The next approach to parallelism we shall look at is to have constructs as part of the language.
The next approach to parallelism we shall look at is to have constructs as part of the language

For example, a *monitor* is a language construct that combines mutual exclusion and synchronisation in a way that can be easier to use than the concurrency primitives

```
monitor Name
  local variable declarations
  func fun1(args) body
  func fun2(args) body
  ...
end
```

The actual syntax will vary by language
Mutual exclusion is enforced by

*only one thread at a time may be executing any function inside a given monitor*
Concurrency Control

Monitors

Mutual exclusion is enforced by

only one thread at a time may be executing any
function inside a given monitor

So, if one thread is executing $\text{fun1}$ and another thread tries to execute $\text{fun2}$, it will have to wait until the first thread exits the monitor
Concurrency Control

Monitors

So there is mutual exclusion on the local variables and within the dynamic scope of the functions in the monitor, i.e., mutual exclusion continues even if \( \text{fun1} \) calls a function defined outside the monitor.
So there is mutual exclusion on the local variables and within the dynamic scope of the functions in the monitor, i.e., mutual exclusion continues even if `fun1` calls a function defined outside the monitor.

The mutual exclusion finishes when the thread of control exits the (top level) monitor function.
Concurrency Control
Monitors

Synchronisation is provided by the use of condition variables
Synchronisation is provided by the use of condition variables

\texttt{wait}(c); \texttt{and signal}(c);
Synchronisation is provided by the use of condition variables

wait(c); and signal(c);

The associated lock is the monitor mutual exclusion lock, and is implicit
Synchronisation is provided by the use of condition variables

\texttt{wait(c);} and \texttt{signal(c);} \\

The associated lock is the monitor mutual exclusion lock, and is implicit \\

Just like the POSIX version, \texttt{wait()} will drop the monitor lock \\
to allow other threads access; and try to regain it when it resumes
We can easily implement a lock using a monitor:

```plaintext
monitor Lock
    int flag = 0;
    condition c;
    lock() { while (flag == 1) wait(c); flag = 1; }
    unlock() { flag = 0; signal(c); }
end
```
We can easily implement a lock using a monitor:

```
monitor Lock
    int flag = 0;
    condition c;
    lock() { while (flag == 1) wait(c); flag = 1; }
    unlock() { flag = 0; signal(c); }
end
```

The monitor lock provides the atomicity we need in the definition of lock.
Monitors help with management of mutual exclusion, but the usual nesting deadlock is still possible. For monitors m1 and m2:

\[
\text{monitor m1}
\text{fun1()} \{ \ldots \text{fun2()} \ldots \} \\
\text{fun2()} \{ \ldots \text{fun1()} \ldots \} \\
\ldots
\text{end}
\]

1 fun1 in monitor m1 calls fun2 in monitor m2
fun2 in monitor m2 (waits)

2 fun2 in monitor m2 calls fun1 in monitor m1 (waits)
Modularity might even encourage this error, though monitors are high enough level to be easy to analyse so there are source code tools to spot this
Concurrency Control
Monitors

Modularity might even encourage this error, though monitors are high enough level to be easy to analyse so there are source code tools to spot this

They require careful use and are not a universal solution!
Monitors clearly fit well with object oriented languages: for example, Java implements monitors on a per-object level:

```java
class foo {
    private int n = 0;
    public synchronized int inc() { n++; }
    public synchronized int dec() { n--; }
    ...
}
```

Methods with the `synchronized` keyword are within a per-object monitor
Only one of \texttt{inc} and \texttt{dec} can be executing on a given instance of \texttt{foo} at a time
Concurrency Control
Java Monitors

Only one of \texttt{inc} and \texttt{dec} can be executing on a given instance of \texttt{foo} at a time.

Condition variables: \texttt{wait()}, \texttt{notify()} and \texttt{notifyAll()}

Concurrency Control
Java Monitors

Only one of \texttt{inc} and \texttt{dec} can be executing on a given instance of \texttt{foo} at a time.

Condition variables: \texttt{wait()}, \texttt{notify()} and \texttt{notifyAll()}

Class methods (\texttt{static}) can be synchronised, too, locking the class but not its instances.
Monitors are fairly easy to use, but are somewhat large grained: the whole of each monitor, for example *all* methods marked *synchronized* in a Java object

class foo {
    private int n = 0, m = 0;
    public synchronized int incn() { n++; }
    public synchronized int decn() { n--; }
    public synchronized int incm() { m++; }
    public synchronized int decm() { m--; }
}
Concurrency Control

Monitors

To have separate locks on some of the methods requires code refactoring (or see below): You can do this, but this is driving the code towards complexity.
To have separate locks on some of the methods requires code refactoring (or see below): You can do this, but this is driving the code towards complexity.

Similarly, it is a bit fiddly to decide on what functionality goes into which monitor: if you are not careful you end up with all your code in one big monitor—sequential!
Java recognises that monitors are sometimes too large, so it allows synchronising of *statements* (rather than whole methods) as a way of providing finer gain control

```java
public class locket {
    private Object nlock = new Object();
    private int n = 0;
    public void inc() {
        synchronized(nlock) { n++; }
    }
    public void dec() {
        synchronized(nlock) { n--; }
    }
}
```
synchronized takes an arbitrary object as argument
synchronized takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor
Concurrency Control
Java Monitors

synchronized takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor

This is fine, but we have just reinvented mutexes!
synchro*ized takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor

This is fine, but we have just reinvented mutexes!

But in a more convenient form: you can’t forget to lock or unlock these!
Java also has a library of *atomic* datatypes, e.g., AtomicInteger with a few methods, that does the obvious thing
Java also has a library of *atomic* datatypes, e.g., `AtomicInteger` with a few methods, that does the obvious thing.

But these are tiresome to use as Java does not have operator overloading, like C++: thus `n.incrementAndGet()` rather than overloading `++` and using the simpler `n++`
Concurrency Control
Conditional Critical Regions

Exercise. A similar, but simpler, kind of idea is *conditional critical regions*, where a semaphore is associated with blocks of code (the critical regions)

```
let s = Semaphore::new(1);
...
region s {
    // critical region
    ...
    <set condition>
    await <some condition>
    ...
}
```

Read about this (e.g., in Ada).