Today: Synchronization

• Synchronization
  – Mutual exclusion
  – Critical sections

• Example: Too Much Milk

• Locks

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

Computing Parable

• The Lost Key

* courtesy S. Keshav
Recap: Synchronization

• What kind of knowledge and mechanisms do we need to get independent processes to communicate and get a consistent view of the world (computer state)?

• Example: Too Much Milk

<table>
<thead>
<tr>
<th>Time</th>
<th>You</th>
<th>Your roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Arrive home</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Look in fridge, no milk</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Leave for grocery</td>
<td></td>
</tr>
<tr>
<td>3:15</td>
<td></td>
<td>Arrive home</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive at grocery</td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3:35</td>
<td>Arrive home, put milk in fridge</td>
<td></td>
</tr>
<tr>
<td>3:45</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td>Arrive home, put up milk</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td>Oh no!</td>
</tr>
</tbody>
</table>

Recap: Synchronization Terminology

• **Synchronization**: use of atomic operations to ensure cooperation between threads

• **Mutual Exclusion**: ensure that only one thread does a particular activity at a time and excludes other threads from doing it at that time

• **Critical Section**: piece of code that only one thread can execute at a time

• **Lock**: mechanism to prevent another process from doing something
  – Lock before entering a critical section, or before accessing shared data.
  – Unlock when leaving a critical section or when access to shared data is complete
  – Wait if locked

=> All synchronization involves waiting.
Too Much Milk: Solution 1

- What are the correctness properties for this problem?
  - Only one person buys milk at a time.
  - Someone buys milk if you need it.
- Restrict ourselves to atomic loads and stores as building blocks.
  - Leave a note (a version of lock)
  - Remove note (a version of unlock)
  - Do not buy any milk if there is note (wait)

Thread A

```java
if (noMilk & NoNote) {
    leave Note;
    buy milk;
    remove note;
}
```

Thread B

```java
if (noMilk & NoNote) {
    leave Note;
    buy milk;
    remove note;
}
```

Does this work?

Too Much Milk: Solution 2

How about using labeled notes so we can leave a note before checking the milk?

Thread A

```java
leave note A
if (noNote B) {
    if (noMilk) {
        buy milk;
    }
}
remove note;
```

Thread B

```java
leave note B
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note;
```

Does this work?
Too Much Milk: Solution 3

Thread A

leave note A
X: while (Note B) {
    do nothing;
}
if (noMilk){
    buy milk;
}
remove note A;

Thread B

leave note B
Y: if (noNote A) {
    if (noMilk){
        buy milk;
    }
}
remove note B;

Does this work?

Correctness of Solution 3

• At point Y, either there is a note A or not.
  1. If there is no note A, it is safe for thread B to check and buy milk, if needed.
     (Thread A has not started yet).
  2. If there is a note A, then thread A is checking and buying milk as needed or is
     waiting for B to quit, so B quits by removing note B.

• At point X, either there is a note B or not.
  1. If there is not a note B, it is safe for A to buy since B has either not started or quit.
  2. If there is a note B, A waits until there is no longer a note B, and either finds milk
     that B bought or buys it if needed.

• Thus, thread B buys milk (which thread A finds) or not, but either way it
  removes note B. Since thread A loops, it waits for B to buy milk or not, and
  then if B did not buy, it buys the milk.
Is Solution 3 a good solution?

- It is too complicated - it was hard to convince ourselves this solution works.

- It is asymmetrical - thread A and B are different. Thus, adding more threads would require different code for each new thread and modifications to existing threads.

- A is busy waiting - A is consuming CPU resources despite the fact that it is not doing any useful work.

=> This solution relies on loads and stores being atomic.

Language Support for Synchronization

Have your programming language provide atomic routines for synchronization.

- **Locks:** one process holds a lock at a time, does its critical section releases lock.

- **Semaphores:** more general version of locks.

- **Monitors:** connects shared data to synchronization primitives.

=> All of these require some hardware support, and waiting.
Locks

- Locks: provide mutual exclusion to shared data with two “atomic” routines:
  - Lock.Acquire - wait until lock is free, then grab it.
  - Lock.Release - unlock, and wake up any thread waiting in Acquire.

Rules for using a lock:

- Always acquire the lock before accessing shared data.
- Always release the lock after finishing with shared data.
- Lock is initially free.

Implementing Too Much Milk with Locks

Too Much Milk

<table>
<thead>
<tr>
<th></th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lock.Acquire();</td>
<td>Lock.Acquire();</td>
</tr>
<tr>
<td></td>
<td>if (noMilk){</td>
<td>if (noMilk){</td>
</tr>
<tr>
<td></td>
<td>buy milk;</td>
<td>buy milk;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>Lock.Release();</td>
<td>Lock.Release();</td>
</tr>
</tbody>
</table>

- This solution is clean and symmetric.
- How do we make Lock.Acquire and Lock.Release atomic?
Hardware Support for Synchronization

- Implementing high level primitives requires low-level hardware support
- What we have and what we want

<table>
<thead>
<tr>
<th>Concurrent programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-level atomic operations (hardware)</td>
</tr>
<tr>
<td>High-level atomic operations (software)</td>
</tr>
</tbody>
</table>

Implementing Locks By Disabling Interrupts

- There are two ways the CPU scheduler gets control:
  - **Internal Events:** the thread does something to relinquish control (e.g., I/O).
  - **External Events:** interrupts (e.g., time slice) cause the scheduler to take control away from the running thread.
- On uniprocessors, we can prevent the scheduler from getting control as follows:
  - **Internal Events:** prevent these by not requesting any I/O operations during a critical section.
  - **External Events:** prevent these by disabling interrupts (i.e., tell the hardware to delay handling any external events until after the thread is finished with the critical section)
Implementing Locks by Disabling Interrupts

- For uniprocessors, we can disable interrupts for high-level primitives like locks, whose implementations are private to the kernel.
- The kernel ensures that interrupts are not disabled forever, just like it already does during interrupt handling.

```cpp
class Lock {
public:
    void Acquire();
    void Release();
private:
    int value;
    Queue Q;
}
Lock::Lock {
    // lock is free
    value = 0;
    // queue is empty
    Q = 0;
}
Lock::Acquire(Thread T) {
    // syscall: kernel execs this
disable interrupts;
    if (value == BUSY) {
        add T to Q
        put T to Sleep;
    } else {
        value = BUSY;
    }
    enable interrupts; }
Lock::Release() {
    disable interrupts; 
    if queue not empty {
        take thread T off Q
        put T on ready queue
    } else {
        value = FREE
    }
    enable interrupts; }
```

Atomic read-modify-write Instructions

- Atomic read-modify-write instructions *atomically* read a value from memory into a register and write a new value.
  - Straightforward to implement simply by adding a new instruction on a uniprocessor.
  - On a multiprocessor, the processor issuing the instruction must also be able to *invalidate* any copies of the value the other processes may have in their cache, i.e., the multiprocessor must support some type of *cache coherence*.

- **Examples:**
  - **Test&Set:** (most architectures) read a value, write ‘1’ back to memory.
  - **Exchange:** (x86) swaps value between register and memory.
  - **Compare&Swap:** (68000) read value, if value matches register value r1, exchange register r2 and value.
Implementing Locks with Test&Set

- **Test&Set:** reads a value, writes ‘1’ to memory, and returns the old value.

```c++
class Lock {
public:
    void Acquire();
    void Release();
private:
    int value;
};
Lock() { value = 0; }

void Acquire();
while (test&set(value) == 1);
}

void Release();
value = 0;
}
```

- If lock is free (value = 0), test&set reads 0, sets value to 1, and returns 0. The Lock is now busy: the test in the while fails, and Acquire is complete.
- If lock is busy (value = 1), test&set reads 1, sets value to 1, and returns 1. The while continues to loop until a Release executes.

---

Busy Waiting

```c++
Acquire()
//if Busy, do nothing
while (test&set(value) == 1);
}
```

- What's wrong with the above implementation?
  - What is the CPU doing?
  - What could happen to threads with different priorities?
- How can we get the waiting thread to give up the processor, so the releasing thread can execute?
Locks using Test&Set with minimal busy-waiting

• Can we implement locks with test&set without any busy-waiting or disabling interrupts?
• No, but we can minimize busy-waiting time by atomically checking the lock value and giving up the CPU if the lock is busy

```java
class Lock {
    // same declarations as earlier
    private int guard;
}

Acquire(T:Thread) {
    while (test&set(guard) == 1) ;
    if (value != FREE) {
        put T on Q;
        T.Sleep() & set guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}

Release() {
    // busy wait
    while (test&set(guard) == 1) ;
    if Q is not empty {
        take T off Q;
        put T on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

Summary

• Communication among threads is typically done through shared variables.

• Critical sections identify pieces of code that cannot be executed in parallel by multiple threads, typically code that accesses and/or modifies the values of shared variables.

• Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.
  – Achieving synchronization directly with loads and stores is tricky and error-prone
  – Solution: use high-level primitives such as locks, semaphores, monitors