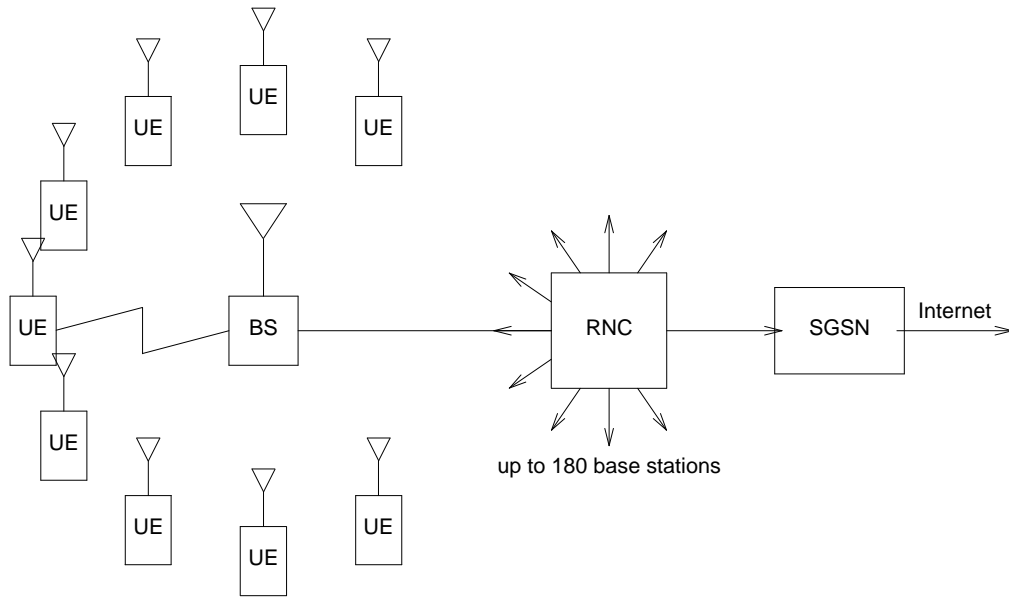


## **Living with Indeterminacy**

*Sape Mullender,*

Chief Cook and Bottle Washer  
Network Systems Laboratory  
Bell Laboratories  
2018 Antwerp, Belgium

# Cellular Networks



## Base Station Router

Joint work with [Peter Bosch](#)

- Integrate RNC and SGSN functionality into the Base Stations
- Use good old distributed systems algorithms to do handover
- Temporarily forward packets from old to new base station
- Use Mobile IP (or Not Mobile IP) in the backhaul network

Reduces round-trip latency between UE and RNC

Improves reliability of the network (no single point of failure)

Reduces Cost

## Prototype Implementation

We built a prototype (200K lines of code) on an old Lucent OneBTS Base Station after porting Plan to it.

Although that wasn't the plan, it had to become a hard real-time system:  
Miss a 10ms deadline and the radio crashes ... (it was an **old** OneBTS).

I implemented a real-time scheduler for Plan 9

But locks in the thread library still caused us to miss occasional deadlines

**But ...**

I didn't come here to talk about Base Station Routers

## Multicore Processors

When I started using computers, in the 1970s, we had a PDP-11/45 as big as a room that ran at 300KHz. Ever since, computers kept getting faster and smaller. At some point, computers actually *had* to get smaller *in order* to get faster.

Now, typical processors runs 10,000 times faster than the PDP-11 and they're no longer getting much faster very quickly.

Today, instead of getting faster, they're multiplying (it's the Darwin Year).

The **multicore processor** is here.

Two processors on a chip is now old hat, four is normal, eight is possible and 64 is just round the corner.

How do you use them?

## Multicore Processing

A multicore processor is a **shared-memory multiprocessor** — all processors see the same memory. Processors communicate through shared memory.

To use them efficiently, we can do two things:

1. Run different applications on different processors
2. Make applications parallel so they can run faster by running on more than one processor

Method 1 doesn't scale well, because we never run more than a few applications at once.

Method 2 doesn't work well either — we're not very good at writing parallel applications — They're so ... well ... *indeterministic*.

## The Need for Synchronization

When two processes (which share memory) both execute

```
v += 1000;
```

each process copies  $v$  into a register, adds 10,000 to it and copies it back into the variable  $v$ . The end result may thus be that  $v$  gets incremented only once.

Locks or mutexes are the normal protection mechanism to prevent such race conditions.

Now consider two processes, one executing

```
lock(a); lock(b); xfer(a, b, 1000); unlock(b); unlock(a);
```

and the other

```
lock(b); lock(a); xfer(b, a, 1000); unlock(a); unlock(b);
```

What might happen?

## Alternatives to Locks

Locks are nasty but necessary things. With locks, correctly used, in all the right places, concurrent and parallel applications will run correctly.

But,

- It's hard to get them in all the right places
- And to make sure every path through the code releases them
- What's the cost of locks?
- Are there alternatives to locks?

Locks can be expensive and, yes, there are alternatives ...

## Programming without Locks

The Plan 9 Thread Library presents a programming model in which

- Threads belong to a process (“proc”)
- Only one thread per proc can be running and the *thread* must give up the processor
- Threads communicate by sending messages through *channels*
- Channel operations *may* wait on empty/full channels (causing thread scheduling)
- An indeterministic *choice operator* can pick at random between a list of channel operations (see Guarded Commands, CSP).

Threaded programs are usually designed so that all threads manipulating shared data share a proc. This makes locks unnecessary.

As a result, any locks left in Plan 9 applications tend to be in the thread library itself.

## How Locks Work

Locks are implemented using special, atomic **Read-Modify-Write** instructions, such as *Test-and-Set*.

Here's how `tas` is implemented on the Pentium and how it's used to implement `lock()`.

```
TEXT    _tas(SB), $0
        MOVL    $0xdeadead, AX
        MOVL    1+0(FP), BX
        XCHGL   AX, (BX)
        RET

void
lock(Lock *lk)
{
    int i;

    if(!_tas(&lk->val))
        return;
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(0);
    }
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(100);
    }
    while(_tas(&lk->val))
        sleep(1000);
}
```

## What Locks Cost

- If the lock is not held, the cost is a few instructions plus the cost of the RMW instruction (e.g., `tas`).
- If the lock *is* already held, there is the additional cost of, at least, a system call, two context switches and one more RMW instruction.

Context switches, on modern processors, are very expensive. In addition to the cost of user space/kernel space transitions, there is the cost of cleaning out/repopulating the Translation Lookaside Buffers and the attendant cache misses.

RMW instructions are expensive too. They have to bypass the cache and lock the memory bus for all processors (in order to guarantee atomicity). Since memory latencies are on the order of 100ns, One RMW instruction may take as much as 300 to 1000 instruction times — per processor.

This is still in the noise, when compared to a context switch, but the cost of an atomic instruction is significant.

So, this begs the question ...

## Can Wait-Free Data Structures Help?

If there is a lot of lock contention, there'll be lots of context switches. Unnecessary context switches are very expensive, so it's worth investigating.

Real-time applications, especially, suffer from the presence of locks (in library routines).

- Either, we must go the whole hog and deal with the interprocess dependencies caused by locks (priority inversion, schedulability analysis, knowing, for everything that shares a lock with a real-time process, how much time is spent holding the lock);
- Or we just get rid of the locks ...

## Example — Wait-free Queue

```
typedef struct Q {
    /* Single producer, single consumer Queue */
    long   produced;
    long   consumed;
    ulong  mask;
    void   *data[];      /* size is power of two */
} Q;

int
qput(Q *q, void *x)
{
    if (q->produced - q->consumed > q->mask)
        return 0;
    q->data[q->produced & q->mask] = x;
    q->produced++;
    return 1;
}

void *
qget(Q *q)
{
    void *x;

    if (q->produced == q->consumed)
        return nil;
    x = q->data[q->consumed & q->mask];
    q->consumed++;
    return x;
}
```

There are those who'd write

```
q->data[q->produced++ & q->mask] = x;
```

They'd be wrong.

## Multiproducer/multiconsumer Queue

Can it be done? These two statements

```
q->data[q->produced & q->mask] = x;  
q->produced++;
```

must be executed atomically. If not, this race is possible:

```
/* Producer 1 */          /* Producer 2 */  
q->data[q->produced & q->mask] = x;      q->data[q->produced & q->mask] = y;  
q->produced++;              q->produced++;
```

## RMW instructions to the rescue ...

Compare-and-Swap, `cas(ptr, old, new)`, is an instruction that atomically executes:

```
if(*ptr != old)
    return 0;      /* failure */
*ptr = new; return 1; /* success */
```

What about this:

```
int
qput(Q *q, void *x)
{
    long prod;

    prod = q->produced;
    if (prod - q->consumed > q->mask)
        return 0;
    /* Location in queue must contain NIL */
    assert(x != nil);
    if(cas(&q->data[prod & q->mask], nil, x)){
        q->produced = prod+1;
        return 1;
    }
    /* it's not nil, there must have been a race; now what? */
}
```

Close, but no cigar. When `cas` fails, we'd like to try again, but we'd have to wait until our competing producer increments `q->produced`.

Or do we?

## One process can finish another's job!

When Producer 1 fails to grab the desired location in the queue, it is known what Producer 2 will do with `q->produced`: it'll increment it. Producer 1 can do that in its place.

But then `q->produced` may be incremented twice. Once again, compare-and-swap to the rescue:

```
int
qput(Q *q, void *x)
{
    long prod;
    int success;

    do{
        prod = q->produced;
        if(prod - q->consumed > q->mask)
            return 0;          /* Q is full */
        success = cas(&q->data[prod & q->mask], nil, x);
        cas(&q->produced, prod, prod+1);
    }while(!success);
    return 1;
}
```

Now, `cas` sees to it that `q->produced` only gets incremented if it still has the expected old value; in other words, it cannot accidentally be incremented twice. There's no need to check whether `cas` is successful — it doesn't matter.

There's another, more subtle, race condition that I'll explain if anybody asks. But first:

## Wait-free Thread Library

Plan 9 threads communicate using **channels**. They are multiproducer, multiconsumer fixed-size queues of messages with a wake-up mechanism for threads that indicate they need to wait for items or space in the channel.

Wait-free channels are implemented using the queues I just showed. But there's more to them, of course. Sleeping threads must be woken up when there's work for them to do and this is done with more atomic instructions.

All in all, we need roughly 5 c a s instructions per channel operation.

I recompiled all Plan 9 applications that use the thread library with the new, wait-free library. Here is my conclusion:

## Comparing the Locking Thread Library to the Wait-Free One

1. Everything still works! (In fact, all threaded applications on this laptop were compiled using the wait-free thread library.)
2. It doesn't feel faster or slower.

But maybe the thread library only accounts for a very small fraction of the time.

We ran some measurements with a toy program that creates 1200 threads (either in one process or in 1200 processes) and that makes 20,000 channel operations:

	<i>u=user, s=system, r=real time</i>					
	<i>conventional</i>			<i>wait-free</i>		
<i>one proc, all threads</i>	0.09u	0.05s	0.14r	0.11u	0.02s	0.15r
<i>one thread per proc</i>	0.00u	0.03s	7.68r	0.01u	0.01s	1.45r

So, wait-free can be five times faster than conventional!

How did that happen? We took another good look at the implementation of `lock()`:

## Lock implementation in Plan 9

```
void
lock(Lock *lk)
{
    int i;

    /* once fast */
    if(!_tas(&lk->val))
        return;
    /* a thousand times pretty fast */
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(0);
    }
    /* now nice and slow */
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(100);
    }
    /* take your time */
    while(_tas(&lk->val))
        sleep(1000);
}
```

In 175,000 lock calls, we found 1,900,000 calls to tas. No wonder lock can be slow.  
We reimplemented lock:

## Greased Locks

```
void
lock(Lock *lk)
{
    /* ainc (atomic inc) returns new value */
    if(ainc(&lk->key) == 1)
        return; /* got it! */
    semacquire(&lk->sem, Block);    /* Sleeping Beauty */
}

void
unlock(Lock *lk)
{
    /* adec returns new value */
    if(adec(&lk->key) == 0)
        return; /* nobody's sleeping */
    semrelease(&lk->sem, 1);        /* Prince (white stallion) */
}
```

Now, the locking library and the wait-free library have pretty much the same performance.

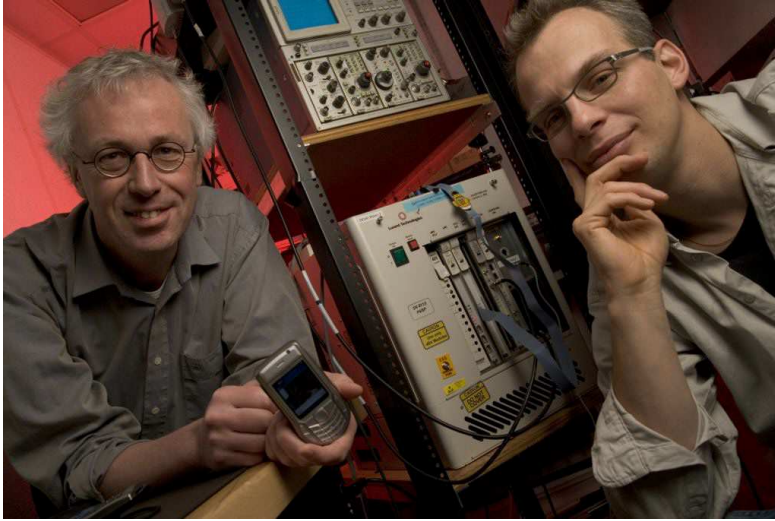
So, wait-free implementations don't have any performance advantage over conventional, locking ones. But ...

## Base Station Router

Lock-free data structures work wonders in (many) real-time applications:

- No need for dealing with priority inversion
- No unnecessary context switches
- We mostly used simple single-consumer/single-producer queues, so no need for a lot of expensive RMW instructions.
- `Semacquire` and `semrelease` provide an excellent mechanism to allow one process (the real-time one, typically) to wake up another
- and we made sure `semrelease` has no locks — but that's another story

Thank You



## Atomic Queues, the Real Thing

We use load-linked/store-conditional (this example) or 64-bit compare-and-swap.

```
int
qput(Q *q, uintptr new)
{
    uint prod;
    Qelem old, *data;

    assert(new != Mark);
    for(;;){
        prod = q->prod;
        if(prod - q->cons >= q->size)
            return 0; /* Q is full */
        data = &q->data[prod & q->mask];
        old = loadlink(data);
        /* Check if old is still good; i.e., no puts happened: */
        if(prod != q->prod)
            continue;
        if(old != Mark) /* Too late, try next slot */
            cas((ulong*)&q->prod, (ulong)prod, (ulong)(prod+1));
        else if(storecond(data, new))
            break; /* Cool, we're done */
        /* else: don't update q->prod, try again first */
    }
    cas((ulong*)&q->prod, (ulong)prod, (ulong)(prod+1));
    return 1;
}
```