Managing Concurrency in Distributed Systems

Pierre Jansen
Sape Mullender

Concurrency in Shared Memory

If you have processes sharing memory, you have to protect *all* shared data structures:

- Those you design to be shared
- All the others you never gave much thought:
  Memory allocator
  - Standard I/O
  - Printf or Fprint
  - Graphics

  Library routines need to be *reentrant*, or calls into them must be protected against concurrent invocation.

  Bugs here are hard to find: usually, you don't know how the libraries work; you may not even have access to their source code.

Silver Bullet

Is there a single mechanism that *deals* with concurrency one that, if you use it properly and everywhere, will just take care of it?

No

Of course not. We wouldn't have this class if there was.

But there are some important rules and tools.

The rules tend to be more important than the tools.

And, what are the rules? Here's what's common to all of them:

Discipline! Make your rules and stick to them.

Atomicity and Serializability

Atomicity is the key to dealing with concurrency: it gives us *serializability*.

A set of operations is serializable if, executing them concurrently gives the same result as executing them one after the other in some order.

Updating unrelated data structures is a (not very interesting) example.

But here's a very interesting one:

> On most multiprocessors, reading and writing aligned machine words is atomic: If one processor writes a word and another reads it, the value read is either the old or the new value.

There is often but certainly not always even stronger correlation between reads and writes:

> If one processor writes a word and the other processor then reads it, the new value will be read.

This behavior is only obvious if you don't know how modern processors are built: they have caches and they are heavily pipelined. Obtaining this type of coherency is not an accident; it is carefully designed in.

How can we use the atomicity of one-word reads and writes?

Non-blocking Queue

```
typedef struct Queue Queue;
struct Queue {
        long inp, outp;
        void *data[Size];       /* Size must be power of two */
}

void*
nbconsume(Queue *q)
{
        void *datum;

        if(q->inp == q->outp) return nil;
        datum = *q->data[q->outp & (Size-1)];
        q->outp++;
        return datum;
}

int
nbproduce(Queue *q, void *datum)
{
        if(q->inp - q->outp == Size) return 0;
        q->data[q->inp & (Size-1)] = datum;
        q->inp++;
        return 1;
}
```

Read-Modify-Write

An assembler instruction that exists on almost all architectures is

```
INC   (Rx)
```

Register Rx contains the address of a memory word, the `INC` instruction increments the word pointed to.

Such instructions are called *Read-Modify-Write* instructions because that's what they do.

They are not usually atomic on multiprocessors and they aren't even atomic on a few uniprocessors an interrupt or page fault can stop the instruction half-way through.

Thus, two processors, concurrently incrementing a memory location using a single instruction to do so may not exhibit serializable behavior.

Let's go back to the previous slide and see what happens when there is more than one producer or consumer ï

Atomic Read-Modify-Write Instructions

I won’t bore you with the proof that, if you don’t have some species of atomic RMW instruction, you can’t solve the multi-consumer (or multi-producer) queue problem. But it’s a fact.

That’s why pretty much every multiprocessor worth its voltage has them.

- Test-and-set
- Compare and swap
- Lock (Intel)

Mutexes Spin Locks

```
void
lock(ulong *l)
{
        while(!tas(l))
                delay(1);
}

void
unlock(ulong *l)
{
        *l = 0;
}

ulong l;

lock(&l)
update(shared_something);
unlock(&l);
```

Real Locks

Spin locks are cute, but they cause an awful lot of processor cycles to go to waste by, well, spinning.

Real locks come with a mechanism for the process trying to get the lock to go to sleep and the process having the lock to wake up the sleeper when it releases the lock.

This is tricky stuff, because the race that lurks is that the process releases the lock just when the process trying to get it is about to go to sleep, so that the wakeup misses it and it sleeps forever.

```
void
qlock(QLock *q)
{
        Proc *p;

        lock(&q->use);
        if(!q->locked) {
                q->locked = 1;
                unlock(&q->use);
                return;
        }
        p = q->tail;
        if(p == 0)
                q->head = up;
        else
                p->qnext = up;
        q->tail = up;
        up->qnext = 0;
        up->state = Queueing;
        up->qpc = getcallerpc(&q);
        unlock(&q->use);
        sched();
}
```

Multi Producer/Consumer Queue

```
struct Q {
        Lock;
        long    produced;
        long    consumed;
        ulong   mask;   /* Q size is power of 2; mask is one less */
        void    *data[1];       /* additional elements are mallocated */
};

int
qput(Q *q, void *x)
{
        lock(q);
        while(q->produced - q->consumed > q->mask){
                unlock(q);
                sleep(1);
                lock(q);
        }
        q->data[q->produced & q->mask] = x;
        q->produced++;
        unlock(q);
        return 1;
}
```

Beware of optimizing compilers!

Locks and Real Time

A special nightmare.

1. A low-priority process grabs a lock
2. A medium-priority process preempts the low-priority one
3. A high-priority process preempts the medium-priority one
4. The high-priority process attempts to get the lock (and fails)
5. The high-priority process gives up the processor `sleep(1)`
6. The medium-priority process resumes execution
7. The high-priority process misses its deadline

The classic example of *Priority Inversion*.

Dealing with Priority Inversion

Dealing with priority inversion is a flourishing cottage industry.

One technique:

> Changing the priority of a process that holds a lock to that of the highest-priority process that wants it (priority inheritance)

Another technique (Jansen et al):

> Processes that might request a lock held by a process already running are not scheduled until that process has relinquished the lock. Schedulability analysis is used to prevent a mix of processes where this can lead to deadline miss.

But can you avoid blocking (i.e., locking) altogether?

Why would we like to avoid resource scheduling?

Good question.

Because it has lots of overhead. Grabbing a lock (no contention) is two or three instructions, freeing it is one instruction.

But, if the lock protects a shared resource in a real-time system, you must tell the real-time scheduler about grabbing and releasing the lock: a system call, or at least a few thousand instructions.

Doing this for shared data structures just isn’t practical, so let’s ask again,

Can you avoid blocking (i.e., locking) altogether?

Yes, you can avoid locking (and priority inversion)

Provided the environment is right.

- Consider two robots, sharing a block of 3D space, each controlled by a real-time process. The shared space is a shared resource and, when one robot enters it, the other can't. This requires classical solutions to priority inversion as mentioned in the previous slide.
- Now consider two or more processes sharing data structures. Here, by restructuring code, you can avoid almost any blocking operation. (We have, in our hard real-time UMTS base station router.)

If there are physical resources, you may try to organize your code to appoint one process to manage that resource, and have other processes send it requests to do stuff asynchronously.

This isn't practical int he robot case, but, hey, robots are slow anyway.

Techniques for avoiding locks

- Non Blocking Queues
  We showed these already. One consumer, multiple producers? Just use one queue for each producer/consumer pair and have the consumer round-robin over the queues. Multiple consumers? I can't think of an example where you might need one.

- Memory Allocation
  Malloc/free uses locks. But if you're allocating fixed-size blocks from a free list, you can use Compare-and-Swap:

```
do {
        if((b = head) == nil) return nil;
} while(! cas(&head, b, b->next));
return b;
```

Thread Library

Most concurrent applications in Plan 9 (editors, the window system, GUIs, user-level file systems, etc.) use the Plan 9 Thread Library.

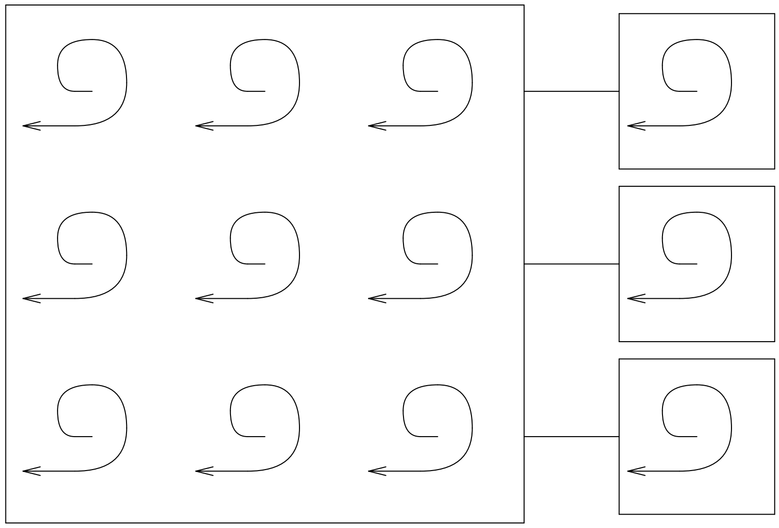
The thread library implements *Processes* (‘*Procs*’), *Threads*, and *Channels* for interprocess and interthread communication.

- A *proc* is a container for one or more *threads*.
- *Procs* are independently scheduled and, especially on multiprocessors, run truly concurrently.
- *Procs* in a threaded application share memory.
- *Threads* are *coroutines* within a *proc*.
- A *thread* runs until *it* relinquishes the processor and allows another *thread* in the same *proc* to run.
- A *channel* is a concurrency-protected *pipe* in shared memory through which threads in the same or in different procs can exchange fix-sized *messages* using *send* and *recv* primitives.

Plan 9 only has synchronous (‘blocking’) system calls. There is no *select* system call.

Instead, applications are usually organized to have a number of I/O *procs* with a single *thread* each (so that, when that *thread* blocks on a system call, no other *threads* are affected), and one or more separate *procs*, usually with multiple *threads* that do the actual work.

Application Structure



Run blocking system calls in a single-threaded proc.

Run data-sharing threads in one proc.

You won't need locks!

Plan 9 Threaded Applications

- We use the thread package very extensively. All new applications that have to deal with any sort of concurrency use it.
- The lock-less style of programming you can achieve by concentrating threads sharing data in a single proc and leaving shared data consistent when yielding is used by everybody.
- We do have several kinds of locks and they do find occasional use. But the vast majority of programs can do without them. Programs are fairly easy to debug.
- Stack overflow is not detected and this sometimes is a problem.

Communication

Threads can communicate through shared memory, but for synchronized communication, we use *channels*.

A *channel* is like a pipe through which fixed-size messages can be sent, pointers to `Pacbuf` structs in this example:

```
struct Pacbuf {
        Pmsg;
        int len;
        char data[Pacbufsize];
};
Channel *full, *empty;

void
playinit(void)
{
        int i;

        full = chancreate(sizeof(Pacbuf*), 1);
        empty = chancreate(sizeof(Pacbuf*), NPacbuf);
        for(i = 0; i < NPacbuf; i++)
                sendp(empty, malloc(sizeof(Pacbuf)));
        procrfork(pacproc, nil, 32*1024, RFFDG);
        procrfork(pcmproc, nil, 32*1024, RFFDG);
}
```

Channel Operations

```
send(c, m);

sendp(c, p);
sendul(c, n);
if (nbsend(c, m) == 0) error("fail");
if (nbsendp(c, p) == 0) error("fail");
if (nbsendul(c, n) == 0) error("fail");

recv(c, m);

p = recvp(c);
n = recvul(c);
if (nbrecv(c, m) == 0) error("fail");
if ((p = nbrecvp(c)) == nil)
	error("fail (or nil ptr received)");
if ((n = nbrecvul(c)) == 0)
	error("fail (or zero received)");
```

Alt

```
void
pacproc(void*)
{
        Pacbuf *pb;
        Alt a[3] = {
                {empty, &pb, CHANNOP},
                {playc, &newstate.m, CHANRCV},
                {nil, nil, CHANEND},
        };

        ...
        for(;;){
                switch(alt(a)){
                case 0:
                        /* Play out next buffer (pb points to one already) */
                        ...
                case 1:
                        /* Deal with incoming command */
                        ...
                }
        }
}
```

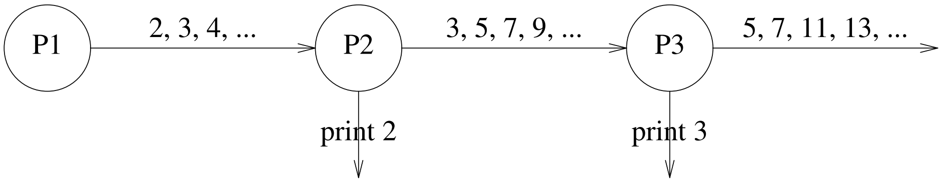
Alt

```
void
pacproc(void*)
{
	Pacbuf *pb;
	Alt a[3] = {
		{empty, &pb, CHANNOP},
		{playc, &newstate.m, CHANRCV},
		{nil, nil, CHANNOBLK},
	};

	...
	for(;;){
		switch(alt(a)){
		case 0:
			/* Play out next buffer (pb points to one already) */
			...
		case 1:
			/* Deal with incoming command */
			...
		case 2:
			/* Nothing happened */
			sleep(10);
		}
	}
}
```

Thread Example

Generating prime number, using Eratosthenes' Sieve.




```
void
filter(void *v){
        Channel *c, *nc;
        ulong p, n;

        c = v;
        p = recvul(c);
        print("%d\n", p);
        nc = chancreate(sizeof(long), 0);
        threadcreate(filter, nc, 1024);
        for(;;){
                n = recvul(c);
                if (n % p) sendul(nc, n);
        }
}
```

Back to the Playlist File System

The play loop consists of two procs, one starts the decompression proc and reads the output, the other writes the audio device.

```
Channel *full;
Channel *empty;

void
pacproc(void*)
{
	int fd;
	Pacbuf *pb;
	Alt a[3] = {
		{empty, &pb, CHANNOP},
		{playc, &newstate.m, CHANRCV},
		{nil, nil, CHANEND},
	};

	fd = -1;
	for(;;){
		switch(alt(a)){
		case 0:	/* Play out next buffer (pb points to one already) */
			pb->m = playstate.m;
			pb->len = read(fd, pb->data, sizeof pb->data);
			if(pb->len > 0){
				sendp(full, pb);
				break;
			}
			/* error handling deleted ... */
		case 1:
			/* command from File System (deleted) */
			...
		}
	}
}
```

The dispatcher in the file system

The problem to be solved is the following:

- Requests come into the file server by reading them from a pipe
- Some of these cannot be serviced immediately; a state change must occur first

Our solution: have a single-threaded proc read requests from the pipe and dispatch them to a set of threads in another proc. Here's how to do this elegantly:

```
void
srv(void*)
{
	Req *r;
	Channel *dispatchc;
	/*
	 * This is the proc with all the action.
	 * When a file request comes in, it is dispatched to this proc
	 * for processing.  Two extra threads field changes in play state
	 * and volume state.
	 * By keeping all the action in this proc, we won't need locks
	 */

	dispatchc = chancreate(sizeof(Req*), 1);
	procrfork(srvio, dispatchc, 4096, RFFDG);

	threadcreate(volumeupdater, nil, 4096);
	threadcreate(playupdater, nil, 4096);
	threadcreate(playlistsrv, nil, 4096);

	while(r = recvp(dispatchc))
		allocwork(r);

}
```

At the other end of the dispatch channel ...

```
void
srvio(void *arg)
{
	char e[32];
	int n;
	Req *r;
	Channel *dispatchc;

	dispatchc = arg;
	r = reqalloc();
	for(;;){
		n = read9pmsg(srvfd[0], r->indata, messagesize);
		if(n == 0)
			continue;
		if(n < 0){
			rerrstr(e, sizeof e);
			if (strcmp(e, "interrupted") == 0)
				continue;
			sysfatal("srvio: %s", e);
		}
		if(convM2S(r->indata, n, &r->ifcall) == 0)
			continue;
		sendp(dispatchc, r);
		r = reqalloc();
	}
}
```

Allocwork: Managing the worker pool

A worker is a thread that carries out a file-service request.

In concept, we create a worker when a request comes in, the worker carries out the request, returns a reply and commits suicide.

In practice, this is inefficient: creating a thread isn't very expensive, but it's not a no-op either. So when a worker thread finishes, it doesn't terminate but asks for new work. We create new worker threads when there are no idle ones available. This way, we'll have just enough of them.

Worker threads put themselves on a channel called `workers` when they're ready to accept a new job.

```
void
worker(void *arg)
{
        Worker *w;
        Wmsg m;

        w = arg;
        recv(w->eventc, &m);
        for(;;){
                w->r = m.arg;
                work(w);        /* This actually does the work */
                sendp(workers, w);
                recv(w->eventc, &m);
        }
}
```

Allocwork

```
void
allocwork(Req *r)
{
	Worker *w;
	Wmsg m;

	m.cmd = Work;
	m.off = 0;
	m.arg = r;
	if(w = nbrecvp(workers)){
		send(w->eventc, &m);
		return;
	}
	/* No worker ready to accept request, allocate one */
	w = malloc(sizeof(Worker));
	w->eventc = chancreate(sizeof(Wmsg), 1);
	threadcreate(worker, w, 4096);
	send(w->eventc, &m);
}
```

If you can't use threads ...

The Plan 9 kernel (and any other operating system kernel) must deal with real interrupts, page faults and other traps. Using a thread library is not an option here.

The Plan 9 kernel uses three kinds of *locks*. In order of severity (how much they restrict concurrent activity), they are:

- `Qlock`,
- `Lock`, and
- `Ilock`.

Qlock

Qlocks can only be used by *processes*.

A `Qlock` has a queue of waiting processes associated with it. When a process gives up a `Qlock`, it wakes up the first process in the queue. This process can then get the lock.

Locks

`Locks` are simple spin locks:

```
void
lock(ulong *l)
{
        while(!tas(l))
                delay(1);
}
```

Such locks are not allowed to be held for long.

If a process holding a lock is preempted, other processes wanting the lock must wait until that process is scheduled again.

Real-time processes would suffer greatly, so when Plan 9 got a real-time scheduler, lock management changed: Processes holding a lock are not preempted until they give up the lock (a flag is tested in `unlock()`).

Interrupt routines cannot call `lock()` because this will cause deadlock if they interrupt the process holding the lock (on a uniprocesor). However, they are allowed to call `trylock()`.

For interrupt routines we have `ilock()`:

Ilock

`Ilock()` turns off interrupts and obtains a lock.

Thus, while an `Ilock` is being held, no interrupts can occur. Therefore, a kernel process or an interrupt routine will always be able to get an `Ilock` (immediately on a uniprocessor, perhaps after some *spinning* on a multiprocessor).

`Ilocks`, like `Locks`, may not be held long, because they delay interrupt processing and real-time processing. Some years ago, we measured that the maximum scheduling delay for real-time processes due to locks is on the order of 50 μs (200 MHz Pentium). That's not too bad.