Parallel Computing
CM30225

Russell Bradford

2023/24

### 1. Concurrency Primitives

#### Locks

Locks are definitely needed when we update (read then modify) the value of a variable

The question arises regarding whether we need a lock around a simple read of a multi-byte value, such as a 32-bit (4 byte) integer

It is easy to believe some bytes of a value might be written while half-way through being read, resulting in a mix of the bits of the old and new values

Called read (or write) *tearing*

### 2. Concurrency Primitives

#### Locks

However, for most (non-embedded) machine architectures these days it is likely (not certain!) to be safe to read simple values like integers or doubles that fit in a register: the hardware read is atomic (another side effect of the caching mechanism)

Though you do need to be careful on strange machine architectures, or with compilers that try to be too clever (For hackers: think about non-aligned accesses)

Certainly, though, for reading all of a larger object or structure, a lock will be necessary to ensure consistency across the multiple machine reads it takes to read in the whole structure

### 3. Concurrency Primitives

#### Locks

int x, y;
...
y = x;

Usually safe as reads of ints are generally atomic

### 4. Concurrency Primitives

#### Locks

// Also OO classes or objects
struct rational {
 int num, den;
};
struct rational r, s;
...
r = s;

Possibly unsafe, as it could take two machine reads to get all of s, which might be modified halfway through by another thread

Unlikely, but you can’t rely on that

Analogously for the write of r

### 5. Concurrency Primitives

#### Locks

**Exercise** For C geeks. There is an aliasing problem with bit fields in a struct

struct {
 int a: 5;
 int b: 3;
}

where an update to field a might be implemented as a read of a byte, modifying the bits of a, then writing a byte. Investigate

**Exercise** What about a 128-bit long long int on a 64-bit machine?

### 6. Concurrency Primitives

#### Locks

What about when we need to use more than one lock?

Of course, we can and should have separate locks in order to protect separate resources: we *could* use countlock to protect updates to another shared variable sum, but that would prevent one thread updating count while another is updating sum, which is perfectly safe to do

The only real reason to share a lock like this would be in when there are severe memory limitations: but lock implementations tend to use only a little memory per lock

### 7. Concurrency Primitives

#### Locks

But we do need to be careful about what we protect from what as it all has a cost

Getting and releasing a lock can be relatively cheap (in some architectures and operating systems; expensive in others) but it is not free: it is an overhead to be taken into account and avoided if you can

In many implementations these days the cost of getting an uncontended lock (not already locked) is cheap, while the cost of getting a lock that is already held is expensive

So the common (you hope) case is cheap

### 8. Concurrency Primitives

#### Locks

Also note, locks can be used to protect anything, not just variables, e.g., whole function calls or whole loops. But we should try too keep the regions small

get\_lock(mux);
someone\_elses\_dodgy\_code();
free\_lock(mux);

Another reason to use a single lock could be that the code you want to protect is so complicated you are not clear on how to proceed!

### 9. Concurrency Primitives

#### Locks

Locks are a simple, low level mechanism

Too low level: they are easy to use incorrectly

Suppose we have a couple of variables x and y we are protecting with mutexes mx and my respectively. We want to swap their values; elsewhere replace them both by their average

tmp = x; av = (x+y)/2;
x = y; x = av;
y = tmp; y = av;

### 10. Concurrency Primitives

#### Locks

To make this safe we have to use both locks

get\_lock(mx);
get\_lock(my);
tmp = x;
x = y;
y = tmp;
free\_lock(my);
free\_lock(mx);

### 11. Concurrency Primitives

#### Locks

Some pages of code later

get\_lock(my);
get\_lock(mx);
av = (x+y)/2;
x = av;
y = av;
free\_lock(mx);
free\_lock(my);

Spot the bug!

### 12. Concurrency Primitives

#### Locks

This will probably work most of the time, but occasionally just hangs doing nothing

Sometimes we will get

1 2
get\_lock(mx); get\_lock(my);
get\_lock(my); (waits) get\_lock(mx); (waits)

This is simple deadlock, another race condition

### 13. Concurrency Primitives

#### Locks

A very easy error to make, but often very difficult to find, particularly as the locks of mx and my may be widely separated in the code, or in someone else’s code

The use of locks requires a great deal of careful management when the code gets large

**Exercise** Why wouldn’t having another mutex mxy to protect both x and y solve things?

### 14. Concurrency Primitives

#### Locks

If we want to use a lock in portable code, we can use a library specification like *POSIX*

This is a standard that covers a large number of functions, specifying their use and behaviour

### 15. Concurrency Primitives

#### POSIX pthread

The pthread section on the POSIX specification contains several functions that we shall soon be looking at:

* Locks: pthread\_mutex\_ init, lock, unlock, destroy
* Barriers: pthread\_barrier\_ init, wait, destroy
* Condition Variables: pthread\_cond\_ init, wait, signal, broadcast, destroy
* Semaphores: sem\_ init, post, wait, destroy
* Management: pthread\_ create, join

And many others

### 16. Concurrency Primitives

#### POSIX pthread

For example, pthread\_create (we shall come back to this later)

#include <pthread.h>
int pthread\_create(pthread\_t \*thread,
 const pthread\_attr\_t \*attr,
 void \*(\*start\_routine) (void \*),
 void \*arg);
}

is how to create a new thread: it takes an *attribute* (always NULL for our purposes), a function of one argument to start executing, and a value to pass as the argument to that function

It returns a *thread identifier* in the first argument

### 17. Concurrency Primitives

#### POSIX pthread

Documentation for POSIX pthread functions is available everywhere, online and possibly on your own computer

For example, on Linux you can use manual pages, e.g.,
man pthread\_create
to get detailed information

### 18. Concurrency Primitives

#### POSIX Locks

A real example of locks, as defined by the POSIX standard, where they are called mutexes

#include <pthread.h>
pthread\_mutex\_t mutex;

An (uninitialised) mutex

### 19. Concurrency Primitives

#### POSIX Locks

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,
 const pthread\_mutexattr\_t
 \*restrict attr)

Initialises the mutex pointed at by the first argument, returns a 0 that indicates success or non-0 to indicate failure

POSIX locks come with various attributes: the default (NULL) is normally what you want

pthread\_mutex\_t mut;
if (pthread\_mutex\_init(&mut, NULL) != 0) { ...error... }

### 20. Concurrency Primitives

#### POSIX Locks

There is a alternative static way to initialise mutexes if all you need is a basic lock:

// declare and initialise
pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

### 21. Concurrency Primitives

#### POSIX Locks

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);
int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

The main grab and free functions

It is an error to try and unlock a mutex that is held by another thread: the thread that locks must be the thread that unlocks

This is a POSIX specification designed to make locks widely implementable of a variety of architectures

And this is not a limitation: it is a desired behaviour. If you allowed another thread to unlock a mutex you can bet this would be misused by some programmers thus opening a new opportunity to write buggy code

### 22. Concurrency Primitives

#### POSIX Locks

“It is an error”: some implementations return an error value, while others (depending on the OS) have undefined behaviour

Some versions of mutexes also allow *recursive* (or *reentrant*) locking, where a thread that already owns a lock can lock it again; it needs to do the same number of unlocks to free the lock

Non-recursive versions just self-deadlock, or have undefined behaviour

### 23. Concurrency Primitives

#### POSIX Locks

On fairness of POSIX mutexes:

Posix says “the scheduling policy shall determine which thread shall acquire the mutex” if more than one is waiting

This allows implementations to take pthread\_attr\_setschedpolicy and thread priorities into account: we shall not talk about that here!

### 24. Concurrency Primitives

#### POSIX Locks

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

Like pthread\_mutex\_lock but return immediately (without getting the lock) if the lock was already held. It returns a value of 0 if it got the lock, a non-zero otherwise <+(0)->

This function is occasionally useful, but less than you might believe, as the result doesn’t quite mean what people think it means (sequential assumptions…)

### 25. Concurrency Primitives

#### POSIX Locks

It doesn’t say “the mutex *is* locked”, but really says “the mutex *was* locked”

It gives the instantaneous state of the lock at the time of the trylock function call: it is possible that by the time the calling thread looks at the value that was returned by trylock the lock is already free

### 26. Concurrency Primitives

#### POSIX Locks

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

It’s good to clear up when you no longer need the mutex as this may free up some system resources

### 27. Concurrency Primitives

#### POSIX Locks

Example code:

#include <pthread.h>
...
pthread\_mutex\_t m;
/\* ought to check values returned by these calls \*/
pthread\_mutex\_init(&m, NULL);
...
pthread\_mutex\_lock(&m);
... <CR> ...
pthread\_mutex\_unlock(&m);
...
pthread\_mutex\_destroy(&m);

We can lock and unlock a mutex as often as we wish: we would typically create it once and use it many times before tidying up

### 28. Concurrency Primitives

#### POSIX Locks

The properties of POSIX locks are specified just to the point to make them useful: in a portable program you can’t rely on any feature not explicitly mentioned

For example, calling destroy on an uninitialised lock; or calling init on an already-initialised lock; or destroying a lock while another thread holds it; or using a bitwise copy of a lock structure; and so on

Remember that a lot of machines don’t have the nice predictable architecture of a PC

And even PC architectures are very complicated these days

### 29. Concurrency Primitives

#### POSIX pthread

**Exercise** Read about <pthread_spin_lock> and <pthread_rwlock>

**Advanced Exercise** Think about mutexes in the context of async programming, where we have concurrency (but not necessarily parallelism) and we require threads never to block

### 30. How to make threads

Now we have been introduced to POSIX, we need to take a little diversion from talking about primitives to cover something essential to parallelism

Namely, how do we create new threads to run?

As always, a simple idea that can have unexpected consequences

We shall look at the POSIX mechanism

### 31. Concurrency Control

#### POSIX

Creating threads:

#include <pthread.h>
int pthread\_create(pthread\_t \*thread,
 const pthread\_attr\_t \*attr,
 void \*(\*start\_routine) (void \*),
 void \*arg);

Link with -lpthread

This looks ugly, but is quite simple in practice: it creates a new thread running the function start\_routine on the argument arg

### 32. Concurrency Control

#### POSIX

It returns a thread identifier in argument thread. This can be used to do things to the thread

attr is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread)

The start\_routine names a function of one argument that the thread will start executing when it begins running

The arg is the argument passed to the function (a pointer)

### 33. Concurrency Control

#### POSIX

Roughly:

void \*hello(void \*n)
{
 printf("hello %
 return n;
}

int main(void)
{
 int m;
 pthread\_t thr;

 m = 1;
 // should check return value from create ...
 pthread\_create(&thr, NULL, hello, (void\*)&m);
 ...
}

### 34. Concurrency Control

#### POSIX

pthread\_create returns (pretty much) immediately with an error code, 0 indicating success

It makes a new thread that runs separately from the main thread

Possibly simultaneously with the main thread, depending on the number of cores and the OS’s scheduling

### 35. Concurrency Control

#### POSIX

It runs the function hello with argument a pointer to m

It does this concurrently with the main function, which continues to run

The start\_function will generally call lots of other functions to perform whatever the thread needs to do

Ugly type casting is common in C

### 36. Threads

#### Aside

This also works on uniprocessor systems: the threads are scheduled in a similar way to processes

You can debug a concurrent program on a sequential machine, but it may not exhibit some of the more subtle race conditions or deadlocks as the threads won’t truly be running in parallel

### 37. Threads

#### Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not to have more threads than cores, as both creating threads and OS scheduling eats up CPU time

A common error is to create hundreds of threads and then wonder why everything is running slowly

Threads create concurrency, not parallelism