Indeterminism in Plan 9

Sape Mullender

Bell Laboratories
2018 Antwerp, Belgium

ABSTRACT

Many applications use concurrency to deal with indeterminism: an editor may have separate and concurrent processes to deal with mouse input, keyboard input, display output, file I/O, etc.

But writing concurrent applications is hard and writing concurrent real-time applications is even harder. The Plan 9 programming environment has a comprehensive tool set that avoids a great deal of actual concurrency in indeterministic programs. It builds on work on CSP by Tony Hoare, the language Squeek by Rob Pike, Alef by Phil Winterbottom and the thread library written by Sape Mullender and Russ Cox.

This paper describes the underlying philosophy and techniques and contains lots of tutorial examples.

1. Introduction

Living with concurrency is hard. Traditional programming methods are all based on sequential programming and our brains are not very good at reasoning about the interactions between activities occurring simultaneously.

The first thing we need to ask ourselves is why or whether things need to happen simultaneously in the first place. Perhaps it is the case that much of the complexity of concurrency can be avoided. If a computer has only a single CPU, it is obvious that there can never be real concurrency (if we disregard the concurrency deep in the operating system caused by peripheral devices accessing memory and registers). Using multiple processes running in virtual concurrency can be used — and is frequently used — as an expression of independent activity in those processes.

But there is a fine line between independence and interdependence. Consider a file server. It receives requests to read or write files from users and responds back to those users with data, metadata, acknowledgements or errors. Carrying out a request usually involves disk operations and, while those disk operations are in progress, it is a good idea to allow the server to work on other requests.

Requests are mostly independent, but sometimes they involve the same directory or file and then they become interdependent. Worse, requests all involve memory allocation and deallocation and they often involve the same physical disk drives, so undesirable dependencies between processes are created too.

Let's get back to these interdependencies in a moment and let's first look at how one would write code for such a file server. Basically, there are two ways in general use, **state machines** and **threads**.

1.1. State Machines

In the *state machine* approach, we associate a data structure with each file server request and we recognize the work we have to do as well as the periods of waiting for I/O to complete by associating each waiting period with a *state*. Completion of I/O, I/O errors, or cancellations from the requesting user are *events*.

The *state* of a request is stored in the request data structure, along with any data relevant to the request. When an event occurs, the associated request structure is found, and an action is carried out that depends on the current *state* and the *event* which can results is the initiation of more I/O. It also triggers a **state transition** which results in a new *state*

The system then waits for the next event to occur. State machines can simply be represented by directed graphs, *nodes* representing *states* and *edges* representing *state transitions* caused by incoming *events*. It's not the case that each event corresponds to exactly one state transition, nor is it the case that only one event can cause a particular state transition. However, each state-transition can be labelled with a set of events that can cause it to happen.

From each node, the outgoing edges represent the state transitions that can occur in a given state and the collection of events that label the state transition are the events that are considered possible in the given state.

When writing code for state machines, the typical structure is one with a single main loop where all events are collected and then dispatched to code that handles a specific combination of state and event. Something like Figure 1.

The function `getevent()` waits for the next event to occur (they may be collected on a queue) and returns a pointer to it. If the type of the event is `NewRequest`, it cannot be associated with an outstanding request, so we call `createrequest()` to create one. For the remaining events, the function `findrequest()` associates the event with an existing request. In this example, we've made a 2-dimensional array of functions to call when a particular event occurs in a particular state. Nil entries in the array indicate state/event pairs that cannot occur. The function `badevent()` is called in that case. `Badevent()` and `f()` are assumed to change the state of the request as dictated by the state transition triggered by the event.

As shown, programming state machines can be relatively straightforward. The only downside is that the completion of a request is achieved by calling a selection of functions from the array `fns`, in some order. This makes state-machine code hard to understand and even harder to debug: Bugs are often triggered by the history of previous events and that history is not naturally available.

1.2. Threads

Another way to deal with concurrent requests is to associate a *process* or a *thread* with each request. For now, we'll use the word **thread** and we'll deal with the difference between processes and threads later. When a client request comes in, a new thread is created to service the request. When the thread has finished, it returns a reply to the client and exits.

```
void
mainloop(void){
        Event *event;
        Request *request;
        void (*f)(Request*, Event*);

        initialize();
        while(event = getevent()){
                if(event->type == NewRequest){
                        createrequest(event);
                        continue;
                }
                request = findrequest(event);
                if(request == nil){
                        free(event);    /* bad request, ignore */
                        continue;
                }
                f = fns[request->state][event->type];
                if(f == nil){
                        badevent(request, event);
                        continue;
                }
                f(request, event);
        }
        cleanup();
}
```

Figure 1 Example of a main loop for a server state machine.

The dispatcher might look something like this:

```
void
dispatcher(void){
        extern void requestthread(void *request);
        void *request;

        while(request = getrequest())
                threadcreate(requestthread, request);
}
```

The function `getrequest()` reads a request and returns a pointer to it. The function `threadcreate()` creates a new thread and calls the function `requestthread()` in it. The function has one parameter and this will be a pointer to the request. Requests are declared as pointers to `void` since the *threadcreate* mechanism is agnostic of requests. It just calls a function with one parameter and doesn't care about its type.

The function `requestthread()` would look something like this:

```
void
requestthread(void *arg){
        Request *request;

        request = arg;
        /*
         * Code to interpret and carry out request goes here.
         * The reply will be embedded in the request data structure
         */
        sendreply(request);
        free(request);
        threadexits(nil);
}
```

Clearly, `sendreply()` sends the reply back to the client (whose identity, presumably, would be in the request data structure). `Threadexits()` is used by the thread to clean itself up; a *nil* exit status means there are no errors.

When the thread does I/O, for instance to the disk, that I/O must be coordinated with similar ones from other threads. This may, to some extent, be done by the operating system; it may, for example, treat individual *read* and *write* system calls as *atomic*. The programmer would have to be aware of this and also of the consequences for portability of code that relies on such atomicity.

A more generally accepted way to protect the atomicity of accesses to shared resources — such as disks, or shared data structures such as free lists or memory allocators — is to use *locks*, as shown in Figure 2.
`Qlock()`, as the name implies, is a lock with a queue: if a thread (or process) cannot obtain the lock, it queues itself on the lock; when the lock is released by another thread, that thread wakes up the one at the head of the queue to retry getting the lock.

Locks must be implemented with special atomic instructions that all modern processors now supply: Test-and-Set, Compare-and-Swap, etc.

1.3. Threads vs. State machines

Threads and state machines provide us with two different ways to achieve some independence between concurrent and mostly unrelated activities. Which one is better?

That depends on the application at hand, on the amount of interdependency of the concurrent activities on shared resources and data and also on the specification of the application at hand.

Telecom protocols and applications tend to be specified in the form of state machines, so using them is then convenient. Threads come more natural for such things as file servers, and interactive applications such as editors, word processors, etc., because a state-machine description for each of the parallel activities — which are likely all different: mouse handling, keyboard handling, graphics handling, what have you — would be difficult to create.

But even for telecom applications, threads can be useful to separate the *instances* of protocols (e.g., the conversations between a wireless base station and the mobiles using that base station).

I think it’s safe to say that most applications do not easily succumb to a specification in terms of state machines. The number of states could easily explode and setting up the exhaustive list of all state transitions can become very hard to do. And if a mistake is made somewhere it becomes difficult to find.

```
typedef struct Freelist {
        QLock;
        Block *head;
} Freelist;

Freelist *freelist;

Block *
allocb(void)
{
        Block *b;

        qlock(freelist);
        if(b = freelist->head)
                freelist->head = b->next;
        qunlock(freelist);
        if(b == nil)
                b = malloc(sizeof(Block));
        initblock(b);
        return b;
}

void
freeb(Block *b)
{
        qlock(freelist);
        b->next = freelist->head;
        freelist->head = b;
        qunlock(freelist);
}
```

Figure 2 Implementation of a free list using locks.

The wireless UMTS standard specifies state machines for connections at the L1 level (RR), as well as for the various call activities that take place over the established L1 connection: MM (voice), GMM (GPRS), and PDP (IP messaging). All these state machines interact: If one wants to establish a high speed IP connection, the L1 bearer must be reconfigured. To top this off, each of these state machines describes some part of the connection state for a single user. On a base station, there are hundreds such state machines running in parallel.

Initially, we ran each state machine (RR, MM, GMM, and PDP) in its own thread, thus representing a "user" by at least four threads. Later, we abandoned this programming model and we now run at least four state machines in a single thread. State machines and threads can thus coexist and it is a matter of convenience what model best applies to a given situation.

2. Object Orientation in a Concurrent World

Object-oriented programming has become so popular that it has taken on almost religious significance in computer science. Many years ago, Edsger Dijkstra (of semaphore fame) told the world that *goto* statements are bad and the world religiously started to write programs without *gotos*. It took many years for the programming community to come to its senses (but some still haven't done so). *Goto* has its place. It should be used in moderation and in good programming taste and then it can help clarify the structure of a program very nicely.

The same is true for object-oriented programming. Of course it's a good idea *generally* to separate implementation and interface, but it must not become a religion.

This is especially true in concurrent code. Implementing an object as a data structure with a bunch of *methods* for manipulating it is all well and good, but if these methods are called by concurrently running threads, then the implementer of the object and its methods may get a nasty surprise.

The object-oriented approach nicely hides the problems of concurrency:

- Implementers of the object have no idea whether their methods will be called from different threads and may not take precautions to manage concurrent access. All they can really do is

  ```
  void
  method(Object *self, ...)
  {
          lock(self);
          ...
          unlock(self);
  }
  ```

  They are creating locking overhead without knowing whether it is warranted by the application.

- Users of the object have no idea what the object does in terms of concurrency control. It mayn't have any so the caller is forced into something like this:

  ```
          lock(some_lock);
          object.method(...);
          unlock(some_lock);
  ```

  The users can't even put the lock in the object, because it's not their implementation.

  And if the object *does* implement its own concurrency control, the user doesn't know how that will affect the application as a whole. If two objects of the same type each have their own lock, a concurrent application may be deadlock free, but if they share a lock, it may deadlock. And the rule in object-oriented programming "*thou shalt not know how the interface is implemented*" may see to it that a correct implementation of a concurrent application is impossible to create.

  Hiding the implementation behind the interface is all well and good, but every now and then it's a really bad idea.

In a concurrent world, it can make a lot on sense to have a *thread* be the guardian of an object. Access to the object is then serialized as necessary by the controlling thread. Of course, this implies that the traditional method interface to the object then gets replaced by some sort of interthread communication interface. In other words, you have to send a message to the controlling thread to ask it to carry out an operation on the object.

It would still be possible to hide the message-passing interface to the object behind a series of function calls with the appearance of method calls. This is more or less what *remote procedure call* (RPC) is all about. However, the caller must know that the implementation is an RPC one, because in this concurrent world there really is a fundamental difference between calling `sqrt(x)` and `read(fd, buf, sizeof buf)`. The latter may cause the thread to be suspended.

To get back to our file server example, and the idea to launch a thread for each incoming request: This can be viewed as a form of object orientation in which each object (request) encapsulates its state inside a thread. An outside observer can only learn about the state of the object by asking its thread — nicely.

3. Plan 9 Threads

In the previous sections, we've established the background necessary to understand the motivations behind the Plan 9 thread model. We'll explain the model first and, at the end of this section, give a brief historical perspective on how the model came about.

3.1. Threads and Procs

Plan 9 processes are much like Unix processes: an address space and an execution environment consisting of registers, a program counter, a stack, environment variables and open-file state accessible through *file descriptors*.

Plan 9 runs on uniprocessors and shared-memory multiprocessors. Processes can, therefore, run truly concurrently. A scheduler decides which runnable processes will run and on what processor. But since a process has but a single program counter, it can only run on one processor at a time.

Plan 9 applications can achieve concurrency by creating more than one process and these processes may share parts or most of their address spaces (they can never share all of their address spaces because stack segments are always allocated on a per-process basis).

When processes share (part of) an address space, access to it is not concurrency controlled in any way. If a global variable `(int)g` has the value 0 and one process executes `g++`, while another executes `g+=2`, the end result may be 1, 2, or 3. Processes need to protect shared modifiable data with appropriate concurrency controls (e.g., locks).

Within a single process, it is possible create a set of *coroutines*. Coroutines can be compared to subroutines, but they are not the same. When a subroutine is called, control transfers to the subroutine and when it returns, control transfers back to the instruction after the subroutine-call instruction. Coroutines can call each other; when one coroutine calls the other, control transfers from one to the other. When the other returns control to the one, execution is resumed at the instruction after the call, just like a subroutine call. But when the one coroutine calls the other *again* control is transferred to the instruction where the other coroutine transferred control back. Coroutines are like processes and transferring control from one to the other is like having the scheduler schedule another process.

The Plan 9 thread mechanism is a coroutine mechanism with a scheduler that mediates the transfer of control from one coroutine to another. We call our coroutines **threads**. The scheduler is activated — explicitly by calling `yield()` or implicitly by calling one of a set of other functions which we shall discuss later — by the running thread, of which there can only be one in a process. In other words, if a thread gets into an infinite loop, it will deprive all other threads from the opportunity to run. And if a thread calls `sleep(1000)` the process containing the thread will become dormant for a thousand milliseconds and, again, deprive any other thread in the same process from running in the meantime.

Another way of viewing this behaviour of threads is to observe that threads cannot be preempted. They can only do it to themselves. As a consequence, threads can freely manipulate shared data structures without locking them, provided:

1. They make sure the data structure is in a consistent state before *yielding*;
2. No other *processes* access the shared data structure — only threads in the same process may do that.

It is these properties of threads (in contrast to processes) that we exploit in indeterministic Plan 9 applications: for each (somewhat) independent processing activity we create a thread to take care of that activity; and we make sure that all these threads share a single process. We can then allow the threads to manipulate all shared data without any need for locking.

The big remaining complexity is I/O. Since *read* and *write* calls block until the data has been read or written, a thread doing a read — to get the next mouse event, say — will hold up all other threads until the mouse event occurs.

We therefore create helper processes to carry out I/O on behalf of the threads in the main process. To make this work, we need an interthread communication mechanism which we supply in the form of *channels*.

3.2. Channels

A **channel** is a data structure in memory shared between threads and processes. We use the term **proc**, by the way, to identify processes that share memory with others in a multithreaded application. The basic operations on a channel are *send* and *receive*. Any thread or proc can send an item to a channel and any thread or proc can then receive that item. Items are received in the order they are sent, so the channel behaves like a single queue.

A channel can store zero or more items, the number (and size) of the items is specified when the channel is created (see *chancreate*(2)). When a send occurs and there is no room in the channel to store the additional item, the send *blocks* until a receive operation has created room. Similarly, a receive operation *blocks* on a channel with zero items in it, until a send operation has happened.

The special case of a channel with no storage will cause the send and receive operation to do a *rendez-vous*; that is, the first to arrive blocks until the second arrives and then they both proceed, the receiver with the sender's item.

The basic send and receive operations are

```
    int send(Channel *c, void *item);
    int recv(Channel *c, void *item);
```

Send sends the pointed-to item to the channel and blocks until the item has been copied (all items in a channel must have the same size and the size is specified when the channel is created).

For convenience there are special calls for sending an receiving pointers or integers:

```
    int sendp(Channel *c, void *p);
    int sendul(Channel *c, ulong v);
    void *recvp(Channel *c);
    ulong recvul(Channel *c);
```

There are also non-blocking variants of these calls, that is, calls that return an error code (of zero or nil) when the operation cannot immediately be completed:

```
int nbsend(Channel *c, void *item);
int nbsendp(Channel *c, void *p);
int nbsendul(Channel *c, ulong v);
int nbrecv(Channel *c, void *item);
void *nbrecvp(Channel *c);
ulong nbrecvul(Channel *c);
```

The `alt()` operation on a set of channels was modelled on the indeterministic choice operation from Hoare's Communicating Sequential Processes. Figure 3 contains an example to show how it works.

```
void
threadmain(int argc, char *argv[])
{
        char m[48];
        int t;
        Alt a[] = {
        /*       c              v               op    */
                {nil,   m,      CHANRCV},
                {nil,   &t,     CHANRCV},
                {nil,   nil,    CHANEND},
        };

        /* create mouse event channel and mouse process */
        a[0].c = chancreate(sizeof m, 0);
        proccreate(mouseproc, a[0].c, STACK);

        /* create clock event channel and clock process */
        a[1].c = chancreate(sizeof(ulong), 0);  /* clock event channel */
        proccreate(clockproc, a[1].c, STACK);

        for(;;){
                switch(alt(a)){
                case 0: /*mouse event */
                        fprint(2, "click ");
                        break;
                case 1: /* clock event */
                        fprint(2, "tic ");
                        break;
                default:
                        sysfatal("can't happen");
                }
        }
}
```

Figure 3 Example of an `alt()` call

`Alt()` takes as its parameter an array of descriptions of send and receive operations on channels. In particular, each description consists of a pointer to a channel, the address of a source or destination of an item and an operation to be carried out on the channel: CHANRCV, CHANSND, or CHANNOP (the latter means pretend this entry isn't there). The array is terminated by a special entry with the operation CHANEND or CHANNOBLK.

if one or more of the specified operations in the `alt()` statement can be carried out immediately, one of these is selected at random and allowed to complete. If none can be carried out, `alt()` blocks if the terminating entry is CHANEND until one can be carried out, or it terminates if the terminating entry is CHANNOBLK. `Alt()` returns the index of the operation that was carried out (or that of the CHANNOBLK entry).

When a thread blocks on any of these channel operations, it *yields* control to another thread (i.e., it preempts itself). The thread becomes runnable once again when the (or, in case of an `alt`, *an*) operation succeeds.

Channels, as mentioned before, do not have specific end points in threads or procs; any thread can send or receive on any channel. But in practice, channels are mostly used for a specific purpose and connect just two threads. A case in point is a helper proc that carries out the I/O on behalf of, say, the above-mentioned mouse-reading thread. Such a helper proc would look something like Figure 4.

```
void
mouseproc(void *arg)
{
        char m[48];
        int mfd;
        Channel *mc;

        mc = arg;
        if((mfd = open("/dev/mouse", OREAD)) < 0)
                sysfatal("open /dev/mouse: %r\n");
        for(;;){
                if(read(mfd, m, sizeof m) != sizeof m)
                        sysfatal("eof");
                send(mc, m);
        }
}
```

Figure 4 A typical I/O helper proc.

The channel along which mouse events are delivered to the mouse thread (in the proc with all the other threads) is given to the mouse proc as its argument.

The code for creating this I/O proc is shown in the `alt()` example of Figure 3:

```
        proccreate(mouseproc, c, STACK);
```

3.3. Stack Management

Each thread needs a stack. These are allocated when the thread is created. The size of the stack is the third parameter in `proccreate` and `threadcreate`. Plan 9 processes can only have one stack segment, so thread stacks cannot be made to grow automatically as do the stacks of normal Plan 9 processes. Good programming practice therefore dictates that stacks are not used to declare large local arrays. Deep recursion in a thread is also discouraged.

This is a not a very satisfactory state of affairs, but the restrictions on stacks are ones that most applications can live with quite comfortably.

3.4. Historical Perspective

Dijkstra's guarded commands [1976] are earliest influence on threads. A guarded command of the form $G \rightarrow S$, evaluates G and, if true, executes S. The *conditional statement*, of the form $G_0 \rightarrow S_0 \,\square\, G_1 \rightarrow S_1 \,\square \ldots \square\, G_n \rightarrow S_n$ evaluates all of the guards and executes any one of the statements whose guard evaluates to true. The `alt()` statement is modelled on this form of conditional statement: the guard is “can this send or receive operation be carried out now?” and the statement is carrying it out.

Hoare's Communicating Sequential Processes paper [1978] extended the idea to running guarded command sets in parallel. In his book [1985], he added channels to the model.

Luca Cardelli and Rob Pike built a programming language for programs using mice called Squeak [1985] around the idea of using communication between parallel processes via channels. The notation *c*?*var* reads a value from channel *c* into variable *var*, while *var*!*c* sends the variable *var* onto channel *c*.

Cardelli and Pike later used this in the Concurrent Window System [1988] and Pike used it in the improved language Newsqueak [1989].

Phil Winterbottom designed the language Alef [1994] as a C-like language with concurrency via threads and procs, with channels, built in.

Sape Mullender wrote the C thread library for Plan 9 [1999] and this is what is described in this paper.

4. Workers, an example of using threads and channels

In this section, we'll present a serious example of how threads and channels can be used to write concurrent applications that do no need any locks.

The idea is the following. Suppose we have a server of some sort that serves multiple clients making requests (and, presumably, expecting responses). The server can serve many requests simultaneously, so, when a new requests comes in, a thread is dispatched to process the request and return a response.

This can be done by creating a new thread for each incoming request, but, even though thread creation isn't very expensive, it may be expensive enough to prefer a solution where threads are reused to serve more than one request.

So, when a request comes in, we try to reuse an existing (idle) thread to precess the request, or, if there aren't any, we create one more thread. We want all these threads to run in one proc, so that they can share data without needing locks. Let's call these threads **worker** threads, or plain **workers**.

We'll represent a request by a pointer to a function that processes the request and a pointer to the data that represents the request:

```
typedef struct Request Request;
struct Request {
        void    (*func)(Worker*, void*);
        void    *arg;
};
```

We'll assume functions `reqalloc` and `reqfree` allocate and free `Request` structures.

We will let a channel with sufficient storage represent the queue of idle workers:

```
Channel *workerthreads;

void
workinit(void)
{
        ...
        workerthreads = chancreate(sizeof(Worker *), 256);
        ...
}
```

Given a request, then, we find an idle worker — or create a new one — as shown in the `allocwork` function of Figure 5.

Next, there needs to be a mechanism by which the request is given to the worker. We use a channel for this, one per worker. A worker, when it's idle will block by receiving from this (empty) channel and, when there's work, will unblock by receiving the request. So we get the `allocwork` and `worker` functions of Figure 5.

```
struct Worker {
        Channel *chan;
        ...
};

static void
allocwork(Request *r)
{
        Worker *w;

        w = nbrecvp(workerthreads);
        if(w == nil){
                w = malloc(sizeof(Worker));
                w->chan = chancreate(sizeof(void*), 1);
                threadcreate(worker, w, 8*1024);
        }
        sendp(w->chan, r);
}

void
worker(void *arg)
{
        Worker *w;
        Request *r;

        w = arg;
        for(;;){
                r = recvp(w->chan);
                r->func(w, r->arg);
                reqfree(r);
                sendp(workerthreads, w);
        }
}
```

Figure 5 The `allocwork` and `worker` functions.

`Allocwork` does a non-blocking receive on `workerthreads` so it doesn't get stuck when there aren't any idle workers available. It creates a worker, if necessary by allocating a `Worker` struct, populating it with a channel and creating the worker thread. Finally, it sends the request to the worker.

The worker thread executes the function `worker` which stays in an infinite loop accepting and executing requests. The statement "`r = recvp(w->chan);`" blocks until work comes in. Then the function in the request is executed with its request argument (and a pointer to the worker itself). The request is then freed by calling `reqfree`. Finally, the worker prepares for receiving a new request by placing itself in the queue of idle workers: "`sendp(workerthreads, w);`" (a call that does not block because this channel has plenty of storage) and looping back to block on the receive operation for more work.

A few loose ends remain. The function `allocwork()` *must* be called in the proc that contains all the (other) worker threads. If it is not, and if the supply of idle workers runs out, it will call `threadcreate` and create a thread in the proc it's running in instead of the one in which all the other workers are running.

The solution is to use a *dispatcher thread* in the "worker proc" that calls `allocwork`, shown in Figure 6.

```
static void
srve(void *arg)
{
        void *r;
        Channel *dispatchc;

        dispatchc = arg;
        while(r = recvp(dispatchc))
                allocwork(workerthreads, r);
        threadexits("srve");
}

void
workerdispatch(void (*func)(Worker*,void*), void *arg)
{
        Request *req;

        req = reqalloc();
        req->func = func;
        req->arg = arg;
        sendp(dispatchc, req);
}

void
workinit(int flags)
{
        workerthreads = chancreate(sizeof(Worker *), 256);
        dispatchc = chancreate(sizeof(void*), 1);
        procrfork(srve, dispatchc, 2*1024, flags);
        return dispatchc;
}
```

Figure 6 Server and dispatcher for creating worker threads.

`Workerdispatch` can be called from any proc to run `func()` in a worker thread. It does this by sending the request to the thread `srve` which must be (and *is*, by force of the fact that it creates all the worker threads) in the worker proc.

It is useful here to remind the reader that we have presented a mechanism for managing a pool of worker threads that can be commissioned to execute a series of incoming requests in a server. Such code is fairly common in many server applications. The Plan 9 contrast is that the code is very concise — we've presented all of it here — and needs no concurrency control in the form of, for example, locks whatsoever.

This makes the Plan 9 approach to managing concurrency in distributed systems unique and it is a good demonstration of the fact that, choosing the right abstractions and primitives for building a large system goes a very long way in keeping its complexity manageable.

5. Conclusion

In the ten years since the thread library was written, it has been used in all new concurrent applications and many of the existing ones have been retrofitted to use it. The vast majority of these application do not contain any locks. The few that still do are ones that were retrofitted — and it just wasn't worth the effort to restructure the application to remove them.

Using well-chosen abstractions and tools is immensely important in making large complex programs feasible. Threads are one such tool. The addition of channels and the explicit way in which threads within a process are not preemptable make our thread model much more powerful than, for example, Pthreads in Unix.

6. References

[Plan 9, 2000]
Plan 9 Manual, 3rd edition published on-line only at http://plan9.bell-labs.com/sys/man

[Cardelli & Pike, 1985]
L. Cardelli, R. Pike, "Squeak: a Language for Communicating with Mice", *Computer Graphics*, Proceedings of SIGGRAPH, 1985

[Pike, 1988]
"A Concurrent Window System", *Computing Systems*, 1989

[Dijkstra, 1975]
E.W. Dijkstra "Guarded Commands", *Communications of the ACM*, **18**(8), pp. 666-677 1975

[Hoare, 1978]
C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM, **21**(8), pp. 666-677 1978

[Hoare, 1985]
C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

[Mullender, 2000]
"Thread(2)", *Plan 9 Programmer's Manual, Volume I*, 3rd Edition, Lucent Technologies, Murray Hill, NJ, 2000

[Pike, 1990]
R. Pike, "The Implementation of Newsqueak", *Software Practice & Experience*, **20**(7), pp. 649-659, July 1990.

[Pike et al., 1995]
R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, "Plan 9 from Bell Labs", *Computing Systems*, **8**(3), Summer 1995, pp. 221—254

[Winterbottom, 1995]
P. Winterbottom, "Alef Language Reference Manual", *Plan 9 Programmer's Manual, Volume II*, AT&T, Murray Hill, NJ, 1995