# Safety and liveness for critical sections



- ➤ A. Safety
- B. Liveness
- > C. Both
- A thread that wants to enter the critical section will eventually succeed
  - > A. Safety
  - B. Liveness
  - ➤ C. Both
- Bounded waiting: If a thread *i* is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section (only 1 thread is alowed in at a time) before thread *i*'s request is granted.
  - ➤ A. Safety B. Liveness C. Both

Thread Synchronízatíon: Too Much Mílk

## 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

## 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:

- $\succ$  *in*<sub>i</sub>: thread T<sub>i</sub> 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:

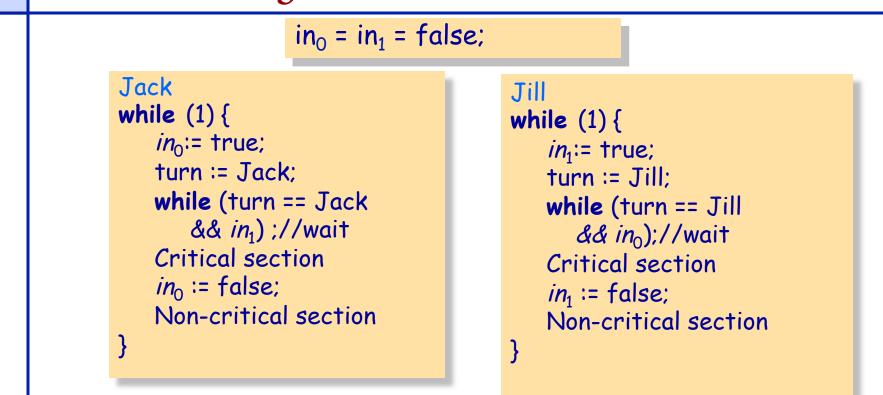
$$\{(\neg in_{j} \lor (in_{j} \land turn = i)) \land in_{i}\}$$

$$CS$$

in<sub>i</sub> = false

 $\begin{array}{l} ((\neg in_0 \lor (in_0 \land turn = 1)) \land in_1) \land \\ ((\neg in_1 \lor (in_1 \land turn = 0)) \land in_0) \\ & \Rightarrow \\ ((turn = 0) \land (turn = 1)) = false \end{array}$ 

# Peterson's Algoríthm



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

The problem boils down to establishing the following right after entry,

 $(\neg in_{i} \lor (in_{i} \land turn = i)) \land in_{i} = (\neg in_{i} \lor turn = i) \land in_{i}$ 

How can we do that?

entry<sub>i</sub> = *in*<sub>i</sub> := true; **while** (*in*<sub>j</sub> ∧ *turn* ≠ *i*);

# We hít a snag

```
Thread T_0
while (!terminate) {
                                                Thread T_1
    more true
                                                while (!terminate) {
    while (in_1 \wedge turn \neq 0);
                                                    in<sub>1</sub>:= true
                                                    \{in_1\}
    \{in_0 \land (\neg in_1 \lor turn = 0)\}
                                                    while (in_0 \land turn \neq 1);
    CS_0
                                                              in, ∨ turn = 1)}
    ....
                                                     CS_1
}
                                                     ....
                                                }
                                                            The assignment to in<sub>0</sub>
                                                         invalidates the invariant!
```

# What can we do?

### Add assignment to *turn* to establish the second disjunct

```
Thread T_0

while (!terminate) {

in_0:= true;

\alpha_0 turn := 1;

\{in_0\}

while (in_1 \land turn \neq 0);

\{in_0 \land (\neg in_1 \lor turn = 0 \lor at(\alpha_1))\}

CS_0

in_0 := false;

NCS_0

}
```

```
Thread T<sub>1</sub>

while (!terminate) {

in_1:= true;

\alpha_1 turn := 0;

\{in_1\}

while (in_0 \land turn \neq 1);

\{in_1 \land (\neg in_0 \lor turn = 1 \lor at(\alpha_0))\}

CS_1

in_1 := false;

NCS_1

}
```

Safe?

```
Thread T<sub>0</sub>

while (!terminate) {

in_0:= true;

\alpha_0 turn := 1;

\{in_0\}

while (in_1 \land turn \neq 0);

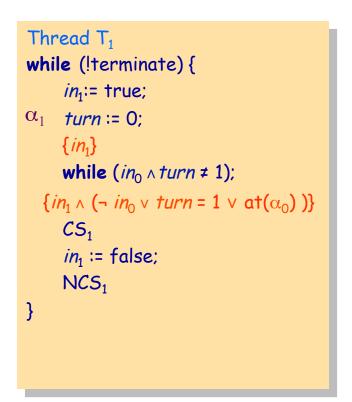
\{in_0 \land (\neg in_1 \lor turn = 0 \lor at(\alpha_1))\}

CS_0

in_0 := false;

NCS_0

}
```



#### If both in CS, then

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

# Líve?

```
Thread T_0

while (!terminate) {

{S_1: \neg in_0 \land (turn = 1 \lor turn = 0)}

in_0:= true;

{S_2: in_0 \land (turn = 1 \lor turn = 0)}

\alpha_0 turn := 1;

{S_2}

while (in_1 \land turn \neq 0);

{S_3: in_0 \land (\neg in_1 \lor at(\alpha_1) \lor turn = 0)}

CS_0

{S_3}

in_0 := false;

{S_1}

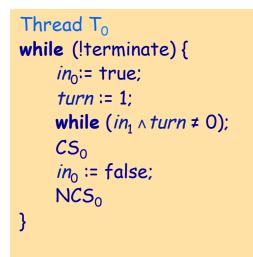
NCS_0

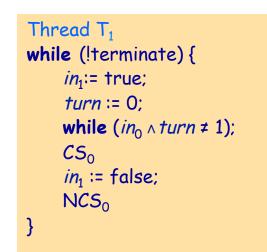
}
```

```
Thread T<sub>1</sub>
while (!terminate) {
       \{\mathsf{R}_1: \neg in_0 \land (turn = 1 \lor turn = 0)\}
       in_1:= true;
       {R<sub>2</sub>: in_0 \land (turn = 1 \lor turn = 0)}
\alpha_1 turn := 0;
       \{R_2\}
       while (in_0 \wedge turn \neq 1);
      {R<sub>3</sub>: in_1 \land (\neg in_0 \lor at(\alpha_0) \lor turn = 1)}
      CS_1
       \{R_3\}
      in1 := false;
       \{\mathsf{R}_1\}
       NCS<sub>1</sub>
```

Non-blocking:  $T_0$  before NCS<sub>0</sub>,  $T_1$  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?





Yup!

Mutual Exclusion: Primitives and Implementation Considerations