Thread Synchronization: Too Much Milk

Implementing Critical Sections in Software Hard

- The following example will demonstrate the difficulty of providing mutual exclusion with memory reads and writes
  - Hardware support is needed
- The code must work *all* of the time
  - Most concurrency bugs generate correct results for *some* interleavings
- Designing mutual exclusion in software shows you how to think about concurrent updates
  - Always look for what you are checking and what you are updating
  - A meddlesome thread can execute between the check and the update, the dreaded race condition

Thread Coordination

Too much milk!

Jack

- Look in the fridge; out of milk
- Go to store
- Buy milk
- Arrive home; put milk away

Jill

- Look in fridge; out of milk
- Go to store
- Buy milk
- Arrive home; put milk away
- Oh, no!

Fridge and milk are shared data structures

Formalizing "Too Much Milk"

- Shared variables
  - "Look in the fridge for milk" – check a variable
  - "Put milk away" – update a variable
- Safety property
  - At most one person buys milk
- Liveness
  - Someone buys milk when needed
- How can we solve this problem?

How to think about synchronization code

- Every thread has the same pattern
  - Entry section: code to attempt entry to critical section
  - Critical section: code that requires isolation (e.g., with mutual exclusion)
  - Exit section: cleanup code after execution of critical region
  - Non-critical section: everything else
- There can be multiple critical regions in a program
  - Only critical regions that access the same resource (e.g., data structure) need to synchronize with each other

```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

The correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if some thread takes forever in non-critical region
- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

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

What if we switch the order of checks?

- 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)

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:

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

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) {
    in0:= true;
    turn := Jill;
    while (turn == Jill
        && in1) ;//wait
    Critical section
    in0 := false;
    Non-critical section
}
```

```
Jill
while (1) {
    in1:= true;
    turn := Jack;
    while (turn == Jack
        && in0);//wait
    Critical section
    in1 := 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 $entry_i$

$$(\neg in_j \lor (in_j \land turn = i)) \land in_i \;=\; (\neg in_j \lor turn = i) \land in_i$$

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 ($in_j \land turn = i$)

How can we do that?

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

We hit a snag

Thread T_0

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

}
```

Thread T_1

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

}
```

The assignment to in_0 invalidates the invariant!

What can we do?

Add assignment to *turn* to establish the second disjunct

Thread T_0

```
while (!terminate) {
      in0 := true;
α0    turn := 1;
      {in0}
      while (in1 ∧ turn ≠ 0);
      {in0 ∧ (¬ in1 ∨ turn = 0 ∨ at(α1) )}
      CS0
      in0 := false;
      NCS0
}
```

Thread T_1

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

Safe?

Thread T_0

```
while (!terminate) {
    in0:= true;
α0  turn := 1;
    {in0}
    while (in1 ∧ turn ≠ 0);
    {in0 ∧ (¬ in1 ∨ turn = 0 ∨ at(α1) )}
    CS0
    in0 := false;
    NCS0
}
```

Thread T_1

```
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

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

Live?

Thread T_0

while (!terminate) {
  $\{S_1: \neg in_0 \wedge (turn = 1 \vee turn = 0)\}$
  in_0 := true;
  $\{S_2: in_0 \wedge (turn = 1 \vee turn = 0)\}$
α_0 $turn$:= 1;
  $\{S_2\}$
  while ($in_1 \wedge turn \neq 0$);
  $\{S_3: in_0 \wedge (\neg in_1 \vee at(\alpha_1) \vee turn = 0)\}$
  CS_0
  $\{S_3\}$
  in_0 := false;
  $\{S_1\}$
  NCS_0
}

Thread T_1

while (!terminate) {
  $\{R_1: \neg in_0 \wedge (turn = 1 \vee turn = 0)\}$
  in_1 := true;
  $\{R_2: in_0 \wedge (turn = 1 \vee turn = 0)\}$
α_1 $turn$:= 0;
  $\{R_2\}$
  while ($in_0 \wedge turn \neq 1$);
  $\{R_3: in_1 \wedge (\neg in_0 \vee at(\alpha_0) \vee turn = 1)\}$
  CS_1
  $\{R_3\}$
  in_1 := false;
  $\{R_1\}$
  NCS_1
}

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

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

Bounded waiting?

Thread T_0	Thread T_1
while (!terminate) {	**while** (!terminate) {
in_0:= true;	in_1:= true;
turn := 1;	*turn* := 0;
while ($in_1 \wedge turn \neq 0$);	**while** ($in_0 \wedge turn \neq 1$);
CS_0	CS_0
in_0 := false;	in_1 := false;
NCS_0	NCS_0
}	}

Yup!