

<p><i>Thread Synchronization: Too Much Milk</i></p>
---

<p>Implementing Critical Sections in Software Hard</p>
<ul style="list-style-type: none"> <li>◆ The following example will demonstrate the difficulty of providing mutual exclusion with memory reads and writes <ul style="list-style-type: none"> <li>➢ Hardware support is needed</li> </ul> </li> <li>◆ The code must work <i>all</i> of the time <ul style="list-style-type: none"> <li>➢ Most concurrency bugs generate correct results for <i>some</i> interleavings</li> </ul> </li> <li>◆ Designing mutual exclusion in software shows you how to think about concurrent updates <ul style="list-style-type: none"> <li>➢ Always look for what you are checking and what you are updating</li> <li>➢ A meddlesome thread can execute between the check and the update, the dreaded race condition</li> </ul> </li> </ul>

<p><i>Thread Coordination</i></p>		
<p>Too much milk!</p> <table style="width: 100%; border: none;"> <tr> <td style="width: 50%; vertical-align: top;"> <p>Jack</p> <ul style="list-style-type: none"> <li>◆ Look in the fridge; out of milk</li> <li>◆ Go to store</li> <li>◆ Buy milk</li> <li>◆ Arrive home; put milk away</li> </ul> </td> <td style="width: 50%; vertical-align: top;"> <p>Jill</p> <ul style="list-style-type: none"> <li>◆ Look in fridge; out of milk</li> <li>◆ Go to store</li> <li>◆ Buy milk</li> <li>◆ Arrive home; put milk away</li> <li>◆ Oh, no!</li> </ul> </td> </tr> </table> <div style="border: 1px solid black; padding: 2px; margin-top: 10px; text-align: center;"> <p>Fridge and milk are shared data structures</p> </div>	<p>Jack</p> <ul style="list-style-type: none"> <li>◆ Look in the fridge; out of milk</li> <li>◆ Go to store</li> <li>◆ Buy milk</li> <li>◆ Arrive home; put milk away</li> </ul>	<p>Jill</p> <ul style="list-style-type: none"> <li>◆ Look in fridge; out of milk</li> <li>◆ Go to store</li> <li>◆ Buy milk</li> <li>◆ Arrive home; put milk away</li> <li>◆ Oh, no!</li> </ul>
<p>Jack</p> <ul style="list-style-type: none"> <li>◆ Look in the fridge; out of milk</li> <li>◆ Go to store</li> <li>◆ Buy milk</li> <li>◆ Arrive home; put milk away</li> </ul>	<p>Jill</p> <ul style="list-style-type: none"> <li>◆ Look in fridge; out of milk</li> <li>◆ Go to store</li> <li>◆ Buy milk</li> <li>◆ Arrive home; put milk away</li> <li>◆ Oh, no!</li> </ul>	

<p>Formalizing “Too Much Milk”</p>
<ul style="list-style-type: none"> <li>◆ Shared variables <ul style="list-style-type: none"> <li>➢ “Look in the fridge for milk” – check a variable</li> <li>➢ “Put milk away” – update a variable</li> </ul> </li> <li>◆ Safety property <ul style="list-style-type: none"> <li>➢ At most one person buys milk</li> </ul> </li> <li>◆ Liveness <ul style="list-style-type: none"> <li>➢ Someone buys milk when needed</li> </ul> </li> <li>◆ How can we solve this problem?</li> </ul>

<p>How to think about synchronization code</p>
<ul style="list-style-type: none"> <li>◆ Every thread has the same pattern <ul style="list-style-type: none"> <li>➢ Entry section: code to attempt entry to critical section</li> <li>➢ Critical section: code that requires isolation (e.g., with mutual exclusion)</li> <li>➢ Exit section: cleanup code after execution of critical region</li> <li>➢ Non-critical section: everything else</li> </ul> </li> <li>◆ There can be multiple critical regions in a program <ul style="list-style-type: none"> <li>➢ Only critical regions that access the same resource (e.g., data structure) need to synchronize with each other</li> </ul> </li> </ul> <pre style="margin-top: 10px;"> while(1) {     Entry section     Critical section     Exit section     Non-critical section } </pre>

<p>The correctness conditions</p>
<ul style="list-style-type: none"> <li>◆ Safety <ul style="list-style-type: none"> <li>➢ Only one thread in the critical region</li> </ul> </li> <li>◆ Liveness <ul style="list-style-type: none"> <li>➢ Some thread that enters the entry section eventually enters the critical region</li> <li>➢ Even if some thread takes forever in non-critical region</li> </ul> </li> <li>◆ Bounded waiting <ul style="list-style-type: none"> <li>➢ A thread that enters the entry section enters the critical section within some bounded number of operations.</li> </ul> </li> <li>◆ Failure atomicity <ul style="list-style-type: none"> <li>➢ It is OK for a thread to die in the critical region</li> <li>➢ Many techniques do not provide failure atomicity</li> </ul> </li> </ul> <pre style="margin-top: 10px;"> while(1) {     Entry section     Critical section     Exit section     Non-critical section } </pre>

### Too Much Milk: Solution #0

```

while(1) {
  if (noMilk) { // check milk (Entry section)
    if (noNote) { // check if roommate is getting milk
      leave Note; //Critical section
      buy milk;
      remove Note; // Exit section
    } // Non-critical region
  }
}

```

- Is this solution
  - 1. Correct
  - 2. Not safe
  - 3. Not live
  - 4. No bounded wait
  - 5. Not safe and not live
- It works sometime and doesn't some other times
  - Threads can be context switched between checking and leaving note
  - Live, note left will be removed
  - Bounded wait ('buy milk' takes a finite number of steps)

What if we switch the order of checks?

### Too Much Milk: Solution #1

turn := Jill // Initialization

```

while(1) {
  while(turn ≠ Jack) : //spin
  while (Milk) : //spin
  buy milk; // Critical section
  turn := Jill // Exit section
} // Non-critical section

```

```

while(1) {
  while(turn ≠ Jill) : //spin
  while (Milk) : //spin
  buy milk;
  turn := Jack
} // Non-critical section

```

- Is this solution
  - 1. Correct
  - 2. Not safe
  - 3. Not live
  - 4. No bounded wait
  - 5. Not safe and not live
- At least it is safe

### Solution #2 (a.k.a. Peterson's algorithm): combine ideas of 0 and 1

Variables:

- $in_i$ : thread  $T_i$  is executing, or attempting to execute, in CS
- $turn$ : id of thread allowed to enter CS if multiple want to

Claim: We can achieve mutual exclusion if the following invariant holds before entering the critical section:

$$((\neg in_j \vee (in_i \wedge turn = i)) \wedge in_i) \wedge CS$$

.....

$in_i = false$

$$((\neg in_0 \vee (in_0 \wedge turn = 1)) \wedge in_1) \wedge ((\neg in_1 \vee (in_1 \wedge turn = 0)) \wedge in_0) \Rightarrow ((turn = 0) \wedge (turn = 1)) = false$$

### Peterson's Algorithm

$in_0 = in_1 = false;$

```

Jack
while (1) {
  in_0 := true;
  turn := Jill;
  while (turn == Jill
    && in_1) : //wait
  Critical section
  in_0 := false;
  Non-critical section
}

```

```

Jill
while (1) {
  in_1 := true;
  turn := Jack;
  while (turn == Jack
    && in_0) : //wait
  Critical section
  in_1 := false;
  Non-critical section
}

```

Spin!

$turn = Jack, in_0 = false, in_1 := true$

Safe, live, and bounded waiting  
But, only 2 participants

### Too Much Milk: Lessons

- Peterson's works, but it is really unsatisfactory
  - Limited to two threads
  - 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?
  - Use hardware to make synchronization faster
  - Define higher-level programming abstractions to simplify concurrent programming

### Towards a solution

The problem boils down to establishing the following right after entry<sub>i</sub>

$$(\neg in_j \vee (in_i \wedge turn = i)) \wedge in_i = (\neg in_j \vee turn = i) \wedge in_i$$

Or, intuitively, right after Jack enters:

- Jack has signaled that he is in the entry section ( $in_i$ )
- And -
  - Jill isn't in the critical section or entry section ( $\neg in_j$ )
  - Or -
    - Jill is also in the entry section but it is Jack's turn ( $in_j \wedge turn = i$ )

How can we do that?

```

entry_i = in_i := true;
while (in_j ∧ turn ≠ i);

```

## We hit a snag

```

Thread T0
while (!terminate) {
  in0 := true;
  while (in1 ∧ turn ≠ 0):
    {in0 ∧ (¬ in1 ∨ turn = 0)}
  CS0
  .....
}

Thread T1
while (!terminate) {
  in1 := true;
  {in1}
  while (in0 ∧ turn ≠ 1):
    {in1 ∧ (¬ in0 ∨ turn = 1)}
  CS1
  .....
}

```

The assignment to  $in_0$  invalidates the invariant!

13

## What can we do?

Add assignment to *turn* to establish the second disjunct

```

Thread T0
while (!terminate) {
  in0 := true;
  α0 turn := 1;
  {in0}
  while (in1 ∧ turn ≠ 0):
    {in0 ∧ (¬ in1 ∨ turn = 0 ∨ at(α0))}
  CS0
  in0 := false;
  NCS0
}

Thread T1
while (!terminate) {
  in1 := true;
  α1 turn := 0;
  {in1}
  while (in0 ∧ turn ≠ 1):
    {in1 ∧ (¬ in0 ∨ turn = 1 ∨ at(α0))}
  CS1
  in1 := false;
  NCS1
}

```

14

## Safe?

```

Thread T0
while (!terminate) {
  in0 := true;
  α0 turn := 1;
  {in0}
  while (in1 ∧ turn ≠ 0):
    {in0 ∧ (¬ in1 ∨ turn = 0 ∨ at(α0))}
  CS0
  in0 := false;
  NCS0
}

Thread T1
while (!terminate) {
  in1 := true;
  α1 turn := 0;
  {in1}
  while (in0 ∧ turn ≠ 1):
    {in1 ∧ (¬ in0 ∨ turn = 1 ∨ at(α0))}
  CS1
  in1 := false;
  NCS1
}

```

If both in CS, then

$$in_0 \wedge (\neg in_1 \vee at(\alpha_1) \vee turn = 0) \wedge in_1 \wedge (\neg in_0 \vee at(\alpha_0) \vee turn = 1) \wedge \neg at(\alpha_0) \wedge \neg at(\alpha_1) = (turn = 0) \wedge (turn = 1) = false$$

15

## Live?

```

Thread T0
while (!terminate) {
  {S1: ¬in0 ∧ (turn = 1 ∨ turn = 0)}
  in0 := true;
  {S2: in0 ∧ (turn = 1 ∨ turn = 0)}
  α0 turn := 1;
  {S3}
  while (in1 ∧ turn ≠ 0):
    {S3: in0 ∧ (¬ in1 ∨ at(α1) ∨ turn = 0)}
  CS0
  {S4}
  in0 := false;
  {S5}
  NCS0
}

Thread T1
while (!terminate) {
  {R1: ¬in0 ∧ (turn = 1 ∨ turn = 0)}
  in1 := true;
  {R2: in0 ∧ (turn = 1 ∨ turn = 0)}
  α1 turn := 0;
  {R3}
  while (in0 ∧ turn ≠ 1):
    {R3: in1 ∧ (¬ in0 ∨ at(α0) ∨ turn = 1)}
  CS1
  {R4}
  in1 := false;
  {R5}
  NCS1
}

```

Non-blocking:  $T_0$  before  $NCS_0$ ,  $T_1$  stuck at while loop

$$S_1 \wedge R_2 \wedge in_0 \wedge (turn = 0) = \neg in_0 \wedge in_1 \wedge in_0 \wedge (turn = 0) = false$$

Deadlock-free:  $T_1$  and  $T_0$  at while, before entering the critical section

$$S_2 \wedge R_2 \wedge (in_0 \wedge (turn = 0)) \wedge (in_1 \wedge (turn = 1)) \Rightarrow (turn = 0) \wedge (turn = 1) = false$$

16

## Bounded waiting?

```

Thread T0
while (!terminate) {
  in0 := true;
  turn := 1;
  while (in1 ∧ turn ≠ 0):
    CS0
  in0 := false;
  NCS0
}

Thread T1
while (!terminate) {
  in1 := true;
  turn := 0;
  while (in0 ∧ turn ≠ 1):
    CS1
  in1 := false;
  NCS1
}

```

Yup!

17