Too Much Milk

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery store	
3:15		Arrive home
3:20	Arrive at grocery store	Look in fridge, no milk
3:25	Buy milk	Leave for grocery store
3:35	Arrive home, put milk in fridge	
3:45		Buy milk
3:50		Arrive home with milk
3:50		Too much milk!

```
if (noMilk) {
  buy milk;
}
```

```
if (noMilk) {
  buy milk;
}
```


Too Much Milk: Solution 1?

Thread A

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

Thread B

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

Too Much Milk: Solution 2?

Thread A

```
1   leave note A;
2   if (noNote B) {
3      if (noMilk) {
4         buy milk;
5      }
6   }
7   remove note A;
```

Thread B

```
1   leave note B;
2   if (noNote A) {
3      if (noMilk) {
4         buy milk;
5      }
6   }
7   remove note B;
```


Too Much Milk: Solution 3?

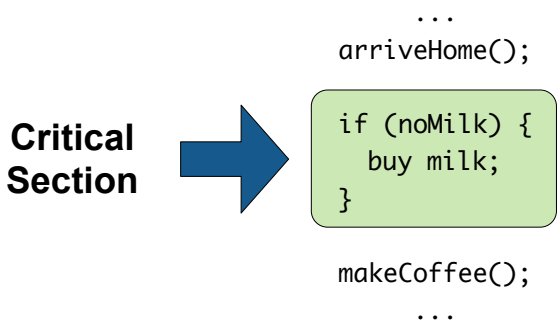
Thread A

```
1   leave note A;
2   while (note B) {
3       do nothing;
4   }
5   if (noMilk) {
6       buy milk;
7   }
8   remove note A;
```

Thread B

```
1   leave note B;
2   if (noNote A) {
3       if (noMilk) {
4           buy milk;
5       }
6   }
7   remove note B;
```

Critical Sections



Requirement of **mutual exclusion**

Locks

Too Much Milk with Locks

Thread A

```
1   lock.acquire();
2   if (noMilk) {
3       buy milk;
4   }
5   lock.release();
```

Thread B

```
1    lock.acquire();
2    if (noMilk) {
3        buy milk;
4    }
5    lock.release();
```


Implementing Locks: Disabling Interrupts

```
class Lock {
  public:
    void acquire();
    void release();
  private:
    int value = FREE;
    Queue Q = empty;
}
```

```
Lock::acquire() {
  disable interrupts;
  if (value == BUSY) {
     add curThread to Q;
     put curThread 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;
}
```

Implementing Locks: Atomic Test&Set

```
class Lock {
  public:
    void acquire();
    void release();
  private:
    int value = FREE; // FREE = 0, BUSY = 1
}
```

```
Lock::acquire() {
  while (test&set(value) == BUSY) {
     // do nothing
   }
}
```

```
Lock::release() {
  value = FREE
}
```


Minimizing Busy-Waiting

```
class Lock {
  public:
    void acquire();
    void release();
  private:
    int value = FREE; // FREE = 0, BUSY = 1
    int guard = 0;
    Queue Q = empty;
}
```

```
Lock::acquire() {
  while (test&set(guard) == 1) {
     // do nothing
   }
   if (value == BUSY) {
     put curThread on Q;
     put curThread to sleep & guard = 0;
   } else {
     value = BUSY;
     guard = 0;
   }
}
```

```
Lock::release() {
  while (test&set(guard) == 1) {
     // do nothing
   }
   if Q is not empty {
     take T off Q;
     put T on ready queue;
   } else {
     value = FREE;
   }
   guard = 0;
}
```