Mutual Exclusion:
Primitives and
Implementation Considerations

Too Much Milk: Lessons

- Last solution works, but it is really unsatisfactory
  - Solution is complicated; proving correctness is tricky even for the simple example
  - While thread is waiting, it is consuming CPU time
- How can we do better?
  - Define higher-level programming abstractions to simplify concurrent programming
  - Use hardware features to eliminate busy waiting
  - Stay tuned…

Introducing Locks

- Locks – a higher-level programming abstraction
  - Two methods
    - Lock::Acquire() – wait until lock is free, then grab it
    - Lock::Release() – release the lock, waking up a waiter, if any
- With locks, too much milk problem is very easy!

```
Lock→Acquire();
if (noMilk) {
        buy milk;
}
Lock→Release();
```

How can we implement locks?

Implementing Locks

- Generally requires some level of hardware support
- Two common implementation approaches
  - Disable interrupts
    - Uni-processor architectures only
  - Atomic read-modify-write instructions
    - Uni- and multi-processor architectures
- Other implementation alternatives
  - Busy-waiting implementation

Disabling Interrupts

- Key observations:
  - On a uni-processor, an operation is atomic if no context-switch is allowed in the middle of the operation
  - Context switch occurs because of:
    - Internal events: system calls and exceptions
    - External events: interrupts
  - Mutual exclusion can be achieved by preventing context switch
- Prevention of context switch
  - Eliminate internal events: easy (under program control)
  - Eliminate external events: disable interrupts
    - Hardware delays the processing of interrupts until interrupts are enabled

Lock Implementation: A Naïve Solution

```
Lock::Acquire() { disable interrupts; }
Lock::Release() { enable interrupts;}
```

- Will this work on a uni-processor?
- What is wrong with this solution?
  - Once interrupts are disabled, the thread can't be stopped ➔ Can starve other threads
  - Critical sections can be arbitrarily long ➔ Can't bound the amount of time needed to respond to interrupts

A Better Solution

```
Class Lock{
    int value = FREE;
}
```

```
Lock::Acquire() {
    Disable interrupts;
    while (value != FREE) {
            Enable interrupts;
            Disable interrupts;
    }
    value = BUSY;
    Enable interrupts;
}
```

```
Lock::Release() {
    Disable interrupts;
    value = FREE;
    Enable interrupts;
}
```

Why do we need to enable interrupt inside the while loop in Acquire?

Atomic Read-Modify-Write (ARMW)

- Disabling interrupts works only on uni-processors
- For uni- and multi-processor architectures: implement locks using atomic read-modify-write instructions
  - Atomically
    1. read a memory location into a register, and
    2. write a new value to the location
  - Implementing ARMW is tricky in multi-processors
    - Requires hardware support on memory bus
- Examples:
  - Test&set instructions (most architectures)
    - Reads a value from memory
    - Write "1" back to memory location
  - Compare & swap (68000), exchange (x86), …
    - Test the value against some constant
    - If the test returns true, set value in memory to different value
    - Report the result of the test in a flag

Implementing Locks with Test&set

```
Class Lock{
   int value = 0;
}
```

```
Lock::Acquire() {
while (test&set(value) == 1);
}
```

- If lock is free, then test&set reads 0 and sets value to 1 ➔ lock is set to busy and Acquire completes
- If lock is busy, the test&set reads 1 and sets value to 1 ➔ no change in lock's status and Acquire loops

```
Lock::Release() {
   value = 0;
}
```


Locks and Busy Waiting

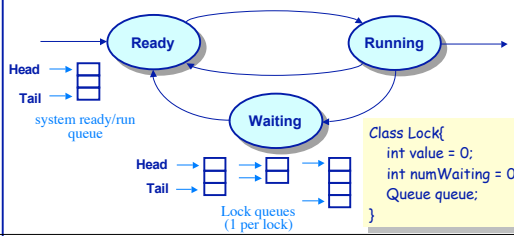
```
Lock::Acquire() {
while (test&set(value) == 1);
}
```

- Busy-waiting:
  - Threads consume CPU cycles while waiting
- Limitations
  - Inefficient
  - What happens if threads have different priorities?
    - Busy-waiting thread remains runnable
    - If the thread waiting for a lock has higher priority than the thread occupying the lock, then ???

Can we do better?

Implementing Locks without Busy Waiting
Using operating system kernel

- Eliminate busy-waiting through the use of multi-programming
- When a thread needs to block inside Acquire()
  - Suspend the currently executing thread
  - Dispatch a ready thread




```
Class Lock{
   int value = 0;
   int numWaiting = 0;
   Queue queue;
}
```


Implementing Locks without Busy Waiting
Using Disable Interrupts

```
Lock::Acquire() {
   Disable interrupts;
   while (value ≠ FREE) {
         Enable interrupts;
         Disable interrupts;
   }
   value := BUSY;
   Enable interrupts;
}
```

With busy-waiting

```
Lock::Acquire() {
   Disable interrupts;
   if (value ≠ FREE) {
      Put TCB on wait queue for lock;
      Switch();  // dispatch a ready thread
      }
   else { value := BUSY; }
   Enable interrupts;
}
```

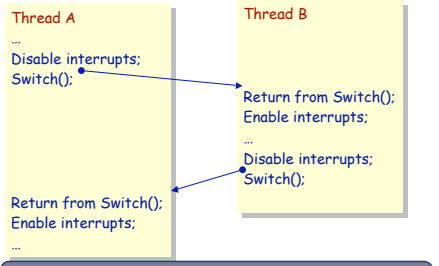
Without busy-waiting

```
Lock::Release() {
   Disable interrupts;
   value := FREE;
   Enable interrupts;
}
```

```
Lock::Release() {
   Disable interrupts;
   if wait queue is not empty {
      Move a waiting thread to ready queue; }
   else {value := FREE; }
   Enable interrupts;
}
```

Interrupts and Switch(): An Aside

- Context switch operation is sandwiched between disable and enable interrupt operations



Invariant: Interrupts are turned off before calling Switch()
And turned back on when Switch() returns

Implementing Locks without Busy Waiting

Using Test&Set

```
Lock::Acquire() {
while (test&set(value) == 1);
}
```

With busy-waiting

```
Lock::Acquire() {
if (test&set(value) == 1) {
   Put TCB on wait queue for lock;
   Switch();  // dispatch a ready thread
   }
}
```

Without busy-waiting

```
Lock::Release() {
   value := 0;
}
```

```
Lock::Release() {
if (wait queue is not empty) {
   Move a waiting thread to ready queue;
   }
   else { value := 0; }
}
```

Does this work?