Now let's figure out how to implement semaphores.

They are themselves shared data and implementing the WAIT and SIGNAL operations will require read/modify/write sequences that must be executed as critical sections.

Normally we'd use a lock semaphore to implement the mutual exclusion constraint for critical sections.

But obviously we can't use semaphores to implement semaphores!

We have what's called a bootstrapping problem: we need to implement the required functionality from scratch.

Happily, if we're running on a timeshared processor with an uninterruptible OS kernel, we can use the supervisor call (SVC) mechanism to implement the required functionality.

We can also extend the ISA to include a special test-and-set instruction that will let us implement a simple lock semaphore, which can then be used to protect critical sections that implement more complex semaphore semantics.

Single instructions are inherently atomic and, in a multi-core processor, will do what we want if the shared main memory supports reading the old value and writing a new value to a specific memory location as a single memory access.

There are other, more complex, software-only solutions that rely only on the atomicity of individual reads and writes to implement a simple lock.

For example, see "Dekker's Algorithm" on Wikipedia.

We'll look in more detail at the first two approaches.

Here are the OS handlers for the WAIT and SIGNAL supervisor calls.

Since SVCs are run kernel mode, they can't be interrupted, so the handler code is naturally executed as a critical section.

Both handlers expect the address of the semaphore location to be passed as an argument in the user's R0.

The WAIT handler checks the semaphore's value and if it's non-zero, the value is decremented and the handler resumes execution of the user's program at the instruction following the WAIT SVC.

If the semaphore is 0, the code arranges to re-execute the WAIT SVC when the user program resumes execution and then calls SLEEP to mark the process as inactive until the corresponding WAKEUP call is made.

The SIGNAL handler is simpler: it increments the semaphore value and calls WAKEUP to mark as active any processes that were WAITing for this particular semaphore.

Eventually the round-robin scheduler will select a process that was WAITing and it will be able to decrement the semaphore and proceed.

Note that the code makes no provision for fairness, i.e., there's no guarantee that a WAITing process will eventually succeed in finding the semaphore non-zero.

The scheduler has a specific order in which it runs processes, so the next-in-sequence WAITing process will always get the semaphore even if there are later-in-sequence processes that have been WAITing longer.

If fairness is desired, WAIT could maintain a queue of waiting processes and use the queue to determine which process is next in line, independent of scheduling order.

Many ISAs support an instruction like the TEST-and-CLEAR instruction shown here.

The TCLR instruction reads the current value of a memory location and then sets it to zero, all as a single operation.

It's like a LD except that it zeros the memory location after reading its value.

To implement TCLR, the memory needs to support read-and-clear operations, as well as normal reads and writes.

The assembly code at the bottom of the slide shows how to use TCLR to implement a simple lock.

The program uses TCLR to access the value of the lock semaphore.

If the returned value in RC is zero, then some other process has the lock and the program loops to try TCLR again.

If the returned value is non-zero, the lock has been acquired and execution of the critical section can proceed.

In this case, TCLR has also set the lock to zero, so that other processes will be prevented from entering the critical section.

When the critical section has finished executing, a ST instruction is used to set the semaphore to a non-zero value.