Shared Memory Systems

Suppose we want to count the number of positive values in a list of numbers

```
count = 0;
for (i = 0; i < 100; i++) {
    if (val[i] > 0) { count = count + 1; }
}
```

In C or C++ or Java or whatever

Shared Memory Systems

Suppose we want to count the number of positive values in a list of numbers

```
count = 0;
for (i = 0; i < 100; i++) {
    if (val[i] > 0) { count = count + 1; }
}
```

In C or C++ or Java or whatever

It's not really worthwhile parallelising this in real life (**Exercise** why?), but let's try

Shared Memory Systems

We could split this into two blocks

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0)  count = count + 1;
}
```

2

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

and by magic to be discussed later have blocks 1 and 2 run in parallel on separate processors, sharing the variables (i.e., shared memory)

Shared Memory Systems

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0) {
      count = count + 1;
    }
}
```

2

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

Note we want to share `val` and `count`, but not the loop variables!

Shared Memory Systems

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0) {
      count = count + 1;
    }
}
```

2

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

Note we want to share `val` and `count`, but not the loop variables!

No communication or interaction between the threads: instant speedup of 2?

Shared Memory Systems

It may run twice as fast, but sometimes will give the wrong answer!

Shared Memory Systems

It may run twice as fast, but sometimes will give the wrong answer!

Sometimes it will give a value of `count` that is too small

Shared Memory Systems

It may run twice as fast, but sometimes will give the wrong answer!

Sometimes it will give a value of `count` that is too small

The problem is the *shared resource*, the variable `count`

Shared Memory Systems

It may run twice as fast, but sometimes will give the wrong answer!

Sometimes it will give a value of `count` that is too small

The problem is the *shared resource*, the variable `count`

We have two separate threads reading and updating the value

Shared Memory Systems

Occasionally, just occasionally, the following happens

1

```
read the value of count
into a CPU register
```

2

```
read the value of count
into a CPU register
```

Shared Memory Systems

Occasionally, just occasionally, the following happens

1

```
read the value of count
into a CPU register
add 1
```

2

```
read the value of count
into a CPU register
add 1
```

Shared Memory Systems

Occasionally, just occasionally, the following happens

1

```
read the value of count
into a CPU register
add 1
store the value
```

2

```
read the value of count
into a CPU register
add 1
store the value
```

Shared Memory Systems

So both read a value, 10, say. Both add 1 to get 11. Both store 11.

Even if we don't have hardware that supports simultaneous reads and writes (we might have EREW) it can still go wrong

Shared Memory Systems

So both read a value, 10, say. Both add 1 to get 11. Both store 11.

Even if we don't have hardware that supports simultaneous reads and writes (we might have EREW) it can still go wrong

1	**2**
`read the value of count`	`...`

Shared Memory Systems

So both read a value, 10, say. Both add 1 to get 11. Both store 11.

Even if we don't have hardware that supports simultaneous reads and writes (we might have EREW) it can still go wrong

1	**2**
`read the value of count`	`...`
`add 1`	`read the value of count`

Shared Memory Systems

So both read a value, 10, say. Both add 1 to get 11. Both store 11.

Even if we don't have hardware that supports simultaneous reads and writes (we might have EREW) it can still go wrong

1	**2**
`read the value of count`	`...`
`add 1`	`read the value of count`
`store the value`	`add 1`

Shared Memory Systems

So both read a value, 10, say. Both add 1 to get 11. Both store 11.

Even if we don't have hardware that supports simultaneous reads and writes (we might have EREW) it can still go wrong

1

```
read the value of count
add 1
store the value
...
```

2

```
...
read the value of count
add 1
store the value
```

Shared Memory Systems

The parallel version is simply an incorrect program

Shared Memory Systems

The parallel version is simply an incorrect program

This is another example of a *race condition* where an unexpected or overlooked timing in the execution produces an incorrect result

Shared Memory Systems

The parallel version is simply an incorrect program

This is another example of a *race condition* where an unexpected or overlooked timing in the execution produces an incorrect result

It is a *data race*: an unsynchronized, concurrent access to data involving a write

Shared Memory Systems

The parallel version is simply an incorrect program

This is another example of a *race condition* where an unexpected or overlooked timing in the execution produces an incorrect result

It is a *data race*: an unsynchronized, concurrent access to data involving a write

Read-only data is always safe to share: nothing can go wrong

Shared Memory Systems

The parallel version is simply an incorrect program

This is another example of a *race condition* where an unexpected or overlooked timing in the execution produces an incorrect result

It is a *data race*: an unsynchronized, concurrent access to data involving a write

Read-only data is always safe to share: nothing can go wrong

But when a write (or multiple writes) is involved, things can go badly wrong

Shared Memory Systems

And notice this can even happen on a single processor, when multiple threads are being timeshared by the OS

Shared Memory Systems

And notice this can even happen on a single processor, when multiple threads are being timeshared by the OS

The OS may choose to deschedule thread 1 in between its read and write; and schedule thread 2 that reads the old value

Shared Memory Systems

And notice this can even happen on a single processor, when multiple threads are being timeshared by the OS

The OS may choose to deschedule thread 1 in between its read and write; and schedule thread 2 that reads the old value

Exercise And it might give even worse counts: think why

Shared Memory Systems

And notice this can even happen on a single processor, when multiple threads are being timeshared by the OS

The OS may choose to deschedule thread 1 in between its read and write; and schedule thread 2 that reads the old value

Exercise And it might give even worse counts: think why

So this is a concurrency error, and not just a parallelism error

Shared Memory Systems

The race may or may not happen according all kinds of external events that might affect the timing of the execution of the updates

Shared Memory Systems

The race may or may not happen according all kinds of external events that might affect the timing of the execution of the updates

So the program may often be right, and occasionally wrong

Shared Memory Systems

The race may or may not happen according all kinds of external events that might affect the timing of the execution of the updates

So the program may often be right, and occasionally wrong

Or the program may often be wrong, and occasionally right

Shared Memory Systems

The race may or may not happen according all kinds of external events that might affect the timing of the execution of the updates

So the program may often be right, and occasionally wrong

Or the program may often be wrong, and occasionally right

The program might always give the correct answer on your machine, but give the wrong answer on your customer’s machine

Shared Memory Systems

The race may or may not happen according all kinds of external events that might affect the timing of the execution of the updates

So the program may often be right, and occasionally wrong

Or the program may often be wrong, and occasionally right

The program might always give the correct answer on your machine, but give the wrong answer on your customer's machine

Exercise Compare with deadlocks

Shared Memory Systems

Note: the “obvious solution” of having separate `count1` and `count2` introduces a new, separate, problem we shall address later: for now we need to consider shared resources

Races

Philosophy Exercise A race condition is only a bug if the non-determinism is undesirable. Discuss

Shared Memory Systems

The myriad ways of avoiding race conditions are what keep programmers and theoreticians in their jobs

Shared Memory Systems

The myriad ways of avoiding race conditions are what keep programmers and theoreticians in their jobs

And the people designing debugging tools

Shared Memory Systems

The myriad ways of avoiding race conditions are what keep programmers and theoreticians in their jobs

And the people designing debugging tools

Some debugging tools exist which will find simple errors like the above, but in general we have to rely on programmers finding the bugs by thinking

Shared Memory Systems

Race Condition Detection Tools

Some tools to help detect race conditions:

- Intel Parallel Inspector, a Visual Studio plugin
- Helgrind, a Valgrind plugin
- Data Race Detection (DRD), another Valgrind plugin

Shared Memory Systems

Race Condition Detection Tools

Some tools to help detect race conditions:

- Intel Parallel Inspector, a Visual Studio plugin
- Helgrind, a Valgrind plugin
- Data Race Detection (DRD), another Valgrind plugin

Ideally, the programming language itself would prevent you from writing code with races (see later for examples)

Shared Memory Systems

Race Condition Detection Tools

Some tools to help detect race conditions:

- Intel Parallel Inspector, a Visual Studio plugin
- Helgrind, a Valgrind plugin
- Data Race Detection (DRD), another Valgrind plugin

Ideally, the programming language itself would prevent you from writing code with races (see later for examples)

Experience tells us it is hopeless to rely on the programmer to get it right!

Shared Memory Systems

Areas of code that use a shared resource are called a *critical region* (also called a critical *section*)

Shared Memory Systems

Areas of code that use a shared resource are called a *critical region* (also called a critical *section*)

In the above example, the increments of `count` form a (small) critical region

Shared Memory Systems

Areas of code that use a shared resource are called a *critical region* (also called a critical *section*)

In the above example, the increments of `count` form a (small) critical region

A critical region comprises any pieces of code that access a resource that might be updated in parallel

Shared Memory Systems

Areas of code that use a shared resource are called a *critical region* (also called a critical *section*)

In the above example, the increments of `count` form a (small) critical region

A critical region comprises any pieces of code that access a resource that might be updated in parallel

So, in this example, *any* region of code that updates `count` is critical

Shared Memory Systems

Areas of code that use a shared resource are called a *critical region* (also called a critical *section*)

In the above example, the increments of `count` form a (small) critical region

A critical region comprises any pieces of code that access a resource that might be updated in parallel

So, in this example, *any* region of code that updates `count` is critical

So these pieces of code have to be carefully thought out to avoid race conditions

Shared Memory Systems

Such critical regions are rife in parallel programs and appear in many different guises

Shared Memory Systems

Such critical regions are rife in parallel programs and appear in many different guises

Sometimes you can run a program 100 times and get the right answer, but on the 101st time it is wrong

Shared Memory Systems

Such critical regions are rife in parallel programs and appear in many different guises

Sometimes you can run a program 100 times and get the right answer, but on the 101st time it is wrong

Such events can have a very low probability, making them hard to debug by “run it and see if it works”

Shared Memory Systems

Such critical regions are rife in parallel programs and appear in many different guises

Sometimes you can run a program 100 times and get the right answer, but on the 101st time it is wrong

Such events can have a very low probability, making them hard to debug by “run it and see if it works”

But they do happen, so you have to find them by hard thought instead

Shared Memory Systems

Locks

The problem is that two (or more) threads are trying to update something at the same time (update $=$ read, modify, write)

The problem is that two (or more) threads are trying to update something at the same time (update $=$ read, modify, write)

In between the read and the write another thread might have gone behind the first's back and updated the thing itself

Shared Memory Systems

Locks

The simplest solution to stop multiple threads updating a resource is to allow only *one* thread at a time to do an update on a shared resource

Shared Memory Systems

Locks

The simplest solution to stop multiple threads updating a resource is to allow only *one* thread at a time to do an update on a shared resource

If a second thread wishes to update while a first has already started, the second is forced to wait until the first has finished

Shared Memory Systems

Locks

The simplest solution to stop multiple threads updating a resource is to allow only *one* thread at a time to do an update on a shared resource

If a second thread wishes to update while a first has already started, the second is forced to wait until the first has finished

This will ensure correct updates by avoiding the update overlap we saw earlier

Shared Memory Systems

Locks

The simplest solution to stop multiple threads updating a resource is to allow only *one* thread at a time to do an update on a shared resource

If a second thread wishes to update while a first has already started, the second is forced to wait until the first has finished

This will ensure correct updates by avoiding the update overlap we saw earlier

Note, though, the second thread will have to wait: this is an inefficiency and if that happens a lot the system as a whole will be slower than it ought

We are forcing the bits of code in the critical region into executing sequentially, which Amdahl tells us is bad for speedup

Concurrency Primitives

Locks

We are forcing the bits of code in the critical region into executing sequentially, which Amdahl tells us is bad for speedup

But the sequential execution is essential for the code to be correct

Concurrency Primitives

Locks

We are forcing the bits of code in the critical region into executing sequentially, which Amdahl tells us is bad for speedup

But the sequential execution is essential for the code to be correct

So we need to make critical regions as small and fast as possible

One simple way of enforcing this *mutual exclusion* on critical regions is the use of *locks*

Concurrency Primitives

Locks

One simple way of enforcing this *mutual exclusion* on critical regions is the use of *locks*

Also called: mutexes. Some confused people use *semaphores* (see later), but these are better employed for other problems

Concurrency Primitives

Locks

One simple way of enforcing this *mutual exclusion* on critical regions is the use of *locks*

Also called: mutexes. Some confused people use *semaphores* (see later), but these are better employed for other problems

A lock is a simple flag that says “Please wait, this region is busy”

Concurrency Primitives

Locks

We must surround all critical regions that update a given shared resource with a grab and release of the lock:

```
get lock                    get lock
do stuff on a resource      other stuff on same resource
release lock                release lock
```

Concurrency Primitives

Locks

We must surround all critical regions that update a given shared resource with a grab and release of the lock:

```
get lock                      get lock
do stuff on a resource        other stuff on same resource
release lock                  release lock
```

If a second thread tries to grab the lock it will be made to wait until the lock is released by the first thread

Concurrency Primitives

Locks

We must surround all critical regions that update a given shared resource with a grab and release of the lock:

```
get lock                      get lock
do stuff on a resource        other stuff on same resource
release lock                  release lock
```

If a second thread tries to grab the lock it will be made to wait until the lock is released by the first thread

In this way we can ensure that two updates never overlap

Concurrency Primitives

We will get either

```
get lock                       try to get lock
do stuff on a resource         (wait)
release lock                   (wait)
                               get lock
                               other stuff on same resource
                               release lock
```

or

```
try to get lock                get lock
(wait)                         other stuff on same resource
(wait)                         release lock
get lock
do stuff on a resource
release lock
```

No parallelism on access to the resource!

Concurrency Primitives

Locks

Note that *every* piece of parallel code in the program that updates that resource will have to have to be wrapped in the grab of the lock

Concurrency Primitives

Locks

Note that *every* piece of parallel code in the program that updates that resource will have to have to be wrapped in the grab of the lock

If we miss protecting *any* occurrence of a parallel update, the whole thing is broken

Concurrency Primitives

Locks

Note that *every* piece of parallel code in the program that updates that resource will have to have to be wrapped in the grab of the lock

If we miss protecting *any* occurrence of a parallel update, the whole thing is broken

This is clearly a good source of bugs

Concurrency Primitives

Locks

Note that *every* piece of parallel code in the program that updates that resource will have to have to be wrapped in the grab of the lock

If we miss protecting *any* occurrence of a parallel update, the whole thing is broken

This is clearly a good source of bugs

Locks are a very crude method to prevent race conditions, but they are widely used

Concurrency Primitives

Locks

This also applies to more than two threads, of course

Concurrency Primitives

Locks

This also applies to more than two threads, of course

The first grab of the lock will succeed, the others will have to wait until the lock is released

Concurrency Primitives

Locks

This also applies to more than two threads, of course

The first grab of the lock will succeed, the others will have to wait until the lock is released

If more than one thread tries to grab the lock at the same instant, just one will succeed. The others will have to wait

Concurrency Primitives

Locks

This also applies to more than two threads, of course

The first grab of the lock will succeed, the others will have to wait until the lock is released

If more than one thread tries to grab the lock at the same instant, just one will succeed. The others will have to wait

If there are several threads waiting on a lock, just one will get the lock when it is released: the other threads continue to wait

Concurrency Primitives

Locks

Also, most implementations of locks are *not fair* in the sense that *any* one of the waiting threads will get the lock, there's no first-in-first-out enforced

Concurrency Primitives

Locks

Also, most implementations of locks are *not fair* in the sense that *any* one of the waiting threads will get the lock, there's no first-in-first-out enforced

This is because (a) it's extra overhead for the OS to implement such a FIFO and (b) most programs don't need it, so why have an overhead that most programs don't want?

Concurrency Primitives

Locks

Also, most implementations of locks are *not fair* in the sense that *any* one of the waiting threads will get the lock, there's no first-in-first-out enforced

This is because (a) it's extra overhead for the OS to implement such a FIFO and (b) most programs don't need it, so why have an overhead that most programs don't want?

The threads are likely arriving at the lock in a non-deterministic order, so what's the sense in preserving that random order?

Concurrency Primitives

Locks

Also, **it's bad practice for the programmer to rely on the order of things happening in a parallel system**

Concurrency Primitives

Locks

Also, **it's bad practice for the programmer to rely on the order of things happening in a parallel system**

If certain things need to happen in a certain order, the programmer must write code to ensure that this happens

Concurrency Primitives

Locks

Also, **it's bad practice for the programmer to rely on the order of things happening in a parallel system**

If certain things need to happen in a certain order, the programmer must write code to ensure that this happens

You can't rely on luck, or that they usually happen in the right order

Concurrency Primitives

Locks

Also, **it's bad practice for the programmer to rely on the order of things happening in a parallel system**

If certain things need to happen in a certain order, the programmer must write code to ensure that this happens

You can't rely on luck, or that they usually happen in the right order

Also note that specifying orders on events is another form of sequentiality, which we would like to minimise

Concurrency Primitives

Locks

Often, the wait on the lock is implemented and enforced by the operating system, which might deschedule the waiting thread to free up the CPU for something else to run

Concurrency Primitives

Locks

Often, the wait on the lock is implemented and enforced by the operating system, which might deschedule the waiting thread to free up the CPU for something else to run

With this kind of lock implementation, a thread takes no CPU time while locked

Concurrency Primitives

Locks

Often, the wait on the lock is implemented and enforced by the operating system, which might deschedule the waiting thread to free up the CPU for something else to run

With this kind of lock implementation, a thread takes no CPU time while locked

Thus the overhead of this lock is the CPU time it takes for the OS to deschedule and later reschedule the thread (not trivial!)

Concurrency Primitives

Spinlocks

In contrast, sometimes the lock wait is implemented as a *busy wait*: the thread keeps trying in a tight (busy) loop to grab the lock, continually burning CPU cycles

Concurrency Primitives

Spinlocks

In contrast, sometimes the lock wait is implemented as a *busy wait*: the thread keeps trying in a tight (busy) loop to grab the lock, continually burning CPU cycles

These are called *spinlocks*

Concurrency Primitives

Spinlocks

In contrast, sometimes the lock wait is implemented as a *busy wait*: the thread keeps trying in a tight (busy) loop to grab the lock, continually burning CPU cycles

These are called *spinlocks*

The argument is that critical regions should be small to maintain efficiency, so it will only be a short time before the lock will be released

Concurrency Primitives

Spinlocks

In contrast, sometimes the lock wait is implemented as a *busy wait*: the thread keeps trying in a tight (busy) loop to grab the lock, continually burning CPU cycles

These are called *spinlocks*

The argument is that critical regions should be small to maintain efficiency, so it will only be a short time before the lock will be released

And by the time the OS has descheduled the waiting thread the lock could already be free, so instead just keep busy trying

Concurrency Primitives

Spinlocks

This is good for when responsiveness is more important than CPU cost, e.g., real-time systems, but too expensive for many systems

Concurrency Primitives

Spinlocks

This is good for when responsiveness is more important than CPU cost, e.g., real-time systems, but too expensive for many systems

Note that spinlocks use CPU cycles, thus occupying the CPU, while blocking locks release the CPU so it can potentially used for something else

Concurrency Primitives

Spinlocks

You should take care over using spinlocks rather than blocking locks

Concurrency Primitives

Spinlocks

You should take care over using spinlocks rather than blocking locks

They assume that the holding thread only holds the lock for a brief time: but the holding thread can be preempted by the OS at any time

Concurrency Primitives

Spinlocks

You should take care over using spinlocks rather than blocking locks

They assume that the holding thread only holds the lock for a brief time: but the holding thread can be preempted by the OS at any time

Thus preventing release of the lock for an arbitrarily long period of time

Concurrency Primitives

Spinlocks

Exercise And read about the cache-thrashing behaviour that occurs if the spinlock is not implemented carefully

Concurrency Primitives

Spinlocks

Exercise And read about the cache-thrashing behaviour that occurs if the spinlock is not implemented carefully

> *. . . do not use spinlocks in user space, unless you actually know what you're doing. And be aware that the likelihood that you know what you are doing is basically nil*
>
> Linus Torvalds

A hybrid implementation will spin for a short while, then pass to the OS: trying to get the best of both approaches

Though there is still great debate over the best approach

Concurrency Primitives

Locks

To use a lock, in pseudocode:

```
countlock = make_a_new_lock();
...
get_lock(countlock);         get_lock(countlock);
count = count + 1;           count = 2*count;
free_lock(countlock);        free_lock(countlock);
```

Concurrency Primitives

Locks

To use a lock, in pseudocode:

```
countlock = make_a_new_lock();
...
get_lock(countlock);          get_lock(countlock);
count = count + 1;            count = 2*count;
free_lock(countlock);         free_lock(countlock);
```

Remember we must put a grab and release of the `countlock` around *all* updates to `count` in code where there might be more than one thread wanting to update the value

Concurrency Primitives

Locks

For most programming languages it is the responsibility of the programmer to spot all the shared resources that need this treatment and to write correct code to enforce exclusive access

Concurrency Primitives

Locks

For most programming languages it is the responsibility of the programmer to spot all the shared resources that need this treatment and to write correct code to enforce exclusive access

Getting this wrong (e.g., overlooking an update to `count` and not putting in the lock) is the source of one of the most common bugs in parallel programming

Concurrency Primitives

Locks

For most programming languages it is the responsibility of the programmer to spot all the shared resources that need this treatment and to write correct code to enforce exclusive access

Getting this wrong (e.g., overlooking an update to `count` and not putting in the lock) is the source of one of the most common bugs in parallel programming

Particularly for programmers trained in sequential programming; for sequential programs *all* accesses are already sequential!

Also, be careful not to over-lock

Concurrency Primitives

Locks

Also, be careful not to over-lock

We don’t need locks when there can only be one thread updating `count`, e.g., in a non-parallel part of the code, or we are already in some protected larger critical region

Concurrency Primitives

Locks

Also, be careful not to over-lock

We don’t need locks when there can only be one thread updating `count`, e.g., in a non-parallel part of the code, or we are already in some protected larger critical region

Over-locking is safe, but simply wastes time and thereby reduces speedup