Today: Synchronization: Locks and Semaphores

What are semaphores?

Implementing locks using test and busy waiting

More on hardware support for synchronization

<table>
<thead>
<tr>
<th>Critical Section at a time.</th>
</tr>
</thead>
</table>

Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.

<table>
<thead>
<tr>
<th></th>
<th>send &amp; receive</th>
<th>monitors</th>
<th>semaphores</th>
<th>locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Hardware)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt disable</td>
<td>load/store</td>
<td>test/last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low-level atomic operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Software)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-level atomic operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The multiprocessor must support some type of cache coherence. Invalidating any copies of the value the other processes may have in their caches is a common issue. On a multiprocessor, the processor issuing the instruction must also be able to store information to memory. Atomic read-modify-write instructions atomically read a value from memory into a register and write a new value.

### Atomic Read-Modify-Write Instructions

- **compareandswap**: Read value, if value matches register value, exchange
- **exchange**: Swap values between register and memory
- **compare**: Read a value, write '1' back to memory

The implementation locks with `Test&Set`
{ 
    thread: 
        { 
            thread: 
                { 
                    thread: 
                        { 
                            thread: 
                                { 
                                    thread: 
                                        { 
                                            thread: 
                                                { 
                                                    thread: 
                                                        { 
                                                            thread: 
                                                                { 
                                                                    thread: 
                                                                        { 
                                                                            thread: 
                                                                                { 
                                                                                    thread: 
                                                                                        { 
                                                                                            thread: 
                                                                                                { 
                                                                                                    thread: 
                                                                                                        { 
                                                                                                            thread: 
                                                                                                                { 
                                                                                                                    thread: 
                                                                                                                        { 
                                                                                                                            thread: 
                                                                                                                                { 
                                                                                                                                    thread: 
                                                                                                                                            { 
                                                                                                                                                thread: 
                                                                                                                                                    { 
                                                                                                                                                        thread: 
                                                                                                                                                            { 
                                                                                                                                                                thread: 
                                                                                                                                            } 
                                                                                                                                                    } 
                                                                                                                                    } 
                                                                                                                                } 
                                                                                                                        } 
                                                                                                                } 
                                                                                                            } 
                                                                                                        } 
                                                                                                    } 
                                                                                                } 
                                                                                            } 
                                                                                        } 
                                                                                    } 
                                                                                } 
                                                                        } 
    } 
}

lock value and giving up the CPU if the lock is busy

No, but we can minimize busy-waiting time by atomically checking the

disabling interrupts?

Can we implement locks with tesseract without any busy-waiting or

Locks using Tesseract with minimal busy-waiting

How can we get the waiting thread to give up the processor, so the

What could happen to threads with different priorities?

What is the CPU doing?

What’s wrong with the above implementation?

{ 
    while (tesseract(valve) == 1) 
        { 
            while (tesseract(valve) == 1) 
                { 
                    while (tesseract(valve) == 1) 
                        { 
                            while (tesseract(valve) == 1) 
                                { 
                                    while (tesseract(valve) == 1) 
                                        { 
                                            while (tesseract(valve) == 1) 
                                                { 
                                                    while (tesseract(valve) == 1) 
                                                        { 
                                                            while (tesseract(valve) == 1) 
                                                                { 
                                                                    while (tesseract(valve) == 1) 
                                                                        { 
                                                                            while (tesseract(valve) == 1) 
                                                                                { 
                                                                                    while (tesseract(valve) == 1) 
                                                                                        { 
                                                                                            while (tesseract(valve) == 1) 
                                                                                                { 
                                                                                                    while (tesseract(valve) == 1) 
                                                                                                        { 
                                                                                                            while (tesseract(valve) == 1) 
                                                                                                                { 
                                                                                                                    while (tesseract(valve) == 1) 
                                                                                                                        { 
                                                                                                                            while (tesseract(valve) == 1) 
                                                                                                                                { 
                                                                                                                                    while (tesseract(valve) == 1) 
                                                                                                                                            { 
                                                                                                                                                while (tesseract(valve) == 1) 
                                                                                                                                                    { 
                                                                                                                                                        while (tesseract(valve) == 1) 
                                                                                                                                                            { 
                                                                                                                                                                while (tesseract(valve) == 1) 
                                                                                                                                            } 
                                                                                                                                                    } 
                                                                                                                                    } 
                                                                                                                                } 
                                                                                                                        } 
                                                                                                                } 
                                                                                                            } 
                                                                                                        } 
                                                                                                    } 
                                                                                                } 
                                                                                            } 
                                                                                        } 
                                                                                    } 
                                                                                } 
                                                                        } 
    } 

Busy Waiting
A process can acquire access so long as at least one unit of the resource is available.

- The initial count to which the semaphore is initialized is usually the number of resources.
- It is initialized to zero (value = 0)
- Can vary from 0 to \( N \) (section at a time).
- Guarantees mutually exclusive access to a resource (only one process is in the critical section at a time).
- Special atomic instructions.
- Semaphore: an integer variable that can be updated only using two

**Semaphores: Key Concepts**

4. \( S > 0 \) signal(): unblocks one process on semaphore S's wait queue.

Semaphore S

4. \( S > 0 \) wait(): the OS puts the process on the wait queue for

- if semaphore S is not free, the OS puts the process on the wait queue for

Each semaphore supports a queue of processes that are waiting to access the critical section (e.g. to buy milk).

\[ \text{Semaphore} \rightarrow \text{Wait()} \text{ and } \text{Semaphore} \rightarrow \text{Signal()} \]

\[ \text{Like locks, a semaphore supports two atomic operations,} \]

**Semaphores**
Implementing `Signal` and `Wait`:

```java
class Semaphore {
    volatile int value;
    public Semaphore() { value = 0; }
    public Semaphore(int init_value) { value = init_value; }
    public void signal() { value++; }
    public void wait() throws InterruptedException {
        while (value == 0) Thread.sleep(1);
        value--;
    }
}
```

Binary Semaphore Example:

Thread A

Too Much Milk using semaphores:

```java
Semaphore lock, milk;
lock.acquire();
lock.acquire();
lock.release();
lock.release();
```

Thread B

Too Much Milk using locks:

```java
lock.acquire();
lock.acquire();
lock.release();
lock.release();
```
Signal and Wait: Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Execute</th>
<th>Empty</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiple Consumers and Producers

Semaphore and Semaphores

Semaphore is a linear object that provides a simple synchronization mechanism used to control access to resources by multiple threads.

Examples:
- Example: You can implement thread pool (for the Unix system call waitpid(PID))
- The initial value of the semaphore is usually 0 in this case.

Synchronization Constraints: used to express general scheduling constraints

- Critical section
  - S-Wait() S-Enter() Thread:Join
  - S-Wait() S-Enter() Semaphore initialization
  - Semaphore S:

- S-Wait() S-Enter() Thread:Join

Using Semaphores
Semaphores can be used for three purposes:

- To cause one thread to wait for a specific action to be signaled from another thread.
- To control access to a shared pool of resources (using a counting semaphore).
- To ensure mutually exclusive execution of a critical section (as locks do).

Semaphores are a generalization of locks.

Locks can be implemented by disabling interrupts or busy waiting.

### Summary

Multile Consumers and Producers Problem