Concurrency Primitives

Transactional Memory

Some machine architectures also support more sophisticated mechanisms
Some machine architectures also support more sophisticated mechanisms

For example, Intel have *Transactional Synchronization eXtensions* (TSX) in their latest chips, that allows multiple threads to work on shared data with potentially less lock contention.
Concurrence Primitives

Transactional Memory

Some machine architectures also support more sophisticated mechanisms

For example, Intel have *Transactional Synchronization eXtensions* (TSX) in their latest chips, that allows multiple threads to work on shared data with potentially less lock contention

This is somewhat like the way databases deal with parallel updates in such a way the common case of non-conflicting updates is dealt with efficiently, while conflicting updates are a bit more expensive (using rollbacks)
Concurrency Primitives

Transactional Memory

Some machine architectures also support more sophisticated mechanisms.

For example, Intel have *Transactional Synchronization eXtensions* (TSX) in their latest chips, that allows multiple threads to work on shared data with potentially less lock contention.

This is somewhat like the way databases deal with parallel updates in such a way the common case of non-conflicting updates is dealt with efficiently, while conflicting updates are a bit more expensive (using rollbacks).

But done in hardware, in special machine instructions.
Similarly, ARM has *Reservations* that watch out for simultaneous updates to memory locations.
Concurrency Primitives
Reservations

Similarly, ARM has *Reservations* that watch out for simultaneous updates to memory locations.

Of course, these need care from the programmer to use correctly, probably working in assembler.
Similarly, ARM has Reservations that watch out for simultaneous updates to memory locations.

Of course, these need care from the programmer to use correctly, probably working in assembler.

Exercise. Read about Transactional Memory and Reservations.
Concurrent Programming

Locks

Locks are definitely needed when we update (read then modify) the value of a variable.
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.
Concurreny 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.
Concurrenty 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.
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)
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)
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
int x, y;
...
y = x;

Usually safe as reads of ints are generally atomic
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
// 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
// Also 00 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.
Exercise for C geeks. There is an aliasing problem with bit fields in a struct

```c
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
Of course, we should have separate locks in order to protect separate resources: we could use countlock to protect updates to another shared variable \(\text{sum}\), but that would prevent one thread updating \(\text{count}\) while another is updating \(\text{sum}\), 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: lock implementations tend to use only a little memory per lock.
Back to locks

Of course, we 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.
Back to locks

Of course, we 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: lock implementations tend to use only a little memory per lock.
Concurrency Primitives

Locks

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

```c
get_lock(mux);
someone_elses_dodgy_code();
free_lock(mux);
```
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!
Concurrency Primitives

Locks

Locks are a simple, low level mechanism

tmp = x; av = (x+y)/2;
x = y; x = av;
y = tmp; y = av;
Locks are a simple, low level mechanism

Too low level: they are easy to use incorrectly
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 $m_x$ and $m_y$ respectively. We want to swap their values; elsewhere replace them both by their average

\[
\begin{align*}
\text{tmp} &= x; \\
\text{av} &= (x+y)/2; \\
x &= y; \\
x &= \text{av}; \\
y &= \text{tmp}; \\
y &= \text{av};
\end{align*}
\]
To make this safe we have to use both locks

```c
get_lock(mx);
get_lock(my);
tmp = x;
x = y;
y = tmp;
free_lock(my);
free_lock(mx);
```
Concurrent 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);
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!
Concurrency Primitives
Locks

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

Sometimes we will get
Concurrency Primitives

Locks

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

Sometimes we will get

1
get_lock(mx);

2
get_lock(my);
Concurrency Primitives

Locks

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

Sometimes we will get

1
get_lock(mx);
get_lock(my); (waits)

2
get_lock(mx); (waits)
Concurrency Primitives

Locks

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

Sometimes we will get

1
get_lock(mx);
get_lock(my);
get_lock(my); (waits)

2
get_lock(my);
get_lock(mx); (waits)

This is simple deadlock, another race condition
Concurrency Primitives

Locks

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

Locks

A very easy error to make, but often very difficult to find, particularly as the locks of \( m_x \) and \( m_y \) may be widely separated in the code, or in someone else’s code.

The use of locks requires a great deal of careful management.

Exercise. Why wouldn’t having another mutex \( m_{xy} \) to protect both \( x \) and \( y \) solve things?