ensure that a thread doesn't get stuck because its condition isn't signaled.

Simple vs. fancy locks

We have described a number of features that you might want in a locking system:

- multiple modes with conversion, for instance from shared to exclusive;
- multiple granularities with escalation from fine to coarse and de-escalation from coarse to fine;
- deadlock detection.

Database systems typically provide these features. In addition, they acquire locks automatically based on how an application transaction touches data, choosing the mode based on what the operation is, and they can release locks automatically when a transaction commits. For a thorough discussion of database locking see Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, Chapter 8, pages 449-492.

The main reason that database systems have such elaborate locking facilities is that the application programmers are quite naive and can't be expected to understand the subtleties of

concurrent programming. Instead, the system does almost everything automatically, and the programmers can safely assume that execution is sequential. Automatic mechanisms that work well across a wide range of applications need to adapt in the ways listed above.

By contrast, a simple mutex has only one mode (exclusive), only one granularity, and no deadlock detection. If these features are needed, the programmer has to provide them using the mutex and condition primitives. We will study one example of this in detail in handout 17 on formal concurrency: building a reader/writer lock from a simple mutex. Many others are possible.

Summary of easy concurrency

There are four simple steps:

1. Protect each shared data item with a lock, and acquire the lock before touching the data.
2. Write down the invariant which holds on shared data when a lock isn't held, and don't depend on any property of the shared data unless it follows from the invariant. Establish the invariant before releasing the lock.
3. If you have to wait for some other thread to do something before you can continue, avoid busy waiting by waiting on a condition; beware of holding any locks when you do this. When you take some action that might allow a waiting thread to continue, signal the proper condition variable.
4. To avoid deadlock, define a partial order on the locks, and acquire a lock only if it is greater in the order than any lock you already hold. To make this work with procedures, annotate a procedure with a pre-condition: the maximum set of locks that are held whenever it's called.