Parallel Computing
CM30225

Russell Bradford

2023/24

### 1. Concurrency Primitives

#### Synchronisation

Now we look at some other problems

Consider our original counting code with a shared variable count. A simple solution might be to make count non-shared:

1 2
for (i = 0; i < 50; i++) { for (j = 50; j < 100; j++) {
 if (val[i] > 0) if (val[j] > 0)
 count1 = count1 + 1; count2 = count2 + 1;
} }
count = count1 + count2;

There is now another, different, problem with this code!

### 2. Concurrency Primitives

#### Synchronisation

The problem now is *when is the count = count1 + count2 executed?*

To be correct, it has to happen after both the loops have finished: any earlier will give a wrong answer

It will definitely happen after loop **1** has finished, but what about loop **2**?

We can’t rely (in a MIMD architecture) on the two loops on different cores running at the same time and finishing at the same time

Timings in the system may have the two loops running in any conceivable arrangement of before, after or overlapped

### 3. Concurrency Primitives

#### Synchronisation

1 2
for (i = 0; i < 50; i++) {
 if (val[i] > 0)
 count1 = count1 + 1;
} for (j = 50; j < 100; j++) {
count = count1 + count2; if (val[j] > 0)
 count2 = count2 + 1;
 }

### 4. Concurrency Primitives

#### Synchronisation

So we must explicitly write code to ensure the final sum only happens when both loops are finished

This is a *synchronisation* between the two threads

It may mean thread **1** waiting for thread **2**

Another sequentialisation!

### 5. Concurrency Primitives

#### Synchronisation

More subtly: if this code is executed more than once (perhaps counting more than one array), thread **2** ought to wait for thread **1** before starting!

It is possible that **1** is slow or paused for some reason, when **2** might do its bit and come around again on the next call to the count code, do the count on some other data, updating count2 as it goes

Finally **1** awakes and gets the wrong count2

This does happen and is a source of bugs

### 6. Concurrency Primitives

#### Semaphores

*Semaphores* can be used for thread synchronisation

Typically, we might have some thread that can only continue its work when one (or more) others have finished doing something, maybe computing some inputs for the thread to process

It can wait on a semaphore, again a simple flag, until another thread sets the flag. Then it knows it can continue

Note that even though both locks and semaphores are flags, they are very different things! Beware it is common for people to confuse the two

### 7. Concurrency Primitives

#### Semaphores

Semaphores are manipulated by two atomic operations P and V that symbolically act atomically as:

P(s): while (s == 0) { V(s): s = 1;
 suspend(); if any process waiting on s
 } unblock one
 s = 0;

### 8. Concurrency Primitives

#### Semaphores

On finding $s=0$ a thread will suspend itself; when awoken it will re-attempt to set the semaphore: and it will often succeed, unless a third thread comes along and gets the semaphore first

Like locks, semaphores are *not fair* on which thread will be awoken if more than one is waiting

Other names for P are: wait, up, lock, enter, open

Other names for V are: signal, down, unlock, exit, close

P stands for “proberen”, V for “verhogen”, which are Dutch for “test” and “increase”

### 9. Concurrency Primitives

#### Semaphores

Semaphores synchronise across threads:

do something
wait(s) prepare data
read data signal(s)
 carry on

 prepare data
do something signal(s)
wait(s) carry on
read data

Thread 1 waits until thread 2 has prepared some data before reading it

The signal and wait might happen in any order

### 10. Concurrency Primitives

#### Counting Semaphores

The above are called *binary* semaphores as the idea can be trivially extended into *counting* semaphores

P(s): while (s == 0) { V(s): s = s + 1;
 suspend(); if any process waiting on s
 } unblock one
 s = s - 1;

When initialised with the value $n$, this will allow $n$ threads to open the semaphore before blocking

### 11. Concurrency Primitives

#### Counting Semaphores

This allows region access control when there can be one than one, but fewer than some limit in the region simultaneously

For example, if there are 5 places at a dining table we can allow no more than 5 people in the room at a time

Or 4 if they are philosophers…

### 12. Concurrency Primitives

#### Semaphores

Mutual exclusion with semaphores happens to be easy:

wait(s);
<CR>
signal(s);

Wait for the semaphore; signal it’s free when you are done

But don’t do this: it’s better to use locks here. Semaphores are more general than locks: they allow a thread to suspend itself and be awoken by another thread when some condition is true

### 13. Concurrency Primitives

#### Semaphores

Mutexes: the thread that sets the flag must be the thread that clears the flag

Semaphores: the thread that sets the flag will generally be different from the thread that clears the flag

Semaphores should be used *across* threads, mutexes must not

The locking effect is in some sense incidental: more useful is using semaphores to synchronise

### 14. Concurrency Primitives

#### POSIX Semaphores

POSIX semaphores:

#include <semaphore.h>
sem\_t sem;
int sem\_init(sem\_t \*sem, int pshared, unsigned int value);
int sem\_destroy(sem\_t \*sem);
int sem\_wait(sem\_t \*sem);
int sem\_post(sem\_t \*sem);
int sem\_trywait(sem\_t \*sem);

“post” for signal

### 15. Concurrency Primitives

#### POSIX Semaphores

**Exercise** Add a semaphore to the count1/count2 example to get thread 1 to wait for thread 2 before doing the final sum

**Exercise** Then add another semaphore to get thread 2 to wait for thread 1 before starting

### 16. Concurrency Primitives

#### Barriers

Another synchronisation primitive is *barriers* (occasionally called *rendezvous)*

A barrier stops threads from continuing until some required number of threads have all hit the barrier; then they can all continue together

This allows us to synchronise parts of the program: recall supersteps

### 17. Concurrency Primitives

#### Barriers

Suppose we have a list of numbers we want to square then add in pairs

for (i = 0; i < 100; i++) {
 v[i] = v[i]\*v[i];
}
for (i = 0; i < 100; i++) {
 s[i] = v[i] + v[99-i];
}

We can parallelise this by having (say) 4 threads; each thread squares a block of values; then they add a block of values

### 18. Concurrency Primitives

#### Barriers

1 2 3 4
v[0]^2 v[25]^2 v[50]^2 v[75]^2
v[1]^2 v[26]^2 v[51]^2 v[76]^2
v[2]^2 v[27]^2 v[52]^2 v[77]^2
... ... ... ...
v[24]^2 v[49]^2 v[74]^2 v[99]^2
v[0]+v[99] v[25]+v[74] v[50]+v[49] v[75]+v[24]
v[1]+v[98] v[26]+v[73] v[51]+v[48] v[76]+v[25]
... ... ... ...
v[24]+v[75] v[49]+v[50] v[74]+v[25] v[99]+v[0]

### 19. Concurrency Primitives

#### Barriers

1 2 3...
for (i = 0; i < 25; i++) { for (j = 25; j < 50; j++) {
 v[i] = v[i]\*v[i]; v[j] = v[j]\*v[j];
} }
for (i = 0; i < 25; i++) { for (j = 25; j < 50; j++) { ...
 s[i] = v[i] + v[99-i]; s[j] = v[j] + v[99-j];
} }

Again, the above might work sometimes, or many times, but it is buggy

### 20. Concurrency Primitives

#### Barriers

The problem here is again that the threads may not all be running at the same speed: perhaps one thread is interrupted and descheduled by the OS; or memory access is not uniform speed; or many other factors

So we can’t rely on all the threads finishing their squares at precisely the same time: one thread might finish its block and start adding using values not yet finished squaring

Another synchronisation problem

### 21. Concurrency Primitives

1 2 3 4
v[0]^2 v[25]^2 v[50]^2
v[1]^2 v[26]^2 v[51]^2
v[2]^2 v[27]^2 v[52]^2 v[75]^2
... ... ... v[76]^2
... ... ... ...
v[24]^2 v[49]^2 v[74]^2 v[97]^2
v[0]+v[99] v[25]+v[74] v[50]+v[49] v[98]^2
v[1]+v[98] v[26]+v[73] v[51]+v[48] v[99]^2
... ... ... v[75]+v[24]
... ... ... ...
v[24]+v[75] v[49]+v[50] v[74]+v[25] v[97]+v[2]
 v[98]+v[1]
 v[99]+v[0]

This is how we get the wrong answer: again just because the lines of code for the adds follows the lines of code for the squares make us believe every add happens after every square

### 22. Concurrency Primitives

#### Barriers

We need to synchronise all the threads at the end of the squares before allowing them to continue with the adds

 b = make\_barrier(4);
<parallel squares> <parallel squares> <parallel squares> ...
barrier\_wait(b); barrier\_wait(b); barrier\_wait(b); ...
<parallel adds> <parallel adds> <parallel adds> ...

Only when all 4 threads have reached the barrier can they all proceed

### 23. Concurrency Primitives

#### Barriers

Barriers are good for the superstep style of programming



Supersteps

But beware: as a barrier synchronises many threads, there is potentially a lot of waiting going on: we can’t progress faster than the slowest thread

Thus barriers are best when all the threads are doing roughly the same amount of work

### 24. Concurrency Primitives

#### POSIX Barriers

#include <pthread.h>
pthread\_barrier\_t barrier;
int pthread\_barrier\_init(
 pthread\_barrier\_t \*restrict barrier,
 const pthread\_barrierattr\_t \*restrict attr,
 unsigned count);
int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);
int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

A barrier can be reused immediately after it has released its threads; it has a fixed value of $n$ set when it is initialised

**Exercise** Have a look at the return value from pthread\_barrier\_wait

### 25. Concurrency Primitives

#### POSIX Barriers

**Exercise** Fix the count1/count2 problem with barriers

**Exercise** Both semaphores and barriers are about synchronisation. Think about how you might implement barriers using semaphores

**Exercise** Think about how you might implement semaphores using barriers

### 26. Concurrency Primitives

#### Condition Variables

One last primitive we are going to look at is *condition variables*

As the name suggests, it is a way a thread can wait until some condition is true

The idea is that one or more threads can wait on a condition variable until another signals that the required condition is now true

The signal can either let just *one* thread continue, or be a *broadcast* that lets all waiting threads continue

Condition variables are normally associated with a mutex, and are used *inside* a critical region protected by that mutex

### 27. Concurrency Primitives

#### Condition Variables

1 2
get\_lock(mx); get\_lock(mx);
<CR> <CR>
condvar\_wait(cv, mx); condvar\_signal(cv);
(wait) free\_lock(mx);
<CR>
free\_lock(mx);

condvar\_wait releases the mutex and waits on the condition variable

When the other thread signal signals and releases the mutex, the first thread regains the mutex and continues within the critical region

### 28. Concurrency Primitives

#### Condition Variables

The condition variable allows thread 1 to “step outside” the critical region, letting another thread to enter and do something

Conditions variables combine mutual exclusion and synchronisation

Again, not fair on which thread gets to continue if more than one is waiting

With a broadcast all other threads are marked as ready to run, but only one will regain the lock; the others will blocked on the lock as normal

One will get the lock when the first thread releases it; and so on

### 29. Concurrency Primitives

#### POSIX Condition Variables

#include <pthread.h>
int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,
 const pthread\_condattr\_t \*restrict attr);
int pthread\_cond\_destroy(pthread\_cond\_t \*cond);
int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,
 pthread\_mutex\_t \*restrict mutex);
int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond,
 pthread\_mutex\_t \*restrict mutex,
 const struct timespec \*restrict abstime);
int pthread\_cond\_signal(pthread\_cond\_t \*cond);
int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

### 30. Concurrency Primitives

#### POSIX Condition Variables

As an example of the kind of grungy detail that parallelism has to address: POSIX recognises that there is a nasty implementation detail that would otherwise make implementing condition variables impractical

The specification for pthread\_cond\_signal says

The pthread\_cond\_signal() function shall unblock at least one of the threads that are blocked on the specified condition variable cond

“at *least* one”: there is a (rare) problem of *spurious wakeups* that is in general too expensive to avoid

### 31. Concurrency Primitives

#### POSIX Condition Variables

This just means you have to be a bit formulaic about the use of condition variables and always have a *condition* to test before continuing

1 2
iteration = 0;
get\_lock(mx); get\_lock(mx);
<CR> <CR>
it = iteration; iteration++;
while (it == iteration) condvar\_signal(cv, mx);
 condvar\_wait(cv, mx); free\_lock(mx);
<CR>
free\_lock(mx);

Thread 1 might get awoken spuriously but it doesn’t want to continue until the next iteration

### 32. Concurrency Primitives

#### POSIX Condition Variables

In general you would test for whatever condition you were waiting for: thread 2 sets the condition, thread 1 should test for it

Condition variables are very useful, but a bit of a pain to use

### 33. Concurrency Primitives

#### Concurrency Primitives

We have called these things *primitives*, but we can implement them in terms of each other

**Exercise** Do this

All eventually go back to the underlying hardware or software support

“Primitive” is actually a good description as they are all very low level

### 34. Concurrency Primitives

#### Concurrency Primitives

And they do have a cost, thus their use does limit the speedup available

Their overhead can be divided into two parts

* the time spent blocked as a necessary part of its function, e.g., wait on a lock
* the time spent in executing the code of the primitive

Note part (a) isn’t really a limitation of the primitive: it’s necessary if it is to work at all. It is (b) that the implementation of a primitive seeks to minimise

### 35. Concurrency Control

#### Higher Level

Semaphores, locks, barriers, etc., and even threads are likened to assembler: low-level, fast, fine control, but very likely to encourage buggy programs

While many programmers are happy using them, others need higher level solutions

These come in many forms

### 36. Concurrency Control

#### Higher Level

Concurrency control can be supported in a high-level language as

* added in to an existing language, in library support. We have seen some of this already: the POSIX examples
* fudged into the syntax of an existing language
* part of the initial design of a new language

We shall be looking at all of these approaches

### 37. Concurrency Control

#### Higher Level

There is a lot of sequential code out there that people would like to run faster on parallel hardware

While there is a lot of effort being put into automatic analysis of code to discover and exploit parallelism, the results are sporadic

Functional languages offer a decent hope here, but not much code is functional style

So code needs to be rewritten to make best advantage of parallelism

The hope (and economics) is we can take existing code using an existing language and modify it

### 38. Concurrency Control

#### Libraries

It’s not a good way of doing things, but rewriting from scratch is just too expensive

Of course, new projects ought to be written with parallelism in mind from their start

Also, there are lots of programmers with extensive expertise in languages like C, Java and C++ — meaning such programmers are cheaper to employ

So we are led to the approach of taking, say C, and adding parallelism to it

The easiest way is to leave the language itself untouched, just adding a library of functions that do parallelism

### 39. Concurrency Control

#### Libraries

For example, the POSIX pthread approach

Note: We have been using C and the POSIX library to illustrate points, but this library technique applies to all sensible languages

But you can’t just add a parallel library to a sequential language and hope everything is OK

### 40. Concurrency Control

#### Threads again

Modern compilers and modern hardware both try their best to execute your code as fast as possible

But in doing so, they can break parallel code

For example, some compiler optimisations can break parallel code

And some hardware optimisations can break parallel code

### 41. Concurrency Control

#### Compiler Reordering

Modern compilers often reorder code to make things more efficient

For example, main memory access is (relatively) slow, so if the value of a variable is needed, the compiler might try to start loading it earlier than the code might suggest

### 42. Concurrency Control

#### Compiler Reordering

Given code

 y = 2;
 x = z;
 x += y; // need to wait for z before we can do this

The compiler might spot it can start loading z earlier, so there is less of a wait before it can do the increment:

 x = z;
 y = 2; // do this without waiting for z to be loaded
 x += y;

The effect is the same, but it goes a little faster. The compiler in effect rewrites your code

### 43. Concurrency Control

#### Compiler Reordering

This could break things. Consider

A B
while (cont == 0) {/\* nothing \*/} x = 42;
print x; cont = 1;

where the intent was to have thread A to wait for thread B to set the cont flag before continuing to print 42

A compiler only seeing the code for B may conclude that the variables cont and x are independent and so (perhaps for whatever reason) it can rearrange the code as

cont = 1;
x = 42;

### 44. Concurrency Control

#### Compiler Reordering

Similarly for A: it is possible that the read of x can done before the loop

Note: *never* write code like this in the hope that it might work: it is simply buggy code! Use a semaphore or equivalent

The problem is that there is a hidden relationship between the variables x and cont that is in the mind of the programmer, but is not expressed in the code

### 45. Concurrency Control

#### Compiler Reordering

Example. Consider the code:

int a = 0;
int b = 0;

A B
a = 42; b = 42;
printf("%

Explain how it might print 0 twice, even though it appears we always print after an update