CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Monitors

Programming cooperating processes in a concurrent environment is tricky

Need to recognize critical sections and correctly apply tools such as semaphores to protect them

Incorrect use of synchronization primites by even one of many cooperating processes can cause failure

Just a few simple ways programmers can make mistakes

```
wait()                signal()              wait()
// critical           // critical           // critical
// section            // section            // section
wait()                wait()
```

One way computer scientists handle tricky things is to abstract them away

One high-level synchronization construct is the **monitor** type

A **monitor** is an abstract data type (essentially, a class) that ensures operations are carried out in a way that respects mutual exclusion

Monitors have local variables that cannot be accessed except through calls to the monitors' functions

Monitor functions constitute the critical sections of code and are automatically protected by the monitor construct

Processes can call monitor functions without needing to worry about acquiring locks or waiting on semaphores

Of course, tools such as locks or semaphores are used to implement monitors, but these are abstracted away

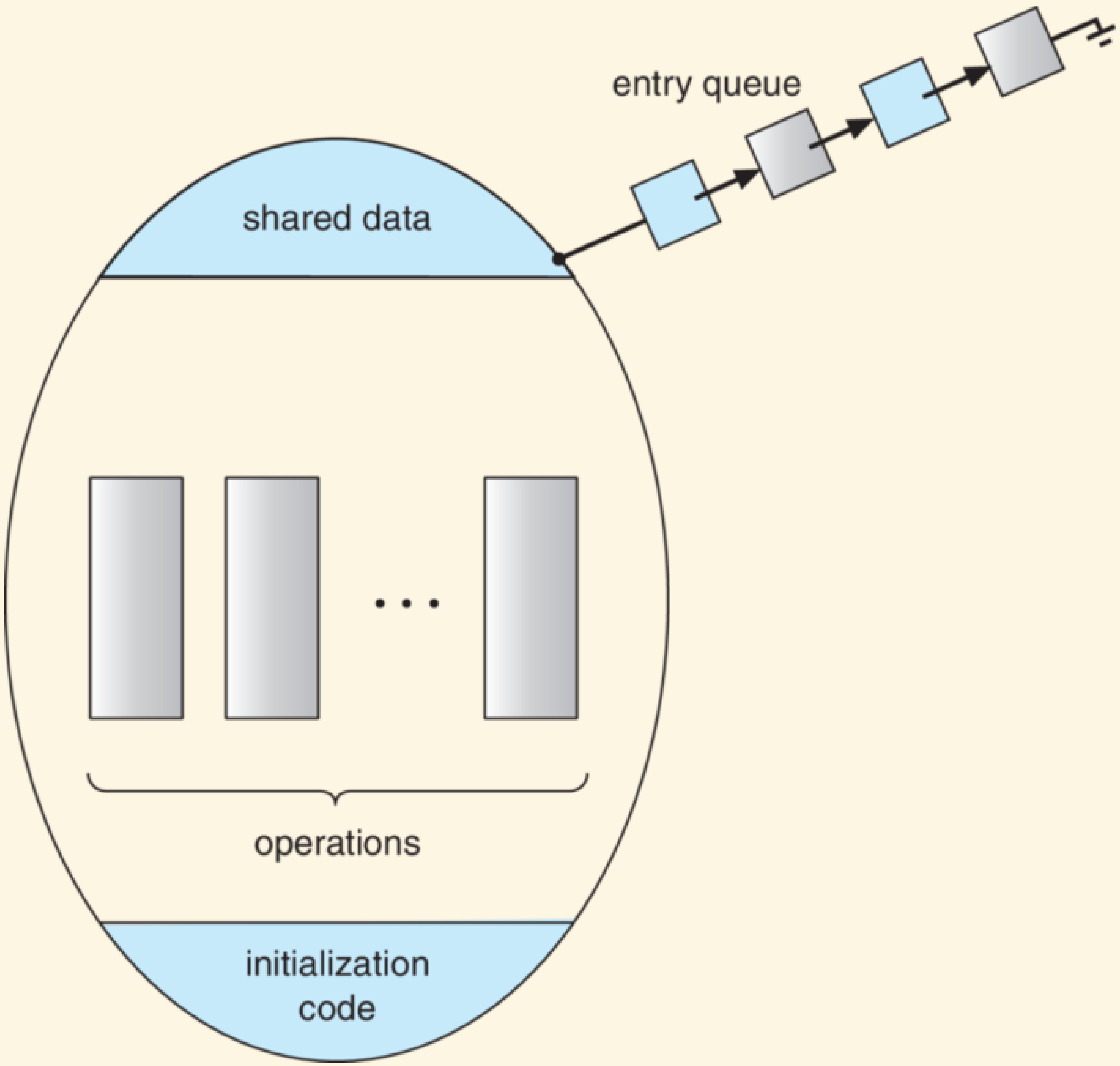
```
monitor monitor_name
{
    /* shared variable declarations */

    function P1 (...) {
    ...
    }
    function P2 (...) {
    ...
    }
    .
    .
    .
    function Pn (...) {
    ...
    }
```

As already mentioned, local variables can be accessed only through calls to monitor functions

The functions within the monitor may access only their own parameters or the local variables of the monitor

Put shared variables into the monitor and make critical section code the functions of the monitor


entry queue
shared data
operations
initialization code

Assume there is a monitor named `AccessBuffer`

```
A                      B

AccessBuffer.add()     AccessBuffer.remove()
```

We have seen how doing something like `counter++` and `counter--` will fail, but this code is almost as simple and will succeed

However, ensuring mutual exclusion alone may not be enough

Often, processes need to know what other processes are doing to be able to synchronize

For example, imagine trying to solve the producer-consumer problem using monitors as described thus far

Could be done, but would involve clumsy spinlocking

```
Producer

while (true)
    while (AccessBuffer.isFull())
        ; // wait
    AccessBuffer.add()
```

This may work, but only because we are lucky. Why is this wrong?

Monitors can't always save us from ourselves

```
Producer

while (true)
    AccessBuffer.addIfFree()
```

`addIfFree` checks the shared counter and adds if possible, otherwise does nothing and returns

Simple, but wasteful

Monitors use **condition variables** to provide synchronization

Very similar to semaphores, but there are a few key differences `condition x`

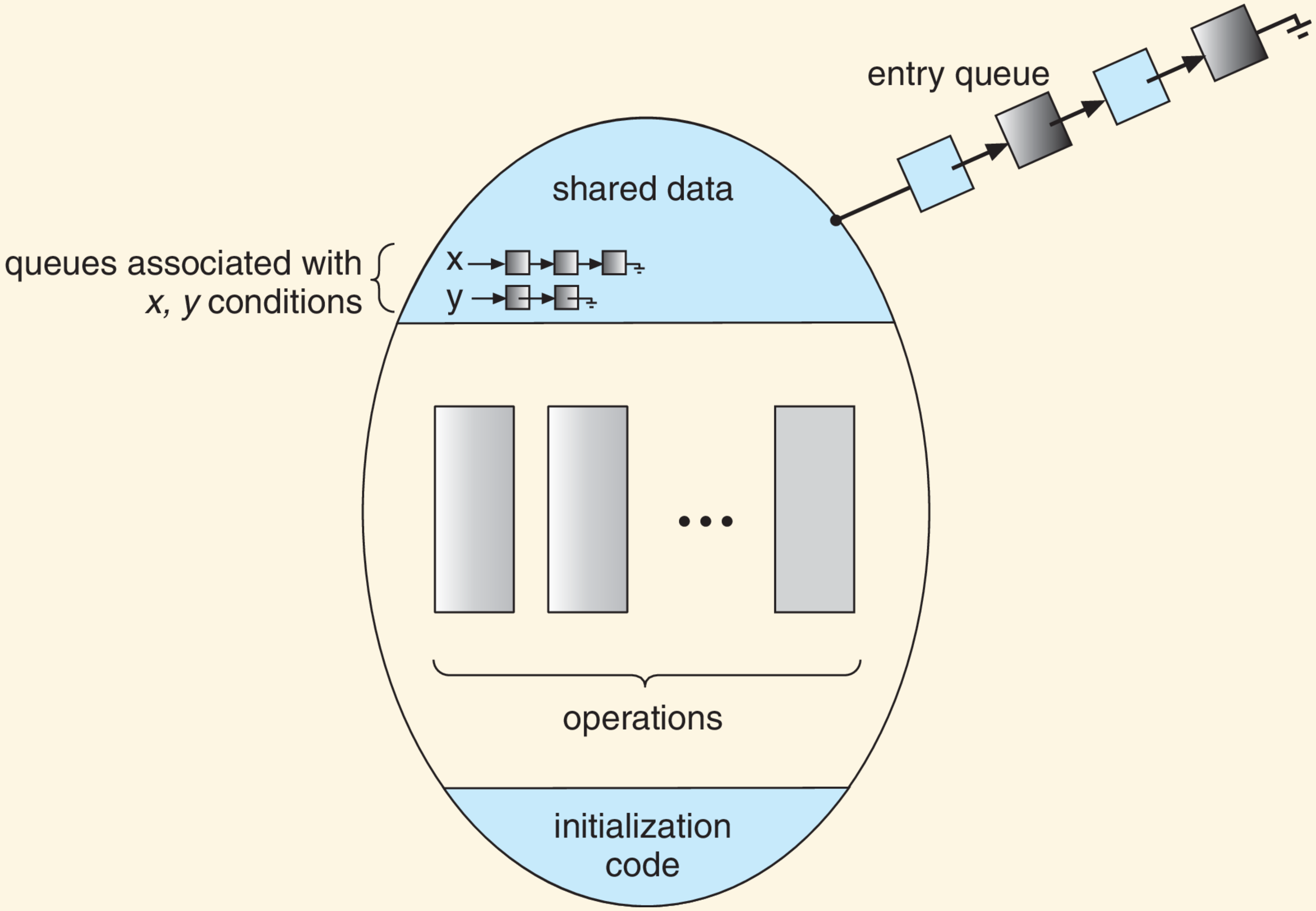
`x` can be used in just two ways: `wait()` and `signal()`

`x.wait()`: invoking process will be suspended until another process calls `x.signal()`

`x.signal()` resumes *exactly one* process that was waiting on x

If no such process exists, `x.signal()` does nothing

Reread the last line -- that is one of the major differences between semaphores and condition variables


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

```
Producer

while (true)
    AccessBuffer.addIfFree()
```

```
monitor AccessBuffer {
    // other local variables and functions missing

    condition added, removed;

    addIfFree() {
        if (counter >= max_size) {
            removed.wait();
        }
        counter++;
        add_to_queue()
        added.signal();
    }
}
```

Monitor will not completely remove complexity, but can abstract away some of the details and help with mutual exclusion in particular

One detail about `signal` with condition variables -- which process continues after the `signal`? Recall that just one process can be in the monitor at a time

Answer: it depends on the language. One reasonable solution is to force the signalling process to leave the monitor immediately after signaling

Monitors are not built into C, but condition variables are available in PThreads