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.

[1] I am indebted to Greg Nelson for this taxonomy, and for the object and set example of deadlock avoidance.

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`.

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.[2]

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.

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 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

[2] 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.

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.[3]

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.[4]

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.[5] 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.[6]

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.[7] The same programming paradigm is still used in many scientific problems however, at a coarser grain, and is called 'data-parallel' computing. In one step each processor does some computation on its private data.

[3] 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).
[4] 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.
[5] 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.
[6] See www.messageq.com or A. Dickman, *Designing Applications With Msmq: Message Queuing for Developers*, Addison-Wesley, 1998.
[7] 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.

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'.

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.[8] It's also possible to check a running program for violations of this discipline.[9]

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 and then releases them (not necessarily in the same order), 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.

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.

[8] This process is mechanized in ESC; see http://www.research.digital.com/SRC/esc/Esc.html.
[9] 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.

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.

```
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
```

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.

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*.

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.

```
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[10] 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.

Another 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

[10] Andrew Birrell, *An Introduction to Programming with Threads*, research report 35, Systems Research Center, Digital Equipment Corporation, January 1989.

concurrency at all. In fact, the atomicity usually includes crash recovery, so she doesn't have to think about fault-tolerance either.

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?

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'.[11]

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.

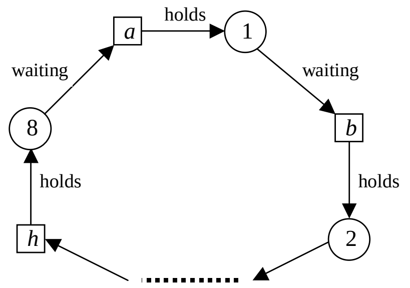
[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (Jan. 1991), pp 124-149. There is a general method for implementing wait-free concurrency, given a primitive at least as strong as compare-and-swap; it is described in M. Herlihy. A methodology 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.

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[12] 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.

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

[12] 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.

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. 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.

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

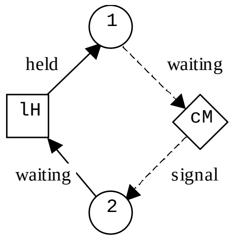
Since a thread can wait on a condition variable as well as on a lock, it's possible to have a deadlock that involves condition variables as well as locks. Usually this isn't a problem because there are many fewer conditions than locks, and the thread that signals a condition is coupled to the thread 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.

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 unless it follows from the invariant.
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.

15. Concurrent Disks and Directories

In this handout we give examples of more elaborate concurrent programs:

Code for `Disk.read` using the same kind of caching used in `BufferedDisk` from handout 7 on file systems, but now with concurrent clients.

Code for a directory tree or graph, as discussed in handout 12 on naming, but again with concurrent clients.

Concurrent buffered disk

The `ConcurrentDisk` module below is similar to `BufferedDisk` in handout 7 on file systems; both implement the `Disk` spec. For simplicity, we omit the complications of crashes. As in handout 7, the buffered disk is based on underlying code for `Disk` called `UDisk`, and calls on `UDisk` routines are in bold so you can pick them out easily.

We add a level of indirection so that we can have names (called `B`'s) for the buffers; a `B` is just an integer, and we keep the buffers in a sequence called `bv`. `B` has methods that let us write `b.db` for `bv(b).db` and similarly for other fields.

The `cache` is protected by a mutex `mc`. Each cache buffer is protected by a mutex `b.m`; when this is held, we say that the buffer is *locked*. Each buffer also has a count `users` of the number of `b`'s to the buffer that are outstanding. This count is also protected by `mc`. It plays the role of a readers lock on the cache reference to the buffer during a disk transfer: if it's non-zero, it is not OK to reassign the buffer to a different disk page. `GetBufs` increments `users`, and `InstallData` decrements it. No one waits explicitly for this lock. Instead, `read` just waits on the condition `moreSpace` for more space to become available.

Thus there are three levels of locking, allowing successively more concurrency and held for longer times:

`mc` is global, but is held only during pointer manipulations;

`b.m` is per buffer, but exclusive, and is held during data transfers;

`b.users` is per buffer and shared; it keeps the assignment of a buffer to a `DA` from changing.

There are three design criteria for the code:

1. Don't hold `mc` during an expensive operation (a disk access or a block copy).
2. Don't deadlock.
3. Handle additional threads that want to read a block being read from the disk.

You can check by inspection that the first is satisfied. As you know, the simple way to ensure the second is to define a partial order on the locks, and check that you only acquire a lock when it is greater than one you already have. In this case the order is `mc` < every `b.m`. The `users` count takes care of the third.

The loop in `read` calls `GetBufs` to get space for blocks that have to be read from the disk (this work was done by `MakeCacheSpace` in handout 7). `GetBufs` may not find enough free buffers, in which case it returns an empty set to `read`, which waits on `moreSpace`. This condition is signaled by the demon thread `FlushBuf`. A real system would have signaling in the other direction too, from `GetBufs` to `FlushBuf`, to trigger flushing when the number of clean buffers drops below some threshold.

The boxes in `ConcurrentDisk` highlight places where it differs from `BufferedDisk`. These are only highlights, however, since the code differs in many details.

```
CLASS ConcurrentDisk EXPORT read, write, size, check, sync =

TYPE
    % Data, DA, DB, Blocks, Dsk, E as in Disk
    I           =  Int
    J           =  Int

    Buf         =  [db, m, users: I, clean: Bool]    % m protects db, mc the rest
    M           =  Mutex
    B           =  Int WITH {m    :=(\b|bv(b).m),    % index in bv
                             db   :=(\b|bv(b).db),
                             users:=(\b|bv(b).users),
                             clean:=(\b|bv(b).clean)}
    BS          =  SET B

CONST
    DBSize      := Disk.DBSize
    nBufs       := 100
    minDiskRead := 5                                 % wait for this many Bufs

VAR
    % uses UDisk's disk, so there's no state for that
    udisk       :  Disk
    cache       := (DA -> B){}                       % protected by mc
    mc          :  M                                 % protects cache, users
    moreSpace   :  Condition.C                       % wait for more space
    bv          :  (B -> Buf)                        % see Buf for protection
    flushing    :  (DA + Null) := nil                % only for the AF

% ABSTRACTION FUNCTION Disk.disk(0) = (\ da |
    (    cache!da /\ (cache(da).m not held \/ da = flushing) => cache(da).db
     [*] UDisk.disk(0)(da) ))
```

The following invariants capture the reasons why this code works. They are not strong enough for a formal proof of correctness.

```
% INVARIANT 1: ( ALL da :IN cache.dom, b |
    b = cache(da) /\ b.m not held /\ b.clean ==> b.db = UDisk.disk(0)(da) )
```

A buffer in the cache, not locked, and clean agrees with the disk (if it's locked, the code in `FlushBuf` and the caller of `GetBufs` is responsible for keeping track of whether it agrees with the disk).

```
% INVARIANT 2: (ALL b | {da | cache!da /\ cache(da) = b}.size <= 1)
```

A buffer is in the cache at most once.

```
% INVARIANT 3: mc not held ==> (ALL b :IN bv.dom | b.clean /\ b.users = 0
                                                  ==> b.m not held)
```

If `mc` is not held, a clean buffer with `users = 0` isn’t locked.

```
PROC new(size: Int) -> Disk =
     self := StdNew(); udisk := udisk.new(size);
     mc.acq; DO VAR b | ~ bv!b => VAR m := m.new() |
          bv(b) := Buf{m := m, db := {}, users := 0, clean := true}
     OD; mc.rel
     RET self

PROC read(e) -> Data RAISES {notThere} =
     udisk.check(e);
     VAR data := Data{}, da := e.da, upto := da + e.size, i |
          mc.acq;
          % Note that we release mc before a slow operation (bold below)
          % and reacquire it afterward.
          DO da < upTo => VAR b, bs |                      % read all the blocks
               IF   cache!da =>
                    b := cache(da);                         % yes, in buffer b; copy it
                    % Must increment users before releasing mc.
                    bv(b).users + := 1; mc.rel;
                    % Now acquire m before copying the data.
                    % May have to wait for m if the block is being read.
                    b.m.acq; data + := b.db; b.m.rel;
                    mc.acq; bv(b).users - := 1;
                    da := da + 1
                [*] i := RunNotInCache(da, upTo);          % da not in the cache
                    bs := GetBufs(da, i); i := bs.size;     % GetBufs is fast
                    IF  i > 0 =>
                         mc.rel; data + := InstallData(da, i); mc.acq;
                         da + := i
                    [*] moreSpace.wait(mc)
                    FI
               FI
          OD; mc.rel; RET data

FUNC RunNotInCache(da, upTo: DA) -> I =                    % mc locked
     RET {i | da + i <= upTo /\ (ALL j :IN i.seq | ~ cache!(da + j)).max
```

`GetBufs` tries to return `i` buffers, but it returns at least `minDiskRead` buffers (unless `i` is less than this) so that `read` won’t do lots of tiny disk transfers. It’s tempting to make `GetBufs` always succeed, but this means that it must do a `Wait` if there’s not enough space. While `mc` is released in the `Wait`, the state of the cache can change so that we no longer want to read `i` pages. So the choice of `i` must be made again after the `Wait`, and it’s most natural to do the `Wait` in `read`.

If `users` and `clean` were protected by `m` (as `db` is) rather than by `mc`, `GetBufs` would have to acquire pages one at a time, since it would have to acquire the `m` to check the other fields. If it couldn’t find enough pages, it would have to back out. This would be both slow and clumsy.

```
PROC GetBufs(da, i) -> BS =
% mc locked. Return some buffers assigned to da, da+1, ..., locked, and
% with users = 1, or {} if there's not enough space. No slow operations.
  VAR bs := {b | b.users = 0 /\ b.clean} |                  % the usable buffers
     IF   bs.size >= {i, minDiskRead}.min =>                % check for enough buffers
          i := {i, bs.size}.min;
          DO VAR b | b IN bs /\ b.users = 0 =>
               % Remove the buffer from the cache if it’s there.
               IF VAR da' | cache(da') = b => cache := cache{da' -> } [*] SKIP FI;
               b.m.acq; bv(b).users := 1; cache(da) := b; da + := 1
          OD; RET {b :IN bs | b.users > 0}
     [*]  RET {}                                            % too few; caller must wait
     FI
```

In handout 7, `InstallData` is done inline in `read`.

```
PROC InstallData(da, i) = VAR data, j := 0 |
% Pre: cache(da) .. cache(da+i-1) locked by SELF with users > 0.
     data := udisk.read(E{da, i});
     DO j < i => VAR b := cache(da + j) |
          bv(b).db := udisk.DToB(data).sub(j); b.m.rel;
          mc.acq; bv(b).users - := 1; mc.rel;
          j + := 1
     OD; RET data
```

`PROC write` is omitted. It sets `clean` to `false` for each block it writes. The background thread `FlushBuf` does the writes to disk. Here is a simplified version that does not preserve write order. Note that, like `read`, it releases `mc` during a slow operation.

```
THREAD FlushBuf() = DO                                      % flush a dirty buffer
     mc.acq;
     IF   VAR da, b | b = cache(da) /\ b.users = 0 /\ ~ b.clean =>
          flushing := true;                                 % just for the AF
          b.m.acq; bv(b).clean := true; mc.rel;
          udisk.write(da, b.db);
          flushing := false;
          b.m.rel; moreSpace.signal
      [*] mc.rel
     OD

% Other procedures omitted

END ConcurrentDisk
```

Concurrent directory operations

In handout 12 on naming we gave an `ObjNames` spec for looking up path names in a tree of graph of directories. Here are the types and state from `ObjNames`:

```
TYPE D          =  Int                                  % Just an internal name
                   WITH {get:=GetFromS, set:=SetInS}    % get returns nil if undefined
     Link       =  [d: (D + Null), pn]                  % d=nil means the containing D
     Z          =  (V + D + Link + Null)                % nil means undefined
     DD         =  N -> Z

CONST
     root       :  D := 0
     s          := (D -> DD){}{root -> DD{}}            % initially empty root

APROC GetFromS(d, n) -> Z =                             % d.get(n)
     << RET s(d)(n) [*] RET nil >>

APROC SetInS  (d, n, z)   =                             % d.set(n, z)
% If z = nil, SetInS leaves n undefined in s(d).
     << IF z # nil => s(d)(n) := z [*] s(d) := s(d){n -> } FI >>
```

We wrote the spec to allow the bindings of names to change during lookups, but it never reuses a `D` value or an entry in `s`. If it did, a lookup of `/a/b` might obtain the `D` for `/a`, say `dA`, and then `/a` might be deleted and `dA` reused for some entirely different directory. When the lookup continues it will look for `b` in that directory. This is definitely not what we have in mind.

Code, however, will represent a `DD` by some data structure on disk (and cached in RAM), and if the directory is deleted it will reuse the space. This code needs to prevent the anomalous behavior we just described. The simplest way to do so is similar to the `users` device in `ConcurrentDisk` above: a shared lock that prevents the directory data structure from being deleted or reused.

The situation is trickier here, however. It's necessary to make sufficiently atomic the steps of first looking up `a` to obtain `dA`, and then incrementing `s(dA).users`. To do this, we make `users` a true readers lock, which prevents changes to its directory. In particular, it prevents an entry from being deleted or renamed, and thus prevents a subdirectory from being deleted. Then it's sufficient to hold the lock on `dA`, look up `b` to obtain `dB`, and acquire the lock on `dB` before releasing the lock on `dA`. This is called 'lock coupling'.

As we saw in handout 12, the amount of concurrency allowed there makes it possible for lookups done during renames to produce strange results. For example, `Read(/a/x)` can return `3` even though there was never any instant at which the path name `/a/x` had the value `3`, or indeed was defined at all. We copy the scenario from handout 12. Suppose:

initially `/a` is the directory `d1` and `/b` is undefined;

initially `x` is undefined in `d1`;

concurrently with `Read(/a/x)` we do `Rename(/a, /b); Write(/b/x, 3)`.

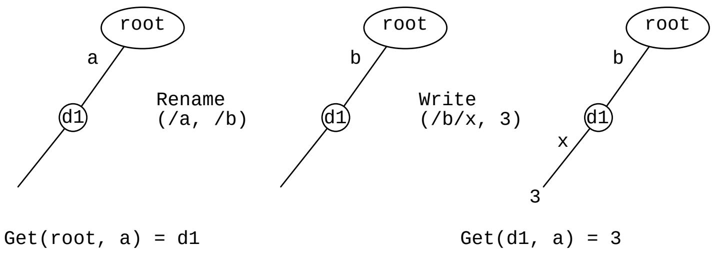
The following sequence of actions yields `Read(/a/x) = 3`:

In the `Read` , `Get(root, a) = d1`

`Rename(/a, /b)` makes `/a` undefined and `d1` the value of `/b`

`Write(/b/x, 3)` makes `3` the value of `x` in `d1`

In the `Read`, `RET d1.get(x)` returns `3`.



Obviously, whether this possibility is important or not depends on how clients are using the name space.

To avoid this kind of anomaly, it's necessary to hold a read lock on every directory on the path. When the directory graph is cyclic, code that acquires each lock in turn can deadlock. To avoid this deadlock, it's necessary to write more complicated code. Here is the idea.

Define some arbitrary ordering on the directory locks (say based on the numeric value of `D`). When doing a lookup, if you need to acquire a lock that is less than the biggest one you hold, release the bigger locks, acquire the new one, and then repeat the lookup from the point of the first released lock to reacquire the released locks and check that nothing has changed. This may happen repeatedly as you look up the path name.

This can be made more efficient (and more complicated, alas) with a 'tentative' `Acquire` that returns a failure indication rather than waiting if it can't acquire the lock. Then it's only necessary to backtrack when another thread is actually holding a conflicting write lock.

16. Paper: Programming with Threads

The attached paper by Andrew Birrell, *Introduction to Programming with Threads*, appeared as report 35 of the Systems Research Center, Digital Equipment Corp., Jan. 1989. A somewhat revised version appears as chapter 4 of *Systems Programming with Modula-3*, Greg Nelson ed., Prentice-Hall, 1991, pp 88-118.

Read it as an adjunct to the lecture on practical concurrency. It explains how to program with threads, mutexes, and condition variables, and it contains a lot of good ad vice and examples.

35

An Introduction to Programming with Threads

by Andrew D. Birrell

January 6, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

An Introduction to Programming with Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with “threads”. A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs.

CR categories and Subject Descriptors: D.1.4 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*Concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Threads, Concurrency, Multi-processing, Synchronization

CONTENTS

INTRODUCTION

Many experimental operating systems, and some commercial ones, have recently included support for concurrent programming. The most popular mechanism for this is some provision for allowing multiple lightweight "threads" within a single address space, used from within a single program.[1]

Programming with threads introduces new difficulties even for experienced programmers. Concurrent programming has techniques and pitfalls that do not occur in sequential programming. Many of the techniques are obvious, but some are obvious only with hindsight. Some of the pitfalls are comfortable (for example, deadlock is a pleasant sort of bug—your program stops with all the evidence intact), but some take the form of insidious performance penalties.

The purpose of this paper is to give you an introduction to the programming techniques that work well with threads, and to warn you about techniques or interactions that work out badly. It should provide the experienced sequential programmer with enough hints to be able to build a substantial multi-threaded program that works—correctly, efficiently, and with a minimum of surprises.

A "thread" is a straightforward concept: a single sequential flow of control. In a high-level language you normally program a thread using procedures, where the procedure calls follow the traditional stack discipline. Within a single thread, there is at any instant a single point of execution. The programmer need learn nothing new to use a single thread.

Having "multiple threads" in a program means that at any instant the program has multiple points of execution, one in each of its threads. The programmer can mostly view the threads as executing simultaneously, as if the computer were endowed with as many processors as there are threads. The programmer is required to decide when and where to create multiple threads, or to accept such decisions made for him by implementers of existing library packages or runtime systems. Additionally, the programmer must occasionally be aware that the computer might not in fact execute all his threads simultaneously.

Having the threads execute within a "single address space" means that the computer's addressing hardware is configured so as to permit the threads to read and write the same memory locations. In a high-level language, this usually corresponds to the fact that the off-stack (global) variables are shared among all the threads of the program. Each thread executes on a separate call stack with its own separate local variables. The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer.

Thread facilities are always advertised as being "lightweight". This means that thread creation, existence, destruction and synchronization primitives are cheap enough that the programmer will use them for all his concurrency needs.

Please be aware that I am presenting you with a selective, biased and idiosyncratic collection of techniques. Selective, because an exhaustive survey would be premature, and would be too exhausting to serve as an introduction—I will be discussing only the most important thread primitives, omitting features such as per-thread context information. Biased, because I present examples, problems and solutions in the context of one

[1] Throughout this paper I use the word "process" only when I mean a single flow of control associated one-to-one with an address space, since this now seems to be the most common usage of that word.

particular set of choices of how to design a threads facility—the choices we made in the Topaz system at Digital's Systems Research Center (SRC). Idiosyncratic, because the techniques presented here derive from my personal experience of programming with threads over the last ten years—I have not attempted to represent colleagues who might have different opinions about which programming techniques are "good" or "important". Nevertheless, I believe that an understanding of the ideas presented here will serve as a sound basis for programming with concurrent threads.

Throughout the paper I use examples written in Modula-2+ [12]. These should be readily understandable by anyone familiar with the Algol and Pascal family of languages. The examples are intended to illustrate points about concurrency and synchronization—don't try to use these actual algorithms in real programs.

Threads are not a tool for automatic parallel decomposition, where a compiler will take a visibly sequential program and generate object code to utilize multiple processors. That is an entirely different art, not one that I will discuss here.

WHY USE CONCURRENCY?

Life would be simpler if you didn't need to use concurrency. But there are a variety of forces pushing towards its use. The most recent is the advent of multi-processors. With these machines, there really are multiple simultaneous points of execution, and threads are an attractive tool for allowing a program to take advantage of the available hardware. The alternative, with most conventional operating systems, is to configure your program as multiple separate processes, running in separate address spaces. This tends to be expensive to set up, and the costs of communicating between address spaces are often high, even in the presence of shared segments. By using a lightweight multi-threading facility, the programmer can utilize the processors cheaply. This seems to work well in systems having up to about 10 processors, rather than 1000 processors.

A second area where threads are useful is in driving slow devices such as disks, networks, terminals and printers. In these cases an efficient program should be doing some other useful work while waiting for the device to produce its next event (such as the completion of a disk transfer or the receipt of a packet from the network). As we will see later, this can be programmed quite easily with threads by adopting an attitude that device requests are all sequential (i.e. they suspend execution of the invoking thread until the request completes), and that the program meanwhile does other work in other threads.

A third source of concurrency is human users. Humans are actually quite good at doing two or three things at a time, and seem to get offended if their computer cannot do as much. Again, threads are a convenient way of programming this. The typical arrangement of a modern window system is that each time the user invokes an action (by clicking a button with the mouse, for example), a separate thread is used to implement the action. If the user invokes multiple actions, multiple threads will perform them in parallel. (Note that the implementation of the window system probably also uses a thread to watch the mouse actions themselves, since the mouse is an example of a slow device.)

A final source of concurrency appears when building a distributed system. Here we frequently encounter shared network servers (such as a file server or a spooling print server), where the server is willing to service requests from multiple clients. Use of multiple threads allows the server to handle clients' requests in parallel, instead of artificially serializing them (or creating one server process per client, at great expense).

Sometimes you can deliberately add concurrency to your program in order to reduce the latency of operations (the elapsed time between calling a procedure and the procedure returning). Often, some of the work incurred by a procedure can be deferred, since it does not affect the result of the procedure. For example, when you add or remove something in a balanced tree you could happily return to the caller before re-balancing the tree. With threads you can achieve this easily: do the re-balancing in a separate thread. If the separate thread is scheduled at a lower priority, then the work can be done at a time when you are less busy (for example, when waiting for user input). Adding threads to defer work is a powerful technique, even on a uni-processor. Even if the same total work is done, reducing latency can improve the responsiveness of your program.

THE DESIGN OF A THREAD FACILITY

We can't discuss how to program with threads until we agree on the primitives provided by a multi-threading facility. The various systems that support threads offer quite similar facilities, but there is a lot of diversity in the details. In general, there are four major mechanisms: thread creation, mutual exclusion, waiting for events, and some arrangement for getting a thread out of an unwanted long-term wait. The discussions in the rest of the paper are organized around the particular primitives provided by the SRC thread facility [4], and I describe here their detailed semantics. In the appendix I describe the corresponding parts of the thread facilities provided by some other systems, and how they relate to the SRC facilities.

Thread creation

A thread is created by calling "Fork", giving it a procedure and an argument record. The effect of "Fork" is to create a new thread, and start that thread executing asynchronously at an invocation of the given procedure with the given arguments. When the procedure returns, the thread dies. Usually, "Fork" returns to its caller a handle on the newly created thread. The handle can be presented to a "Join" procedure. "Join" waits for the given thread to terminate, and returns the result of the given thread's initial procedure. The SRC thread creation facilities are as follows.[2]

```
TYPE Thread;
TYPE Forkee = PROCEDURE(REFANY): REFANY;
PROCEDURE Fork(proc: Forkee; arg: REFANY): Thread;
PROCEDURE Join(thread: Thread): REFANY;
```

As a simple example of their use, the following program fragment executes the procedure calls "a(x)" and "b(y)" in parallel, and assigns the result of calling "a(x)" to the variable "q".

[2]The type REFANY means a dynamically typed pointer to garbage-collected storage.

```
VAR t: Thread;
t := Fork(a, x);
p := b(y);
q := Join(t);
```

In practice, "Join" is not called very much at SRC, and some other thread facilities do not provide "Join" as a primitive. Most forked threads are permanent dæmon threads, or have no results, or communicate their results by some synchronization arrangement other than "Join". If a thread's initial procedure has returned and there is no subsequent call of "Join", the thread quietly evaporates.

Mutual exclusion

The simplest way that threads interact is through access to shared memory. In a high-level language, this is usually expressed as access to global variables. Since threads are running in parallel, the programmer must explicitly arrange to avoid errors arising when more than one thread is accessing the shared variables. The simplest tool for doing this is a primitive that offers mutual exclusion (sometimes called critical sections), specifying for a particular region of code that only one thread can execute there at any time. In the SRC design, this is achieved with the data type "Mutex" and the language's "LOCK" construct.

```
TYPE Mutex;
LOCK mutex DO ... statements ... END;
```

A mutex has two states: locked and unlocked, initially unlocked. The LOCK clause locks the mutex, then executes the contained statements, then unlocks the mutex. A thread executing inside the LOCK clause is said to "hold" the mutex. If another thread attempts to lock the mutex when it is already locked, the second thread blocks (enqueued on the mutex) until the mutex is unlocked.

The programmer can achieve mutual exclusion on a set of variables by associating them with a mutex, and accessing the variables only from a thread that holds the mutex (i.e., from a thread executing inside a LOCK clause that has locked the mutex). This is the basis of the notion of *monitors*, first described by Tony Hoare [9]. For example, in the following fragment the mutex "m" is associated with the global variable "head"; the LOCK clause provides mutual exclusion for adding the local variable "newElement" to a linked list whose head is "head".

```
TYPE List = REF RECORD ch: CHAR; next: List END;
VAR m: Thread.Mutex;
VAR head: List;

LOCK m DO
    newElement^.next := head;
    head := newElement;
END;
```

The simplest sort of mutex is a global variable (as in the fragment above); in this case at most one thread is executing inside the LOCK clause at any instant.

But a mutex might instead be part of a data structure (for example a field of a record). In that case, the variables it protects would also be part of the data structure (for example, the remaining fields of the same record). To express this, the LOCK clause would begin with an expression selecting the mutex field of the record, and the correct description would be that at most one thread can execute with the mutex held. This latter feature is typically used to arrange that at most one thread can access the fields of a particular record, but multiple threads can access different records in parallel. We will examine this usage in more detail in a later section.

Condition variables

You can view a mutex as a simple kind of resource scheduling mechanism. The resource being scheduled is the shared memory accessed inside the LOCK clause, and the scheduling policy is one thread at a time. But often the programmer needs to express more complicated scheduling policies. This requires use of a mechanism that allows a thread to block until some event happens. In the SRC design, this is achieved with a condition variable.

```
TYPE Condition;
PROCEDURE Wait(m: Mutex; c: Condition);
PROCEDURE Signal(c: Condition);
PROCEDURE Broadcast(c: Condition);
```

A condition variable is always associated with a particular mutex, and with the data protected by that mutex. In general a monitor consists of some data, a mutex, and zero or more condition variables. Notice that a particular condition variable is always used in conjunction with the *same* mutex and its data. The “Wait” operation atomically unlocks the mutex and blocks the thread (enqueued on the condition variable)[3]. The “Signal” operation does nothing unless there is a thread blocked on the condition variable, in which case it awakens at least one such blocked thread (but possibly more than one). The “Broadcast” operation is like “Signal”, except that it awakens all the threads currently blocked on the condition variable. When a thread is awoken inside “Wait” after blocking, it re-locks the mutex, then returns. Note that the mutex might not be available, in which case the thread will block on the mutex until it is available.

The mutex associated with a condition variable protects the shared data that is used for the scheduling decision. If a thread wants the resource, it locks the mutex and examines the shared data. If the resource is available, the thread continues. If not, the thread unlocks the mutex and blocks, by calling “Wait”. Later, when some other thread makes the resource available it awakens the first thread by calling “Signal” or “Broadcast”. For example, the following fragment allows a thread to block until a linked list (whose head is “head”) is not empty, then remove the top element of the list.

```
VAR nonEmpty: Thread.Condition;
```

[3]This atomicity guarantee avoids the problem known in the literature as the “wake-up waiting” race [13].

```
LOCK m DO
    WHILE head = NIL DO Thread.Wait(m, nonEmpty) END;
    topElement := head;
    head := head^.next;
END;
```

And the following fragment could be used by a thread adding an element to “head”.

```
LOCK m DO
    newElement^.next := head;
    head := newElement;
    Thread.Signal(nonEmpty);
END;
```

Alerts

The final aspect of the SRC thread facility is a mechanism for interrupting a particular thread, causing it to back out of some long-term wait or computation.

```
EXCEPTION Alerted;
PROCEDURE Alert(t: Thread);
PROCEDURE AlertWait(m: Mutex, c: Condition);
PROCEDURE TestAlert(): BOOLEAN;
```

The state of a thread includes a boolean known as “alert-pending”, initially false. A call of “AlertWait” behaves the same as “Wait”, except that if the thread’s alert-pending boolean is true, then instead of blocking on c it sets alert-pending to false, re-locks m and raises the exception “Alerted”. (Don’t worry about the semantics of Modula-2+ exceptions—think of them as a form of non-ignorable return code.) If you call “Alert(t)” when t is currently blocked on a condition variable inside a call of “AlertWait” then t is awoken, t re-locks the mutex m and then it raises the exception “Alerted”. If you call “Alert(t)” when t is not blocked in a call of “AlertWait”, all that happens is that its alert-pending boolean is set to true. The call “TestAlert” atomically tests and clears the thread’s alert-pending boolean.

For example, consider a “GetChar” routine, which blocks until a character is available on an interactive keyboard input stream. It seems attractive that if some other thread of the computation decides the input is no longer interesting (for example, the user clicked CANCEL with his mouse), then the thread should return from “GetChar”. If you happen to know the condition variable where “GetChar” blocks waiting for characters you could just signal it, but often that condition variable is hidden under one or more layers of abstraction. In this situation, the thread interpreting the CANCEL request can achieve its goal by calling “Thread.Alert(t)”, where “t” is the thread calling “GetChar”. For this to work, “GetChar” must contain something like the following fragment.

```
TRY
    WHILE empty DO Thread.AlertWait(m, nonEmpty) END;
    RETURN NextChar()
EXCEPT
    Thread.Alerted: RETURN EndOfFile
END;
```

Alerts are complicated, and their use produces complicated programs. We will discuss them in more detail later.

USING A MUTEX: ACCESSING SHARED DATA

The basic rule for using mutual exclusion is straightforward: in a multi-threaded program all shared mutable data must be protected by associating it with some mutex, and you must access the data only from a thread that is holding the associated mutex (i.e., from a thread executing within a LOCK clause that locked the mutex).

Unprotected data

The simplest bug related to mutexes occurs when you fail to protect some mutable data and then you access it without the benefits of synchronization. For example, consider the following code fragment. The global variable “table” represents a table that can be filled with REFANY values by calling “Insert”. The procedure works by inserting a non-NIL argument at index “i” of “table”, then incrementing “i”. The table is initially empty (all NIL).

```
            VAR table: ARRAY [0..999] OF REFANY;
            VAR i: [0..1000];

            PROCEDURE Insert(r: REFANY);
            BEGIN
                IF r # NIL THEN
1—                  table[i] := r;
2—                  i := i+1;
                END;
            END Insert;

            FOR i := 0 TO 999 DO table[i] := NIL END;
            i := 0;
```

Now consider what might happen if thread A calls “Insert(x)” concurrently with thread B calling “Insert(y)”. If the order of execution happens to be that thread A executes (1), then thread B executes (1), then thread A executes (2) then thread B executes (2), confusion will result. Instead of the intended effect (that “x” and “y” are inserted into “table”, at separate indexes), the final state would be that “y” is correctly in the table, but “x” has been lost. Further, since (2) has been executed twice, an empty (NIL) slot has

been left orphaned in the table. Such errors would be prevented by enclosing (1) and (2) in a LOCK clause, as follows.

```
PROCEDURE Insert(r: REFANY);
BEGIN
    IF r # NIL THEN
        LOCK m DO
            table[i] := r;
            i := i+1;
        END;
    END;
END Insert;
```

The LOCK clause enforces serialization of the threads’ actions, so that one thread executes the statements inside the LOCK clause, then the other thread executes them.

The effects of unsynchronized access to mutable data can be bizarre, since they will depend on the precise timing relationship between your threads. Also, in most environments the timing relationship is non-deterministic (because of real-time effects like page faults, or the use of real-time timer facilities, or because of actual asynchrony in a multi-processor system).

It would be a good idea for your language system to give you some help here, for example by syntax that prevents you accessing variables until you have locked the appropriate mutex. But most languages don’t offer this yet. Meanwhile, you need programmer discipline and careful use of searching and browsing tools. Such problems will arise less often if you use very simple, coarse grain, locking. For example, use a single mutex to protect all the global state of an entire module. Unfortunately, this can cause other problems, described below. So the best advice is to make your use of mutexes be as simple as possible, but no simpler. If you are tempted to use more elaborate arrangements, be entirely sure that the benefits are worth the risks, not just that the program looks nicer.

Invariants

When the data protected by a mutex is at all complicated, many programmers find it convenient to think of the mutex as protecting the *invariant* of the associated data. An invariant is a boolean function of the data that is true whenever the mutex is not held. So any thread that locks the mutex knows that it starts out with the invariant true. Each thread has the responsibility to restore the invariant before releasing the mutex. This includes restoring the invariant before calling “Wait”, since that unlocks the mutex.

For example, in the code fragment above (for inserting an element into a table), the invariant is that “i” is the index of the first NIL element in “table”, and all elements beyond index “i” are NIL. Note that the variables mentioned in the invariant are accessed only with this mutex held. Note also that the invariant is not true after the first assignment statement but before the second one—it is only guaranteed when the mutex is not being held.

Frequently the invariants are simple enough that you barely think about them, but often your program will benefit from writing them down explicitly. And if they are too complicated to write down, you’re probably doing something wrong. You might find it best to write down your invariants informally, as in the previous paragraph, or you might prefer to use some formal specification language such as Larch [7], the language we used to specify the SRC threads facility [4]. It is also generally a good idea to make it clear (by writing it down in the program) which mutex protects which data items.

Cheating

The rule that you must use a mutex to protect every access to global variables is based on a concurrency model where the actions of the threads are arbitrarily interleaved. If the data being protected by a mutex is particularly simple (for example just one integer, or even just one boolean), programmers are often tempted to skip using the mutex, since it introduces significant overhead and they “know” that the variables will be accessed with atomic instructions and that instructions are not interleaved. Before you succumb to this temptation, you must carefully consider the hardware where your program will run. If your single integer variable is word aligned, and if you are running on a uni-processor, and if your compiler generates the obvious instruction sequences and doesn’t slave variables into registers, then you will probably get the correct answer. In other circumstances you might not get the correct answer; or worse, you might usually get the correct answer but very occasionally get the wrong answer. For machine independent correct code, you absolutely must use a synchronization technique approved by your programming language.[4]

One cheating technique I have found helpful is to use unsynchronized access as a *hint*. In this context by “hint” I mean a cheap way of getting information that is either correct or causes you to invoke a more expensive, correct, way of getting the information. For example, if you want to call an initialization procedure exactly once you might use code like the following.

```
IF NOT initDone THEN
    LOCK m DO
        IF NOT initDone THEN
            Initialize();
            initDone := TRUE;
        END;
    END;
END;
```

This code is correct if you can assume that “initDone” is initialized to FALSE, that reading it without a mutex will not cause a runtime error, and that reading it without a mutex when its value is FALSE will give you the value FALSE. It doesn't matter if you occasionally might get FALSE instead of TRUE, since that will just cause you to lock the mutex and get the correct value. But I repeat: you should only do this after verifying that your programming language guarantees the validity of these assumptions.

[4] There is indeed a strong argument that there should be a way for the programmer to take advantage of the atomicity of instructions, and so avoid the cost of a LOCK clause. However, it seems to be difficult to define such a feature in an efficient but machine-independent fashion.

Deadlocks involving only mutexes

The simplest cases of deadlock occur when a thread tries to lock a mutex that it already holds (although some systems explicitly allow a thread to continue unhindered if it does this). There are numerous more elaborate cases of deadlock involving mutexes, for example:

> Thread A locks mutex M1;
> thread B locks mutex M2;
> thread A blocks trying to lock M2;
> thread B blocks trying to lock M1.

The most effective rule for avoiding such deadlocks is to apply a partial order to the acquisition of mutexes in your program. In other words, arrange that for any pair of mutexes { M1, M2 }, each thread that needs to hold M1 and M2 simultaneously locks M1 and M2 in the same order (for example, M1 is always locked before M2). This rule completely avoids deadlocks involving only mutexes (though as we will see later, there are other potential deadlocks when your program uses condition variables).

There is a technique that sometimes makes it easier to achieve this partial order. In the example above, thread A probably wasn’t trying to modify exactly the same set of data as thread B. Frequently, if you examine the algorithm carefully you can partition the data into smaller pieces protected by separate mutexes. For example, when thread B wanted to lock M1, it might actually be wanting access to data disjoint from the data that thread A was accessing under M1. In such a case you might protect this disjoint data with a separate mutex, M3, and avoid the deadlock. Note that this is just a technique to enable you to have a partial order on the mutexes (M1 before M2 before M3, in this example). But remember that the more you pursue this hint, the more complicated your locking becomes, and the more likely you are to have some unsynchronized access to shared data. Having your program deadlock is almost always a preferable risk to having your program give the wrong answer.

Poor performance through lock conflicts

Assuming that you have arranged your program to have enough mutexes that all the data is protected, and a fine enough granularity that it does not deadlock, the remaining mutex problems to worry about are all performance problems.

Whenever a thread is holding a mutex, it is potentially stopping another thread from making progress—if the other thread blocks on the mutex. If the first thread can use all the machine’s resources, that is probably fine. But if the first thread, while holding the mutex, ceases to make progress (for example by blocking on another mutex, or by taking a page fault, or by waiting for an i/o device), then the total throughput of your program is degraded. The problem is worse on a multi-processor, where no single thread can utilize the entire machine; here if you cause another thread to block, it might mean that a processor goes idle. In general, to get good performance you must arrange that lock conflicts are rare events. The best way to reduce lock conflicts is to lock at a finer granularity; but this introduces complexity. There is no way out of this dilemma—it is a trade-off inherent in concurrent computation.

The most typical example where locking granularity is important is in a module that manages a set of objects, for example a set of open buffered files. The simplest strategy is to use a single mutex for all the operations: open, close, read, write, and so forth. But this would prevent multiple writes on separate files proceeding in parallel, for no good reason. So a better strategy is to use one lock for operations on the global list of open files, and one lock per open file for operations affecting only that file. This can be achieved by associating a mutex with the record representing each open file. The code might look something like the following.

```
TYPE File = REF RECORD m: Thread.Mutex; .... END;
VAR globalLock: Thread.Mutex;
VAR globalTable: ....;

PROCEDURE Open(name: String): File;
BEGIN
    LOCK globalLock DO
        .... (* access globalTable *)
    END;
END Open;

PROCEDURE Write(f: File, ....);
BEGIN
    LOCK f^.m DO
        .... (* access fields of f *)
    END;
END Write;
```

In even more complicated situations, you might protect different parts of a record with different mutexes. You might also have some immutable fields in the record that need no protection at all. In these situations, though, you must be careful about how the record is laid out in memory. If your language allows you to pack fields into records sufficiently tightly that they cannot be accessed by atomic operations on your computer's memory, and if you protect such packed fields with different mutexes, you might get the wrong answer. This is because the generated instructions for modifying such fields involve reading them, modifying them in a register (or on-stack), then writing them back. If two threads are doing this concurrently for two fields that occupy the same memory word, you might get the wrong result. You need to be especially conscious of this potential bug if you over-ride the language's default algorithm for laying out fields of records. Ideally, your programming language will prevent this bug, but it probably won't.

There is an interaction between mutexes and the thread scheduler that can produce particularly insidious performance problems. The scheduler is the part of the thread implementation (often part of the operating system) that decides which of the non-blocked threads should actually be given a processor to run on. Generally the scheduler makes its decision based on a *priority* associated with each thread. (Depending on the details of your system the priority might be fixed or dynamic, programmer assigned or computed by the scheduler. Often the algorithm for deciding who to run is not specified at all.) Lock conflicts can lead to a situation where some high priority thread never makes progress at all, despite the fact that its high priority indicates that it is more urgent than the threads actually running.

This can happen, for example, in the following scenario on a uni-processor. Thread A is high priority, thread B is medium priority and thread C is low priority. The sequence of events is:

C is running (e.g. because A and B are blocked somewhere);
C locks mutex M;
B wakes up and pre-empts C
    (i.e. B runs instead of C since B has higher priority);
B embarks on some very long computation;
A wakes up and pre-empts B (since A has higher priority);
A tries to lock M;
A blocks, and so the processor is given back to B;
B continues its very long computation.

The net effect is that a high priority thread (A) is unable to make progress even though the processor is being used by a medium priority thread (B). This state is stable until there is processor time available for the low priority thread C to complete its work and unlock M.

The programmer can avoid this problem by arranging for C to raise its priority before locking M. But this can be quite inconvenient, since it involves considering for each mutex which other thread priorities might be involved. The real solution of this problem lies with the implementer of your threads facility. He must somehow communicate to the scheduler that since A is blocked on M, the thread holding M should be viewed as having at least as high a priority as A. Unfortunately, your implementer has probably failed to do this—we don't do it in the SRC implementation.

Releasing the mutex within a LOCK clause

There are times when you want to unlock the mutex in some region of program nested inside a LOCK clause. For example, you might want to unlock the mutex before calling down to a lower level abstraction that will block or execute for a long time (in order to avoid provoking delays for other threads that want to lock the mutex). The SRC system provides for this usage by offering the raw operations "Acquire(m)" and "Release(m)" in a lower-level, less-advertised, interface. You must exercise extra care if you take advantage of this. First, you must be sure that the operations are correctly bracketed, even in the presence of exceptions. Second, you must be prepared for the fact that the state of the monitor's data might have changed while you had the mutex unlocked. This can be tricky if you called "Release" explicitly (instead of just ending the LOCK clause) because you were imbedded in some flow control construct such as a conditional clause. Your program counter might now depend on the previous state of the monitor's data, implicitly making a decision that might no longer be valid. So SRC discourages this paradigm, to reduce the tendency to introduce quite subtle bugs.

One other use of separate calls of "Acquire(m)" and "Release(m)" sometimes arises in the vicinity of forking. You might be executing with a mutex held and want to fork a new thread to continue working on the protected data, while the original thread continues without further access to the data. In other words, you would like to transfer the holding of the mutex to the newly forked thread, atomically. You can achieve this by locking the mutex with “Acquire(m)” instead of LOCK, and later calling “Release(m)” in the forked thread. This tactic is quite dangerous—it is difficult to verify the correct functioning of the mutex. Additionally, many thread facilities keep track of which thread is holding a mutex—in those systems this tactic probably will not work. I recommend that you don’t do this.

USING A CONDITION VARIABLE: SCHEDULING SHARED RESOURCES

A condition variable is used when the programmer wants to schedule the way in which multiple threads access some shared resource, and the simple one-at-a-time mutual exclusion provided by mutexes is not sufficient.

Consider the following example, where one or more producer threads are passing data to one or more consumers. The data is transferred through an unbounded buffer formed by a linked list whose head is the global variable “head”. If the linked list is empty, the consumer blocks on the condition variable “nonEmpty” until the producer generates some more data. The list and the condition variable are protected by the mutex “m”.

```
VAR m: Thread.Mutex;
VAR head: List;

PROCEDURE Consume(): List;
    VAR topElement: List;
BEGIN
    LOCK m DO
        WHILE head = NIL DO Thread.Wait(m, nonEmpty) END;
        topElement := head;
        head := head^.next;
    END;
    RETURN topElement
END Consume;

PROCEDURE Produce(newElement: List);
BEGIN
    LOCK m DO
        newElement^.next := head;
        head := newElement;
        Thread.Signal(nonEmpty);
    END;
END Produce;
```

This is fairly straightforward, but there are still some subtleties. Notice that when a consumer returns from the call of “Wait” his first action after re-locking the mutex is to check once more whether the linked list is empty. This is an example of the following general pattern, which I strongly recommend for all your uses of condition variables.

```
WHILE NOT expression DO Thread.Wait(m,c) END;
```

You might think that re-testing the expression is redundant: in the example above, the producer made the list non-empty before calling “Signal”. But the semantics of “Signal” do not guarantee that the awoken thread will be the next to lock the mutex. It is possible that some other consumer thread will intervene, lock the mutex, remove the list element and unlock the mutex, before the newly awoken thread can lock the mutex.[5] A second reason for re-checking is local to the SRC design: we decided to permit the implementation of “Signal” to (rarely) awaken more than one thread, because this allowed us to generate more efficient code for the “Wait” and “Signal” primitives.

But the main reason for advocating use of this pattern is to make your program more obviously, and more robustly, correct. With this style it is immediately clear that the “expression” is true before the following statements are executed. Without it, this fact could be verified only by looking at all the places that might signal the condition variable. In other words, this programming convention allows you to verify correctness by local inspection, which is always preferable to global inspection.

A final advantage of this convention is that it allows for simple programming of calls to “Signal” or “Broadcast”—extra wake-ups are benign. Carefully coding to ensure that only the correct threads are awoken is now only a performance question, not a correctness one (but of course you must ensure that at least the correct threads are awoken).

Using “Broadcast”

The “Signal” primitive is useful if you know that at most one thread can usefully be awoken. “Broadcast” awakens all threads that have called “Wait”. If you always program in the recommended style of re-checking an expression after return from “Wait”, then the correctness of your program will be unaffected if you replace calls of “Signal” with calls of “Broadcast”.

One use of “Broadcast” is when you want to simplify your program by awakening multiple threads, even though you know that not all of them can make progress. This allows you to be less careful about separating different wait reasons into different condition variables. This use trades slightly poorer performance for greater simplicity. Another use of “Broadcast” is when you really need to awaken multiple threads, because the resource you have just made available can be used by multiple other threads.

A simple example where “Broadcast” is useful is in the scheduling policy known as shared/exclusive locking (or readers/writers locking). Most commonly this is used when you have some shared data being read and written by various threads: your algorithm will be correct (and perform better) if you allow multiple threads to read the data concurrently, but a thread modifying the data must do so when no other thread is accessing the data.

The following procedures implement this scheduling policy. Any thread wanting to read your data calls “AcquireShared”, then reads the data, then calls “ReleaseShared”. Similarly any thread wanting to modify the data calls “AcquireExclusive”, then modifies the data, then calls “ReleaseExclusive”. When the variable “i” is greater than zero, it counts the number of active readers. When it is negative there is an active writer. When it

[5] The condition variables described here are not the same as those originally described by Hoare [9]. Hoare’s design would indeed provide a sufficient guarantee to make this re-testing redundant. But the design given here appears to be preferable, since it permits a much simpler implementation, and the extra check is not usually expensive.

is zero, no thread is using the data. If a potential reader inside “AcquireShared” finds that “i” is less than zero, it must block until the writer calls “ReleaseExclusive”.

```
VAR i: INTEGER;
VAR m: Thread.Mutex;
VAR c: Thread.Condition;

PROCEDURE AcquireExclusive();
BEGIN
    LOCK m DO
        WHILE i # 0 DO Thread.Wait(m,c) END;
        i := -1;
    END;
END AcquireExclusive;

PROCEDURE AcquireShared();
BEGIN
    LOCK m DO
        WHILE i < 0 DO Thread.Wait(m,c) END;
        i := i+1;
    END;
END AcquireShared;

PROCEDURE ReleaseExclusive();
BEGIN
    LOCK m DO
        i := 0; Thread.Broadcast(c);
    END;
END ReleaseExclusive;

PROCEDURE ReleaseShared();
BEGIN
    LOCK m DO
        i := i-1;
        IF i = 0 THEN Thread.Signal(c) END;
    END;
END ReleaseShared;
```

Using “Broadcast” is convenient in “ReleaseExclusive”, because a terminating writer does not need to know how many readers are now able to proceed. But notice that you could re-code this example using just “Signal”, by adding a counter of how many readers are waiting, and calling “Signal” that many times in “ReleaseExclusive”. The “Broadcast” facility is just a convenience, taking advantage of information already available to the threads implementation. Notice that there is no reason to use “Broadcast” in “ReleaseShared”, because we know that at most one blocked writer can usefully make progress.

This particular encoding of shared/exclusive locking exemplifies many of the problems that can occur when using condition variables, as we will see in the following

sections. As we discuss these problems, I will present revised encodings of this locking paradigm.

Spurious wake-ups

If you keep your use of condition variables very simple, you might introduce the possibility of awakening threads that cannot make useful progress. This can happen if you use “Broadcast” when “Signal” would be sufficient, or if you have threads waiting on a single condition variable for multiple different reasons. For example, the shared/exclusive locking procedures shown earlier use just one condition variable for readers as well as writers. This means that when we call “Broadcast” in “ReleaseExclusive”, the effect will be to awaken both classes of blocked threads. But if a reader is first to lock the mutex, it will increment “i” and prevent an awoken potential writer from making progress until the reader later calls “ReleaseShared”. The cost of this is extra time spent in the thread scheduler, which is typically an expensive place to be. If your problem is such that these spurious wake-ups will be common, and unless your scheduler is unusually efficient, you should probably separate the blocked threads onto two condition variables—one for readers and one for writers. A terminating reader need only signal the writers’ condition variable; a terminating writer would signal one of them, depending on which was non-empty. With this change, the procedures would look as follows.

```
VAR i: INTEGER;
VAR m: Thread.Mutex;
VAR cR, cW: Thread.Condition;
VAR readWaiters: INTEGER;

PROCEDURE AcquireExclusive();
BEGIN
    LOCK m DO
        WHILE i # 0 DO Thread.Wait(m,cW) END;
        i := -1;
    END;
END AcquireExclusive;

PROCEDURE AcquireShared();
BEGIN
    LOCK m DO
        readWaiters := readWaiters+1;
        WHILE i < 0 DO Thread.Wait(m,cR) END;
        readWaiters := readWaiters-1;
        i := i+1;
    END;
END AcquireShared;
```

```
PROCEDURE ReleaseExclusive();
BEGIN
    LOCK m DO
        i := 0;
        IF readWaiters > 0 THEN
            Thread.Broadcast(cR);
        ELSE
            Thread.Signal(cW);
        END;
    END;
END ReleaseExclusive;

PROCEDURE ReleaseShared();
BEGIN
    LOCK m DO
        i := i-1;
        IF i = 0 THEN Thread.Signal(cW) END;
    END;
END ReleaseShared;
```

Spurious lock conflicts

The straightforward use of condition variables can lead to excessive scheduling overhead. In the reader/writer example, when a terminating reader inside “ReleaseShared” calls “Signal”, it still has the mutex locked. On a uni-processor this would often not be a problem, but on a multi-processor the effect is liable to be that a potential writer is awakened inside “Wait”, executes a few instructions, and then blocks trying to lock the mutex—because it is still held by the terminating reader, executing concurrently. A few microseconds later the terminating reader unlocks the mutex, allowing the writer to continue. This has cost us two extra re-schedule operations, which is a significant expense.

This is a common situation, and it has a simple solution. Since the terminating reader does not access the data protected by the mutex after the call of “Signal”, we can move that call to after the end of the lock clause, as follows. Notice that accessing “i” is still protected by the mutex.

```
PROCEDURE ReleaseShared();
    VAR doSignal: BOOLEAN;
BEGIN
    LOCK m DO
        i := i-1;
        doSignal := (i=0);
    END;
    IF doSignal THEN Thread.Signal(cW) END;
END ReleaseShared;
```

There is a more complicated case of spurious lock conflicts when a terminating writer calls “Broadcast”. First, it does so with the mutex held. But also, only one of the waiting readers at a time can lock the mutex to re-check and increment “i”, so on a multi-processor other awoken readers are liable to block trying to lock the mutex (this is quite unlikely on a uni-processor). If necessary, we can correct this by awakening just one reader in “ReleaseExclusive” (by calling “Signal” instead of “Broadcast”), and having each reader in turn awaken the next, as follows.

```
PROCEDURE AcquireShared();
BEGIN
    LOCK m DO
        readWaiters := readWaiters+1;
        WHILE i < 0 DO Thread.Wait(m,cR) END;
        readWaiters := readWaiters-1;
        i := i+1;
    END;
    Thread.Signal(cR);
END AcquireShared;
```

Starvation

Whenever you have a program that is making scheduling decisions, you must worry about how fair these decisions are; in other words, are all threads equal or are some more favored? When you are using a mutex, this consideration is dealt with for you by the threads implementation—typically by a first-in-first-out rule for each priority level. Mostly, this is also true for condition variables. But sometimes the programmer must become involved. The most extreme form of unfairness is “starvation”, where some thread will *never* make progress. This can arise in our reader-writer locking example (of course). If the system is heavily loaded, so that there is always at least one thread wanting to be a reader, the existing code will starve writers. This would occur with the following pattern.

```
Thread A calls “AcquireShared”; i := 1;
Thread B calls “AcquireShared”; i := 2;
Thread A calls “ReleaseShared”; i := 1;
Thread C calls “AcquireShared”; i := 2;
Thread B calls “ReleaseShared”; i := 1;
etc.
```

Since there is always an active reader, there is never a moment when a writer can proceed; potential writers will always remain blocked, waiting for “i” to reduce to 0. If the load is such that this is really a problem, we need to make the code yet more complicated. For example, we could arrange that a new reader would defer inside “AcquireShared” if there was a blocked potential writer. We could do this by adding a counter for blocked writers, as follows.

```
VAR writeWaiters: INTEGER;

PROCEDURE AcquireShared();
BEGIN
```

```
    LOCK m DO
        readWaiters := readWaiters+1;
        IF writeWaiters > 0 THEN
            Thread.Signal(cW);
            Thread.Wait(m,cR);
        END;
        WHILE i < 0 DO Thread.Wait(m,cR) END;
        readWaiters := readWaiters-1;
        i := i+1;
    END;
    Thread.Signal(cR);
END AcquireShared;

PROCEDURE AcquireExclusive();
BEGIN
    LOCK m DO
        writeWaiters := writeWaiters+1;
        WHILE i # 0 DO Thread.Wait(m,cW) END;
        writeWaiters := writeWaiters-1;
        i := -1;
    END;
END AcquireExclusive;
```

There is no limit to how complicated this can become, implementing ever more elaborate scheduling policies. The programmer must exercise restraint, and only add such features if they are really required by the actual load on the resource.

Complexity

As you can see, worrying about these spurious wake-ups, lock conflicts and starvation makes the program more complicated. The first solution of the reader/writer problem that I showed you had 15 lines inside the procedure bodies; the final version had 30 lines, and some quite subtle reasoning about its correctness. You need to consider, for each case, whether the potential cost of ignoring the problem is enough to merit writing a more complex program. This decision will depend on the performance characteristics of your threads implementation, on whether you are using a multi-processor, and on the expected load on your resource. In particular, if your resource is mostly *not* in use then the performance effects will not be a problem, and you should adopt the simplest coding style.

Usually, I find that moving the call of "Signal" to beyond the end of the LOCK clause is easy and worth the trouble, and that the other performance enhancements are not worth making. But sometimes they are important, and you should only ignore them after explicitly considering whether they are required in your particular situation.

Deadlock

You can introduce deadlocks by using condition variables. For example, if you have two resources (call them (1) and (2)), the following sequence of actions produces a deadlock.

```
Thread A acquires resource (1);
Thread B acquires resource (2);
Thread A wants (2), so it waits on (2)'s condition variable;
Thread B wants (1), so it waits on (1)'s condition variable.
```

Deadlocks such as this are not significantly different from the ones we discussed in connection with mutexes. You should arrange that there is a partial order on the resources managed with condition variables, and that each thread wishing to acquire multiple resources does so according to this order. So, for example, you might decide that (1) is ordered before (2). Then thread B would not be permitted to try to acquire (1) while holding (2), so the deadlock would not occur.

One interaction between condition variables and mutexes is a subtle source of deadlock. Consider the following two procedures.

```
VAR a, b: Thread.Mutex;
VAR c: Thread.Condition;
VAR ready: BOOLEAN;

PROCEDURE Get();
BEGIN
    LOCK a DO
        LOCK b DO
            WHILE NOT ready DO Thread.Wait(b,c) END;
        END;
    END;
END Get;

PROCEDURE Give();
BEGIN
    LOCK a DO
        LOCK b DO
            ready := TRUE; Thread.Signal(c);
        END;
    END;
END Give;
```

If "ready" is FALSE and thread A calls "Get", it will block on a call of "Wait(b,c)". This unlocks "b", but leaves "a" locked. So if thread B calls "Give", intending to cause a call of "Signal(c)", it will instead block trying to lock "a", and your program will have deadlocked. Clearly, this example is trivial, since mutex "a" does not protect any data (and the potential for deadlock is quite apparent anyway), but the overall pattern does occur.

Most often this problem occurs when you lock a mutex at one abstraction level of your program then call down to a lower level, which (unknown to the higher level) blocks. If this block can be freed only by a thread that is holding the higher level mutex, you will deadlock. It is generally risky to call into a lower level abstraction while holding one of your mutexes, unless you understand fully the circumstances under which the called procedure might block. One solution here is to explicitly unlock the mutex before calling the lower level abstraction, as we discussed earlier; but as we discussed, this solution has its own dangers. A better solution is to arrange to end the LOCK clause before calling down. You can find further discussions of this problem, known as the “nested monitor problem”, in the literature [8].

USING FORK: WORKING IN PARALLEL

As we discussed earlier, there are several classes of situations where you will want to fork a thread: to utilize a multi-processor; to do useful work while waiting for a slow device; to satisfy human users by working on several actions at once; to provide network service to multiple clients simultaneously; and to defer work until a less busy time.

It is quite common to find straightforward application programs using several threads. For example, you might have one thread doing your main computation, a second thread writing some output to a file, a third thread waiting for (or responding to) interactive user input, and a fourth thread running in background to clean up your data structures (for example, re-balancing a tree). In the programs we build at SRC, several of our library packages fork threads internally.

When you are programming with threads, you usually drive slow devices through synchronous library calls that suspend the calling thread until the device action completes, but allow other threads in your address space to continue. You will find no need to use older schemes for asynchronous operation (such as interrupts, Unix signals or VMS AST’s). If you don’t want to wait for the result of a device interaction, invoke it in a separate thread. If you want to have multiple device requests outstanding simultaneously, invoke them in multiple threads. If your operating system still delivers some asynchronous events through these older mechanisms, the runtime library supporting your threads facility should convert them into more appropriate mechanisms. See, for example, the design of the Topaz system calls [11] or the exception and trapping machinery included with Sun’s lightweight process library [14,15].

If your program is interacting with a human user, you will usually want it to be responsive even while it is working on some request. This is particularly true of window-oriented interfaces. It is particularly infuriating to the user if the interactive display goes dumb just because the database query is taking a long time. You can achieve responsiveness by using extra threads. Often, the designer of your window system will have already done this for you, and will always call your program in a separate thread. At other times, the window system will call your program in a single thread synchronously with the user input event. In this latter case, you must decide whether the requested action is short enough to do it synchronously, or whether you should fork a thread. The complexity introduced by using forked threads here is that you need to exercise a lot of care in accessing data from the interactive interface (for example, the value of the current selection, or the contents of editable text areas) since these values might change once you start executing asynchronously. This is a difficult design issue, and each window system tackles it differently. I have not yet seen a totally satisfactory design.

Network servers are usually required to service multiple clients concurrently. If your network communication is based on RPC [3], this will happen without any work on your part, since the server side of your RPC system will invoke each concurrent incoming call in a separate thread, by forking a suitable number of threads internally to its implementation. But you can use multiple threads even with other communication paradigms. For example, in a traditional connection-oriented protocol (such as file transfer

layered on top of TCP), you should probably fork one thread for each incoming connection. Conversely, if you are writing a client program and you don’t want to wait for the reply from a network server, invoke the server from a separate thread.

The technique of adding threads in order to defer work is quite valuable. There are several variants of the scheme. The simplest is that as soon as your procedure has done enough work to compute its result, you fork a thread to do the remainder of the work, and then return to your caller in the original thread. This reduces the latency of your procedure (the elapsed time from being called to returning), in the hope that the deferred work can be done more cheaply later (for example, because a processor goes idle). The disadvantage of this simplest approach is that it might create large numbers of threads, and it incurs the cost of calling “Fork” each time. Often, it is preferable to keep a single housekeeping thread and feed requests to it. It’s even better when the housekeeper doesn't need any information from the main threads, beyond the fact that there is work to be done. For example, this will be true when the housekeeper is responsible for maintaining a data structure in an optimal form, although the main threads will still get the correct answer without this optimization. An additional technique here is to program the housekeeper either to merge similar requests into a single action, or to restrict itself to run not more often than a chosen periodic interval.

On a multi-processor you will want to use “Fork” in order to utilize as many of your processors as you can. There isn’t much general advice I can give here—mostly, the decisions about when and what to fork are too problem-specific. One general technique is *pipelining*, which I discuss in the next section.

Pipelining

On a multi-processor, there is one specialized use of additional threads that is particularly valuable. You can build a chain of producer-consumer relationships, known as a *pipeline*. For example, when thread A initiates an action, all it does is enqueue a request in a buffer. Thread B takes the action from the buffer, performs part of the work, then enqueues it in a second buffer. Thread C takes it from there and does the rest of the work. This forms a three-stage pipeline. The three threads operate in parallel except when they synchronize to access the buffers, so this pipeline is capable of utilizing up to three processors. At its best, pipelining can achieve almost linear speed-up and can fully utilize a multi-processor. A pipeline can also be useful on a uni-processor if each thread will encounter some real-time delays (such as page faults, device handling or network communication).

For example, the following program fragment implements a simple three stage pipeline. An action is initiated by calling “PaintChar”. One auxiliary thread executes in “Rasterize” and another in “Painter”. The pipeline stages communicate through unbounded buffers implemented as linked lists whose last elements are “rasterTail” and “paintTail”. The initial values of the tails are dummy elements, to make the program simpler.

```
TYPE RasterList = REF RECORD ch: CHAR; next: RasterList END;
TYPE PaintList = REF RECORD r: Bitmap; next: PaintList END;
VAR rasterTail: RasterList;
VAR paintTail: PaintList;
VAR m: Thread.Mutex;
VAR c1, c2: Thread.Condition;
```

```
PROCEDURE PaintChar(arg: CHAR);
    VAR this: RasterList;
BEGIN
    NEW(this);
    this^.ch := arg; this^.next := NIL;
    (* Enqueue request for Rasterize thread .... *)
    LOCK m DO
        rasterTail^.next := this; rasterTail := this;
    END;
    Thread.Signal(c1);
    (* Now return to our caller .... *)
END PaintChar;

PROCEDURE Rasterize(init: RasterList);
    VAR last: RasterList;
    VAR this: PaintList;
BEGIN
    last := init;
    LOOP
        (* Wait for RasterList request and dequeue it .... *)
        LOCK m DO
            WHILE last^.next = NIL DO Thread.Wait(m, c1); END;
            last := last^.next;
        END;
        (* Convert character to bitmap .... *)
        NEW(this);
        this^.bitmap := Font.Map(last^.ch); this^.next := NIL;
        (* Enqueue request for painter thread .... *)
        LOCK m DO
            paintTail^.next := this; paintTail := this;
        END;
        Thread.Signal(c2);
    END;
END Rasterize;

PROCEDURE Painter(init: PaintList);
    VAR last: PaintList;
BEGIN
    last := init;
    LOOP
        (* Wait for PaintList request and dequeue it .... *)
        LOCK m DO
            WHILE last^.next = NIL DO Thread.Wait(m, c2); END;
            last := last^.next;
        END;
```

```
            (* Paint the bitmap .... *)
            Display.PaintBitmap(last^.bitmap);
        END;
    END Painter;

    NEW(rasterTail); rasterTail.next := NIL;
    NEW(paintTail); paintTail.next := NIL;
    Thread.Fork(Rasterize, rasterTail);
    Thread.Fork(Painter, paintTail);
```

There are two problems with pipelining. First, you need to be careful about how much of the work gets done in each stage. The ideal is that the stages are equal: this will provide maximum throughput, by utilizing all your processors fully. Achieving this ideal requires hand tuning, and re-tuning as the program changes. Second, the number of stages in your pipeline determines statically the amount of concurrency. If you know how many processors you have, and exactly where the real-time delays occur, this will be fine. For more flexible or portable environments it can be a problem. Despite these problems, pipelining is a powerful technique that has wide applicability.

The impact of your environment

The design of your operating system and runtime libraries will affect the extent to which it is desirable or useful to fork threads. Your operating system should not suspend the entire address space just because one thread is blocked for an i/o request (or for a page fault). Your operating system and your libraries must permit calls from multiple threads in parallel. Generally, in a well-designed environment for supporting multi-threaded programs you will find that the facilities of your operating system and libraries are available as synchronous calls that block only the calling thread [11].

You will need to know some of the performance parameters of your threads implementation. What is the cost of creating a thread? What is the cost of keeping a blocked thread in existence? What is the cost of a context switch? What is the cost of a LOCK clause when the mutex is not locked? Knowing these, you will be able to decide to what extent it is feasible or useful to add extra threads to your program.

Potential problems with adding threads

You need to exercise a little care in adding threads, or you will find that your program runs slower instead of faster.

If you have significantly more threads ready to run than there are processors, you will usually find that your performance degrades. This is partly because most thread schedulers are quite slow at making general re-scheduling decisions. If there is a processor idle waiting for your thread, the scheduler can probably get it there quite quickly. But if your thread has to be put on a queue, and later swapped into a processor in place of some other thread, it will be more expensive. A second effect is that if you have lots of threads running they are more likely to conflict over mutexes or over the resources managed by your condition variables.

Mostly, when you add threads just to improve your program's structure (for example driving slow devices, or responding to mouse clicks speedily, or for RPC invocations) you will not encounter this problem; but when you add threads for performance purposes (such as performing multiple actions in parallel, or deferring work, or utilizing multi-processors), you will need to worry whether you are overloading the system.

But let me stress that this warning applies only to the threads that are ready to run. The expense of having threads blocked on condition variables is usually less significant, being just the memory used for scheduler data structures and the thread stack. The programs at SRC often have quite a large number of blocked threads. (50 is not uncommon in application programs; there are usually hundreds blocked inside the operating system—even on personal workstations.)

In most systems the thread creation and termination facilities are not cheap. Your threads implementation will probably take care to cache a few terminated thread carcasses, so that you don’t pay for stack creation on each fork, but nevertheless a call of “Fork” will probably incur a total cost of about two or three re-scheduling decisions. So you shouldn’t fork too small a computation into a separate thread. One useful measure of a threads implementation on a multi-processor is the smallest computation for which it is profitable to fork a thread.

Despite these cautions, be aware that our experience at SRC running multi-threaded applications on a 5-way multi-processor has been that programmers are as likely to err by creating too few threads as by creating too many.

USING ALERT: DIVERTING THE FLOW OF CONTROL

The purpose of alerts is to cause termination of a long running computation or a long-term wait. For example, on a multi-processor it might be useful to fork multiple competing algorithms to solve the same problem, and when the first of them completes you abort the others. Or you might embark on a long computation (e.g. a database query), but abort it if the user clicks a CANCEL button.

The programming convention we use at SRC is that any procedure in a public interface that might incur a long computation or a long-term wait should be alertable. In other words, a long computation should occasionally call “TestAlert” and long-term waits should be calls of “AlertWait” instead of “Wait”. In this context “long” means long enough to upset a human user. The attraction of this convention is that you can be sure that the user can always regain control of the application program. The disadvantage is that programs calling these procedures must be prepared for the “Alerted” exception to come out of them. This convention is less rigorously applied when a single entry point occasionally (but not always) causes a long-term wait.

Another programming convention we have is that you should only alert a thread if you forked the thread. For example, a package should not alert a caller’s thread that happens to be executing inside the package. This convention allows you to view an alert as an indication that the thread should terminate completely.

The problem with alerts (or any other form of asynchronous interrupt mechanism, such as Apollo’s “task_$signal”) is that they are, by their very nature, intrusive. Using them will tend to make your program less well structured. A straightforward-looking flow of control in one thread can suddenly be diverted because of an action initiated by another thread. This is another example of a facility that makes it harder to verify the correctness of a piece of program by local inspection. Unless alerts are used with great restraint, they will make your program unreadable, unmaintainable, and perhaps incorrect.

There are alternatives to using alerts. If you know which condition variable a thread is blocked on, you can more simply prod it by setting a boolean flag and signalling the condition variable. A package could provide additional entry points whose purpose is to prod a thread blocked inside the package on a long-term wait.

Alerts are most useful when you don’t know exactly what is going on. For example, the target thread might be blocked in any of several packages, or within a single package it might be blocked on any of several condition variables. In these cases an alert is certainly the best solution. Even when other alternatives are available, it might be best to use alerts just because they are a single unified scheme for provoking thread termination.

There is no consensus yet among the designers of thread facilities about how to tackle this problem. Each designer has his own solution, and each solution so far has its own problems.

ADDITIONAL TECHNIQUES

Most of the programming paradigms for using threads are quite simple. I’ve described several of them earlier; you will discover many others as you gain experience. A few of the useful techniques are much less obvious. This section describes some of these less obvious ones.

Up-calls

Most of the time most programmers build their programs using layered abstractions. Higher level abstractions call only lower level ones, and abstractions at the same level do not call each other. All actions are initiated at the top level.

This methodology carries over quite well to a world with concurrency. You can arrange that each thread will honor the abstraction boundaries. Permanent dæmon threads within an abstraction initiate calls to lower levels, but not to higher levels. The abstraction layering has the added benefit that it forms a partial order, and this order is sufficient to prevent deadlocks when locking mutexes, without any additional care from the programmer.

This purely top-down layering is not satisfactory when actions that affect high-level abstractions can be initiated at a low layer in your system. One frequently encountered example of this is on the receiving side of network communications. Other examples are user input, and spontaneous state changes in peripheral devices such as disks and tapes.

Consider the example of a communications package dealing with incoming packets from a network. Here there are typically three or more layers of dispatch (corresponding to the data link, network and transport layers in OSI terminology). If you try to maintain a top-down calling hierarchy, you will find that you incur a context switch in each of these layers. The thread that wishes to receive data from its transport layer connection cannot be the thread that dispatches an incoming ethernet packet, since the ethernet packet might belong to a different connection, or a different protocol (for example, UDP instead of TCP), or a different protocol family altogether (for example, DECnet instead of IP). Many implementers have tried to maintain this layering for packet reception, and the effect has been uniformly bad performance—dominated by the cost of context switches.

The alternative technique is known as “up-calls” [5]. In this methodology, you maintain a pool of threads willing to receive incoming data link layer (e.g. ethernet) packets. The receiving thread dispatches on ethernet protocol type and calls *up* to the network layer (e.g. DECnet or IP), where it dispatches again and calls up to the transport layer (e.g. TCP), where there is a final dispatch to the appropriate connection. In some systems, this up-call paradigm extends into the application. The attraction here is high performance: there are no unnecessary context switches. All the top-performing network implementations are structured this way.

You do pay for this performance. As usual, the programmer's task has been made more complicated. Partly this is because each layer now has an up-call interface as well as the traditional down-call interface. But also the synchronization problem has become more delicate. In a purely top-down system it is fine to hold one layer's mutex while calling a lower layer (unless the lower layer might block on a condition variable and cause the sort of nested monitor deadlock we discussed earlier). But in the presence of up-calls this can easily provoke a deadlock involving just the mutexes—if an up-calling thread holding a lower level mutex needs to lock the higher level one. In other words, the presence of up-calls makes it more likely that you will violate the partial order rule for locking mutexes. To avoid this, you should generally avoid holding a mutex while making an up-call (but this is easier said than done).

Version stamps

Sometimes concurrency can make it more difficult to use cached information. This can happen when a separate thread executing at a lower level in your system invalidates some information known to a thread currently executing at a higher level. For example, information about a disk volume might change—either because of hardware problems or because the volume has been removed and replaced. You can use up-calls to invalidate cache structures at the higher level, but this will not invalidate state held locally by a thread. In the most extreme example, a thread might obtain information from a cache, and be about to call an operation at the lower level. Between the time the information comes from the cache and the time that the call actually occurs, the information might have become invalid.

A technique known as "version stamps" can be useful here. In the low level abstraction you maintain a counter associated with the true data. Whenever the data changes, you increment the counter. (Assume the counter is large enough to never overflow.) Whenever a copy of some of the data is issued to a higher level, it is accompanied by the current value of the counter. If higher level code is caching the data, it caches the associated counter value too. Whenever you make a call back down to the lower level, and the call or its parameters depend on previously obtained data, you include the associated value of the counter. When the low level receives such a call, it compares the incoming value of the counter with the current truth value. If they are different it returns an exception to the higher level, which then knows to re-consider its call. (Sometimes, you can provide the new data with the exception). Incidentally, this technique is also useful when maintaining cached data across a distributed system.

Work crews

There are situations that are best described as "an embarrassment of parallelism", when you can structure your program to have vastly more concurrency than can be efficiently accommodated on your machine. For example, a compiler implemented using concurrency

might be willing to use a separate thread to compile each procedure, or even each statement. In such situations, if you fork one thread for each action you will create so many threads that the scheduler becomes quite inefficient, or so many that you have numerous lock conflicts, or so many that you run out of memory for the stacks.

Your choice here is either to be more restrained in your forking, or to use an abstraction that will control your forking for you. Such an abstraction is described in Vandevoorde and Roberts' paper [17]. The basic idea is to enqueue requests for asynchronous activity and have a fixed pool of threads that perform the requests. The complexity comes in managing the requests, synchronizing between them, and co-ordinating the results. See the paper for a full description.

An alternative proposal, which SRC has not yet explored, is to implement "Fork" in such a way that it defers actually creating the new thread until there is a processor available to run it. We call this proposal "lazy forking", but we have not pursued it any further.

BUILDING YOUR PROGRAM

A successful program must be useful, correct, live (as defined below) and efficient. Your use of concurrency can impact each of these. I have discussed quite a few techniques in the previous sections that will help you. But how will you know if you have succeeded? The answer is not clear, but this section might help you towards discovering it.

The place where concurrency can affect usefulness is in the design of the interfaces to library packages. You must design your interfaces with the assumption that your callers will be using multiple threads. This means that you must ensure that all the entry points are thread re-entrant (i.e. they can be called by multiple threads simultaneously), even if this means that each procedure immediately locks a central mutex. You must not return results in shared global variables, nor in global statically allocated storage. Your calls should be synchronous, not returning until their results are available—if your caller wants to do other work meanwhile, he can do it in other threads. Even if you don't presently have any multi-threaded clients of the interface, I strongly recommend that you follow these guidelines so that you will avoid problems in the future.

By "correct" I mean that if your program eventually produces an answer, it will be the one defined by its specification. Your programming environment is unlikely to provide much help here beyond what it already provides for sequential programs. Mostly, you must be fastidious about associating each piece of data with one (and only one) mutex. If you don't pay constant attention to this, your task will be hopeless. If you use mutexes correctly, and you always use condition variables in the recommended style (re-testing the boolean expression after returning from "Wait"), then you are unlikely to go wrong.

By "live", I mean that your program will eventually produce an answer. The alternatives are infinite loops or deadlock. I can't help you with infinite loops. I believe that the hints of the preceding sections will help you to avoid deadlocks. But if you fail and produce a deadlock, it should be quite easy to detect. Your major help in analyzing a deadlock will come from a symbolic debugger. The debugger must provide at least minimal support for threads—enumerating the existing threads and looking at each thread's stack. Hopefully, your debugger will also provide some filtering in the thread enumeration, for example finding all threads that have a stack frame in a particular module, or finding all threads that are blocked on a particular mutex or condition variable. A very nice feature would be a facility to determine which thread is holding a particular mutex.

By "efficient", I mean that your program will make good use of the available computer resources, and therefore will produce its answer quickly. Again, the hints in the previous sections should help you to avoid the problem of concurrency adversely affecting your performance. And again, your programming environment needs to give you some help. Performance bugs are the most insidious of problems, since you might not even notice that you have them. The sort of information you need to obtain includes statistics on lock conflicts (for example, how often threads have had to block on this mutex, and how long they then had to wait) and on concurrency levels (for example, what was the average number of threads ready to execute in your program, or what percentage of the time were "n" threads ready).

One final warning: don't emphasize efficiency at the expense of correctness. It is much easier to start with a correct program and work on making it efficient, than to start with an efficient program and work on making it correct.

CONCLUDING REMARKS

Writing concurrent programs has a reputation for being exotic and difficult. I believe it is neither. You need a system that provides you with good primitives and suitable libraries, you need a basic caution and carefulness, you need an armory of useful techniques, and you need to know of the common pitfalls. I hope that this paper has helped you towards sharing my belief.

Butler Lampson, Mike Schroeder, Bob Stewart and Bob Taylor caused me to write this paper. If you found it useful, thank them.

REFERENCES

1. ACCETTA, M. *et al.* Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference* (July 1986), 93-112.
2. Apollo Computer Inc. *Concurrent Programming Support (CPS) Reference*. Order No. 010233, June 1987.
3. BIRRELL, A., AND NELSON, B. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2,* 1 (Feb. 1984), 39-59.
4. BIRRELL, A., GUTTAG, J., HORNING, J. AND LEVIN, R. Synchronization primitives for a multiprocessor: a formal specification. In *Proceedings of the 11th Symposium on Operating System Principles* (Nov. 1987), 94-102.
5. CLARK, D. The structuring of systems using up-calls. In *Proceedings of the 10th Symposium on Operating System Principles* (Dec. 1985), 171-180.
6. COOPER, E. AND DRAVES, R. C Threads. *Technical report CMU-CS-88-154,* Carnegie Mellon University. Computer Science Department, Pittsburgh, PA (June 1988).
7. GUTTAG, J., HORNING, J. AND WING, J. The Larch family of specification languages. *IEEE Software 2,* 5, (SEP. 1985), 24-36.
8. HADDON, B. Nested monitor calls. *Operating Systems Review 11,* 4 (Oct. 1977), 18-23.
9. HOARE, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17,* 10 (Oct.1974), 549-557.

10. LAMPSON, B AND REDELL, D. Experience with processes and monitors in Mesa. *Commun. ACM 23,* 2 (Feb.1980), 105-117.
11. MCJONES, P AND SWART, G. Evolving the UNIX system interface to support multi-threaded programs. In *Proceedings of the Winter 1989 USENIX Conference* (Feb. 1989).
12. ROVNER, P. Extending Modula-2 to build large, integrated systems. *IEEE Software 3,* 6 (Nov. 1986), 46-57.
13. SALTZER, J. Traffic control in a multiplexed computer system. Th., MAC-TR-30, MIT, Cambridge, Mass., July 1966.
14. Sun Microsystems Inc. *Sun OS 4.0 Reference Manual*, Nov. 1987, section 3L.
15. Sun Microsystems Inc. *System Services Overview*, May 1988, chapter 6.
16. TEVANIAN, A. *et al* Mach threads and the Unix kernel: the battle for control. In *Proceedings of the Summer 1987 USENIX Conference* (June 1987), 185-197.
17. VANDEVOORDE, M. AND ROBERTS, E. Workcrews: an abstraction for controlling parallelism. (Submitted for publication; copies available from Eric Roberts at SRC).

APPENDIX: OTHER THREAD DESIGNS

In the main body of the paper we were talking entirely in terms of the SRC threads design. That design is quite similar to several other, more publicly available designs. In this appendix, I compare some other designs with SRC's. Where the corresponding features of the other designs differ from SRC's, I try to give you some hints on when and how those differences will be useful or dangerous.

The descriptions here assume that you are reasonably familiar with the original documentation for each of these systems. Please read the documentation yourself before relying on any of the material in this appendix, and please accept my apologies if I have mis-represented your favorite system.

Each design (including SRC's) includes mechanisms that I haven't discussed in this paper. For example, each design has facilities for affecting scheduling decisions (by adjusting priorities, or giving hints on when a pre-emption would be useful); SRC and Sun provide facilities for associating context information with a thread; each system provides a "conditional lock" operation on a mutex. Limits on time and patience preclude exploring these here.

You should bear in mind that each of these systems (including SRC's) is equivalent, in the sense that you can (at least) reproduce one system's facilities by programming on top of another system's. You can solve any concurrent programming problem by using any of these systems. The effect of the differences is just that they make some programming paradigms (good ones or bad ones) easier, and some harder.

The following table summarizes the approximate equivalence between the major SRC facilities and those of Apollo [2], Sun [14,15] and MACH [1,6,16]. The absence of an entry means that I cannot see a reasonably equivalent facility in the documentation I have read. The following sections discuss ways in which these facilities differ from SRC's.

SRC	Apollo Domain CPS	Sun release 4.0	MACH C threads
Fork	task_$create	lwp_create	cthread_fork
Join	task_$release		cthread_join
Mutex	task_$handle_t	mon_t	mutex_t
LOCK	mutex_$lock	mon_enter	mutex_lock
... END	mutex_$unlock	mon_exit	mutex_unlock
Condition	ec2_$eventcount_t	cv_t	condition_t
Wait	ec2_$wait	cv_wait	condition_wait
Broadcast	ec2_$advance	cv_broadcast	condition_broadcast
Signal		cv_notify	condition_signal
Alert	task_$signal		

Apollo Domain Concurrent Programming Support (CPS)

Apollo allows you to specify a timeout with calls of "mutex_$lock". There seem to be two reasons why you might want to do this. First, your program might have some alternative algorithm it could use if the required mutex is being held too long. This is a technique that would be useful if lock conflicts are common. Generally, you should arrange that lock conflicts are quite rare and of short duration. You can do this by arranging your locking to be fine enough grain, each mutex protecting only the data you

actually intend to touch. But if you cannot achieve this, then you might find the timeout parameter useful. Second, you can use this timeout parameter for detecting (and even recovering from) deadlocks. This I discourage—I view deadlocks as bugs; you should detect them with a debugger and re-program to avoid them.

Apollo's level 2 eventcounts serve the same purpose as SRC's condition variables. The simplest usage of "ec2_$advance" is for a thread to lock a mutex, inspect some data, and decide that it needs to block until the data is changed by some other thread. It does so by reading the value of the eventcount (with "ec2_$read"), unlocking the mutex, and then calling "ec2_$wait" to block until the eventcount is advanced. This has the same effect as SRC's use of "Wait" and "Broadcast". Reading the value of the eventcount while the mutex is still locked achieves the same atomicity as SRC guarantees in the "Wait" call. (Indeed, this is how SRC implements that atomicity.) Apollo has no equivalent of SRC's "Signal" operation.

Apollo's eventcounts provide additional freedom, since they are not explicitly tied to a mutex or its associated data. This allows threads to block and wake up without synchronizing over shared data. This is rarely what you want to do. Almost always, the reason you know that it is profitable to awaken a thread is because you have just changed some data that thread is interested in. If you take advantage of this extra flexibility, you must be careful to prevent errors such as the following sequence of actions.

1— Thread A looks at the data and decides to block;
2— Thread A unlocks the mutex;
3— Thread B locks the mutex and changes the data;
4— Thread B calls "ec2_$advance";
5— Thread A calls "ec2_$read";
6— Thread A calls "ec2_$wait".

With this sequence thread A will not be awoken, even although the data it is wanting has changed. The intended sequence was that step (5) should happen before step (4). To guarantee this, thread A must execute (5) before (2).

Apollo allows you to wait on multiple eventcounts when you call "ec2_$wait". This is occasionally convenient. Without this feature, you must have one thread for each condition for which you are waiting. The extra functionality of the Apollo design does not seem to introduce any new problems. The only down-side is additional complexity, and scope for performance penalties, in the threads implementation.

Apollo's asynchronous fault facility provided by "task_$signal" substitutes for SRC's facility for alerting a thread. The SRC design provides only for the target thread to poll for alerts (by calling "AlertWait" or "TestAlert"); the Apollo design produces a fault at an arbitrary point in the target thread's execution, provided the thread has not inhibited faults. It is important that you don't interrupt a thread while is is executing with a mutex locked, since that would leave the shared data in an unpredictable state. Apollo enforces this restriction by inhibiting faults while a mutex is locked. The attraction of the polling model is that the flow of control of your program is more clearly visible in the source. The attraction of interrupting at an arbitrary point (outside of LOCK clauses) is that it is easier to be sure that computations can be terminated—there is less likelihood that you will introduce a bug by forgetting to call "TestAlert". These attractions are both valid, and the ideal design would allow for both.

Sun's lightweight process library

Sun's version 4.0 offers "lightweight processes" as a library [14,15]. These are not true threads—within an address space only one lightweight processes is executing at any time. But the creation and synchronization facilities are the same as you would expect them to be for real threads.

Sun allows a thread to lock a mutex recursively. In other words, while a thread is holding a mutex it can call a procedure (perhaps by recursion) that locks the same mutex. In the SRC design this would deadlock. Which is preferable depends on your intent. If you were careful to restore the monitor invariant before making the call that recursively locks the mutex, all will be fine. But if this is really a bug, because you forgot that the called procedure uses the same data, you will get the wrong answer. Additionally, if you take advantage of this feature remember that you can introduce a deadlock if you split a computation into multiple threads—what used to be a recursive acquisition of the mutex might become an acquisition by a different thread.

Sun does not provide any mechanism for alerting a thread. However, they do provide a mechanism for threads to deal with asynchronous Unix signals by calling "agt_create". Some of these signals serve the same purpose as SRC's alerts, and so these cases are easy to deal with. But remember that alerts are not really a fundamental part of threads—they are just an added convenience.

The library also includes a facility for passing messages between threads. Most designers have adopted the view that message passing is an alternative to shared memory synchronization, not an adjunct to it. I don't recommend using both mechanisms within a single program, except to the extent that you need to use Sun's message system to handle asynchronous Unix signals.

Threads in Mach

The procedures I listed in the table come from Eric Cooper's C library interface to MACH threads [6]. That interface is quite similar to SRC's. The only noticeable omission is any facility for alerting a thread. There are no substantive differences.