Concurrency Primitives
Implementation of Locks

How are locks implemented?
Concurrence Primitives
Implementation of Locks

How are locks implemented?

They are a flag: say an integer, or even just one bit.
Concurrency Primitives
Implementation of Locks

How are locks implemented?

They are a flag: say an integer, or even just one bit.

We might use 1 to indicate locked, and 0 to indicate unlocked.
Concurrency Primitives
Implementation of Locks

```c
int lock = 0;

void get_lock()
{
    while (lock == 1) {
        deschedule();
    }
    lock = 1;
}

i.e., test the flag. If it is already 1, wait; else we can grab it by setting the flag to 1
```
Concurrency Primitives
Implementation of Locks

int lock = 0;

void get_lock()
{
    while (lock == 1) {
        deschedule();
    }
    lock = 1;
}

i.e., test the flag. If it is already 1, wait; else we can grab it by setting the flag to 1

Spot the bug!
Concurrency Primitives
Implementation of Locks

There is another update race condition
Concurrency Primitives

Implementation of Locks

There is another update race condition

1 2

1. test flag: OK
2. test flag: OK
There is another update race condition

1
  test flag: OK
  set flag

2
  test flag: OK
  set flag
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>test flag: OK</td>
<td>test flag: OK</td>
</tr>
<tr>
<td>set flag</td>
<td>set flag</td>
</tr>
</tbody>
</table>

And now both calls to `get_lock` succeed and both threads proceed to enter the critical region.
In between the testing of the flag and the setting of the flag all kinds of other things might happen
Concurrency Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen.

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be atomic.
Concurrent Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen.

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be *atomic*.

That is the test and the set are inseparable: nothing can get between them.
Concurrency Primitives
Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen.

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be *atomic*.

That is the test and the set are inseparable: nothing can get between them.

This is another kind of critical region, so we could solve it by using locks...
Fortunately we don’t have to go into an infinite regression as there are two kinds of solution: hardware and software
Fortunately we don’t have to go into an infinite regression as there are two kinds of solution: hardware and software.

Hardware designers understand mutual exclusion, so the instruction sets of all modern processors have an instruction specifically designed for this.
Fortunately we don’t have to go into an infinite regression as there are two kinds of solution: hardware and software.

Hardware designers understand mutual exclusion, so the instruction sets of all modern processors have an instruction specifically designed for this.

For example the *compare and swap* instruction.
Concurrency Primitives
Implementation of Locks

Intel has `cmpxchg` that atomically operates on a register and a byte in memory
Intel has `cmpxchg` that atomically operates on a register and a byte in memory

**CMPXCHG r/m8, r8**

Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL. This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically
In C, its action is like

```c
int compare_and_swap(int *reg, int *mem, int new)
{
    if (*reg == *mem) {
        *mem = new;
        return 1; /* got lock */
    }
    *reg = *mem;
    return 0; /* fail */
}
```

but the entire thing is done *atomically*
Using this:

```c
int flag = 0;
...
int reg = 0;
// try to set flag to 1
while (compare_and_swap(&reg, &flag, 1) == 0) {
    reg = 0; // try again
}
flag = 0;
```

This implements a busy wait
Concurrent Primitives

Implementation of Locks

Using this:

```c
int flag = 0;
...
int reg = 0;
// try to set flag to 1
while (compare_and_swap(&reg, &flag, 1) == 0) {
    reg = 0; // try again
}
flag = 0;
```

This implements a busy wait

You should spend some time going through this!
Concurrency Primitives
Implementation of Locks

Instructions found in other architectures include \texttt{test\_and\_set} and an atomic \texttt{swap}.
Concurrency Primitives
Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`.

Early architectures did not have such instructions, so software versions were devised.
Instructions found in other architectures include `test_and_set` and an atomic `swap`.

Early architectures did not have such instructions, so software versions were devised.

These include: Dekker, Dijkstra and Lamport.
Concurrency Primitives
Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`.

Early architectures did not have such instructions, so software versions were devised.

These include: Dekker, Dijkstra and Lamport.

They are very subtle as they must construct an atomic effect from non-atomic code.
Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include test_and_set and an atomic swap.

Early architectures did not have such instructions, so software versions were devised.

These include: Dekker, Dijkstra and Lamport.

They are very subtle as they must construct an atomic effect from non-atomic code.

Exercise. Go and read about these
Concurrency Primitives

Locks

Locks are widely used to address the shared update problem
Locks are widely used to address the shared update problem.

Many people say a better approach is to avoid using shared updates!
Locks are widely used to address the shared update problem.

Many people say a better approach is to avoid using shared updates!

Not always practical, but many modern programming languages are coming around to this point of view: see later.