Critical Sections in Harmony

```
def thread(self):
    while True
        ... # code outside critical section
        ... # code to enter the critical section
        ... # critical section itself
        ... # code to exit the critical section


spawn T1()
spawn T2()
...
```

How do we check mutual exclusion?

How do we check progress?

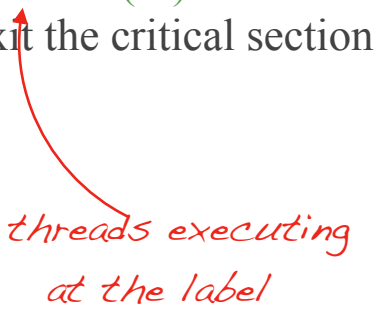
Critical Sections in Harmony

```
def thread(self):
    while True
      …  # code outside critical section
      …  # code to enter the critical section
      …  # critical section itself
      …  # code to exit the critical section


spawn T1()
spawn T2()
…
```

How do we check mutual exclusion?

Critical Sections in Harmony

```
def thread(self):
    while True
      …  # code outside critical section
      …  # code to enter the critical section
      cs:  assert countLabel(cs) ==1
      …  # code to exit the critical section


spawn T1()
spawn T2()
…
```



How do we check mutual exclusion?

Critical Sections in Harmony

```
def thread(self):
    while True
      …  # code outside critical section
      …  # code to enter the critical section
      cs:  assert countLabel(cs) ==1
      …  # code to exit the critical section


spawn T1()
spawn T2()
…
```

How do we check progress?

Critical Sections in Harmony

```
def thread(self):
    while choose({False, True}):
        ...  # code outside critical section
        ...  # code to enter the critical section
        cs:  assert countLabel(cs) ==1
        ...  # code to exit the critical section

spawn T1()
spawn T2()
...
```

thread can choose to enter (or not)

How do we check progress? ✓

If code to enter/exit the critical section cannot terminate, Harmony will complain!

All you need is locks (tatta-rararaaa...)

- At most one thread can hold the lock
- Acquire the lock to enter the CS
- Release the lock when exiting
- But how does one build a lock?

Try 1: A Naïve Lock

```
1   lockTaken = False
2
3   def thread(self):
4       while choose({ False, True }):
5           # Enter critical section
6           await not lockTaken
7           lockTaken = True
8
9           # Critical section
10          cs: assert countLabel(cs) == 1
11
12          # Leave critical section
13          lockTaken = False
14
15  spawn thread(0)
16  spawn thread(1)
```

Try 1: A Naïve Lock

```
1   lockTaken = False
2
3   def thread(self):
4       while choose({ False, True }):
5           # Enter critical section
6           await not lockTaken
7           lockTaken = True
8
9           # Critical section
10          cs: assert countLabel(cs) == 1
11
12          # Leave critical section
13          lockTaken = False
14
15  spawn thread(0)
16  spawn thread(1)
```

Wait till lock is free, then take it

Release the lock

Try 1: A Naïve Lock

```
1   lockTaken = False
2
    def thread(self):
        while choose({ False, True }):
            # Enter critical section
            await not lockTaken
            lockTaken = True

            # Critical section
10          cs: assert countLabel(cs) == 1
11
12          # Leave critical
13          lockTaken = Fals
14
15  spawn thread(0)
16  spawn thread(1)
```

Testing and setting the lock is not atomic!

Wait till lock is free, then take it

Summary: something went wrong in an execution

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init**()
  - Line 1: Initialize lockTaken to False
  - **Thread terminated**
- Schedule thread T2: thread(1)
  - Line 4: Choose True
  - Preempted in thread(1) about to store True into lockTaken in line 7
- Schedule thread T1: thread(0)
  - Line 4: Choose True
  - Line 7: Set lockTaken to True (was False)
  - Preempted in thread(0) about to execute atomic section in line 10
- Schedule thread T2: thread(1)
  - Line 7: Set lockTaken to True (unchanged)
  - Line 10: Harmony assertion failed

Try 2: Flags

```
1   flags = [ False, False ]
2
3   def thread(self):
4       while choose({ False, True }):
5           # Enter critical section
6           flags[self] = True
7           await not flags[1 − self]
8
9           # Critical section
10          cs: assert countLabe
11
12          # Leave critical sectio
13          flags[self] = False
14
15  spawn thread(0)
16  spawn thread(1)
```

Invariant:
Thread i in CS
⟹
flag[i] = True

Signal you want to enter

If someone in the

Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init**()
  - Line 1: Initialize flags to [False, False]
  - **Thread terminated**
- Schedule thread T2: thread(1)
  - Line 4: Choose True
  - Line 6: Set flags[1] to True (was False)
  - Preempted in thread(1) about to load variable flags[0] in line 7
- Schedule thread T1: thread(0)
  - Line 4: Choose True
  - Line 6: Set flags[0] to True (was False)

Final state (all threads have terminated or are blocked):

- Threads:
  - T1: (blocked) thread(0)
    - about to load variable flags[1] in line 7
  - T2: (blocked) thread(1)
    - about to load variable flags[0] in line 7
- Variables:
  - flags: [True, True]

Try 3: Turns

```
1   turn = 0
2
    def thread(self):
        while choose({ False, True }):
            # Enter critical section
            turn = 1 − self
            await turn == self
9           # Critical section
10          cs: assert countLabel(cs)
11
12          # Leave critical section
13
14  spawn thread(0)
15  spawn thread(1)
```

Invariant:
Thread i in CS
⟹
turn = i

After you...

Wait for your turn

Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init**()
  - Line 1: Initialize turn to 0
  - **Thread terminated**
- Schedule thread T1: thread(0)
  - Line 4: Choose False
  - **Thread terminated**
- Schedule thread T2: thread(1)
  - Line 4: Choose True
  - Line 6: Set turn to 0 (unchanged)

Final state (all threads have terminated or are blocked):

- Threads:
  - T1: (terminated) thread(0)
  - T2: (blocked) thread(1)
    - about to load variable turn in line 7
- Variables:
  - turn: 0

Peterson's Algorithm: Flags and Turns!

```
1   sequential flags, turn
2
3   flags = [ False, False ]
4   turn = choose({0, 1})
5
6   def thread(self):
7       while choose({ False, True }):
8           # Enter critical section
9           flags[self] = True
10          turn = 1 - self
11          await (not flags[1 - self]) or (turn == self)
12
13          # Critical section is here
14          cs: assert countLabel(cs) == 1
15
16          # Leave critical section
17          flags[self] = False
18
19  spawn thread(0)
20  spawn thread(1)
```

Prevents out-of-order execution

I'd like to enter...

...but you go first!

Wait until alone or it's my turn

Leave

```
[dhcp-vl2041-5018:~/Documents/harmony/code] lorenzo% harmony Peterson.hny
* Phase 1: compile Harmony program to bytecode
* Phase 2: run the model checker (nworkers = 8)
    * 104 states (time 0.00s, mem=0.000GB)
* Phase 3: analysis
    * 37 components (0.00 seconds)
    * Check for data races
    * **No issues found**
* Phase 4: write results to Peterson.hco
* Phase 5: loading Peterson.hco
```

What about a proof?

- To understand why it works...
- We need to show that, for any execution, all states reached satisfy mutual exclusion
  - i.e., that mutual exclusion is an invariant
- See the Harmony book for a proof!
  - or come talk to me!

Once More unto the Breach: Taking Turns

Thread T_0	Thread T_1
$in_0 := true$	$in_1 := true$
await $\neg in_1$	await $\neg in_0$

The above condition for entering CS_i is too strong: we weaken it by adding turns

Even if in_1, if it is T_0's turn, then T_0 is allowed to enter CS_0

Invariant I: $turn = 0 \vee turn = 1$

The new entry code then is

Thread T_0	Thread T_1
$in_0 := true$	$in_1 := true$
await $\neg in_1 \vee (turn = 0)$	await $\neg in_0 \vee (turn = 1)$

Critical Section: Taking Turns

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

await $\neg in_1 \vee (turn = 0)$

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

await $\neg in_0 \vee (turn = 1)$

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Critical Section: Taking Turns

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Critical Section: Taking Turns

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \ \{\neg in_1 \wedge I\}$

NCS_1

}

Interference Freedom

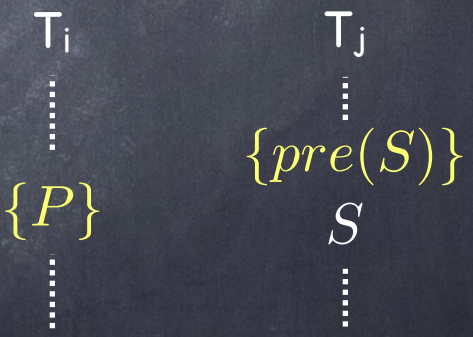
- By executing $in_1 := true$, T_1 can interfere on the truth of T_0's assertion! (and symmetrically for T_0)

$$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$$

- In general, interference freedom requires to establish

$$\{pre(S) \wedge P\} \;\; S \;\; \{P\}$$

for all S in one thread and P in the other



Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true$ $\{in_0 \wedge I\}$ ✓

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$ ✘

CS_0

$in_0 := false$ $\{\neg in_0 \wedge I\}$ ✓

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true$ $\{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false$ $\{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true$ $\{in_0 \wedge I\}$ ✓ ✓

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$ ✗ ✓

CS_0

$in_0 := false$ $\{\neg in_0 \wedge I\}$ ✓ ✓

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true$ $\{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false$ $\{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$\langle in_0 := true \quad \{in_0 \wedge I\}$
$\quad turn = 1 \quad \{in_0 \wedge I\} \rangle$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$\langle in_1 := true \quad \{in_1 \wedge I\}$
$\quad turn = 0 \quad \{in_1 \wedge I\} \rangle$

Operations execute atomically

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Taking stock

- We solved the critical section problem, as long as we know how to execute multiple operations atomically
  - in other words, we can solve the CS problem as long as we can solve the CS problem... sigh...
  - besides, no machine instruction allows for those operations to execute atomically...
- But what if we don't execute the entry code atomically? Where is the problem?

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$\langle in_0 := true \quad \{in_0 \wedge I\}$
$\quad turn = 1 \quad \{in_0 \wedge I\} \rangle$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$\langle in_1 := true \quad \{in_1 \wedge I\}$
$\quad turn = 0 \quad \{in_1 \wedge I\} \rangle$

Operations execute atomically

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

```
while(!terminate) {
```
$in_0 := true \quad \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

```
while (in1 ∧ turn ≠ 0);
```
$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \quad \{\neg in_0 \wedge I\}$

NCS_0

```
}
```

Thread T_1

```
while(!terminate) {
```
PC_{T_1} →
$in_1 := true \quad \{in_1 \wedge I\}$ No problem!

$turn = 0 \quad \{in_1 \wedge I\}$

```
while (in0 ∧ turn ≠ 1);
```
$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \{\neg in_1 \wedge I\}$

NCS_1

```
}
```

Establishing Interference Freedom

Thread T_0

while(!terminate) $\{$

$in_0 := true \;\; \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \;\; \{\neg in_0 \wedge I\}$

NCS_0

$\}$

Thread T_1

while(!terminate) $\{$

$in_1 := true \;\; \{in_1 \wedge I\}$

$turn = 0 \quad \{in_1 \wedge I\}$

PC_{T_1} → while $(in_0 \wedge turn \neq 1);$ No problem!

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \; \{\neg in_1 \wedge I\}$

NCS_1

$\}$

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

PC_{T_1} → $turn = 0 \quad \{in_1 \wedge I\}$ Problem!

We weaken the assertion

while $(in_0 \wedge turn \neq 1);$

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \ \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

$\xrightarrow{PC_{T_1}}$ $turn = 0 \quad \{in_1 \wedge I\}$ No problem!

We weaken the assertion

while $(in_0 \wedge turn \neq 1);$

$\{in_1 \wedge (\neg in_0 \vee turn = 1) \wedge I\}$

CS_1

$in_1 := false \ \{\neg in_1 \wedge I\}$

NCS_1

}

Establishing Interference Freedom

Thread T_0

while(!terminate) {

$in_0 := true \ \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \ \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \ \{in_1 \wedge I\}$

$\xrightarrow{PC_{T_1}}$ $turn = 0 \quad \{in_1 \wedge I\}$ No problem!

We weaken the assertion

while $(in_0 \wedge turn \neq 1);$

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\}$

CS_1

$in_1 := false \ \{\neg in_1 \wedge I\}$

NCS_1

}

Peterson's Algorithm

Thread T_0

while(!terminate) $\{$

$in_0 := true \;\; \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0);$

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \;\; \{\neg in_0 \wedge I\}$

NCS_0

$\}$

Thread T_1

while(!terminate) $\{$

$in_1 := true \;\; \{in_1 \wedge I\}$

$turn = 0 \quad \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1);$

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\}$

CS_1

$in_1 := false \; \{\neg in_1 \wedge I\}$

NCS_1

$\}$

Peterson's Algorithm: Safety

Thread T_0

while(!terminate) {

$in_0 := true \quad \{in_0 \wedge I\}$

$turn = 1 \quad \{in_0 \wedge I\}$

while $(in_1 \wedge turn \neq 0)$;

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\}$

CS_0

$in_0 := false \quad \{\neg in_0 \wedge I\}$

NCS_0

}

Thread T_1

while(!terminate) {

$in_1 := true \quad \{in_1 \wedge I\}$

$turn = 0 \quad \{in_1 \wedge I\}$

while $(in_0 \wedge turn \neq 1)$;

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\}$

CS_1

$in_1 := false \, \{\neg in_1 \wedge I\}$

NCS_1

}

Mutual exclusion?

$$
\begin{array}{rl}
in(CS_0) \Rightarrow & \{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(turn = 0)) \wedge I\} \wedge \\
\wedge & \neg at(turn = 1) \wedge \\
in(CS_1) \Rightarrow & \{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(turn = 1)) \wedge I\} \wedge \\
& \neg at(turn = 0) = \\
& in_0 \wedge in_1 \wedge turn = 0 \wedge turn = 1 = false
\end{array}
$$

✓

Peterson: Non-blocking

```
while(!terminate) {
  {R_1 : ¬in_0 ∧ (turn = 1 ∨ turn = 0)}
   in_0 = true
   {R_2 : in_0 ∧ (turn = 1 ∨ turn = 0)}
α_0 turn = 1
   {R_2}
   while(in_1 ∧ turn ≠ 0);
   {R_3 : in_0 ∧ (¬in_1 ∨ turn = 0 ∨ at(α_1))}
    CS_0
   {R_3}
   in_0 = false
   {R_1}
   NCS_0
}
```

T0's PC

```
while(!terminate) {
  {S_1 : ¬in_1 ∧ (turn = 1 ∨ turn = 0)}
   in_1 := true
   {S_2 : in_1 ∧ (turn = 1 ∨ turn = 0)}
α_1 turn := 0
   {S_2}
   while(in_0 ∧ turn ≠ 1);
   {S_3 : in_1 ∧ (¬in_0 ∨ turn = 1 ∨ at(α_0))}
    CS_1
   {S_3}
   in_1 = false
   {S_1}
   NCS_1
}
```

T1's PC

Blocking Scenario: T_0 before NCS_0, T_1 stuck at while loop

$$R_1 \wedge S_2 \wedge in_0 \wedge (turn = 0) = \neg in_0 \wedge in_1 \wedge in_0 \wedge (turn = 0) = false$$

Peterson: Deadlock-free

```
while(!terminate) {
  {R_1 : ¬in_0 ∧ (turn = 1 ∨ turn = 0)}
   in_0 = true
   {R_2 : in_0 ∧ (turn = 1 ∨ turn = 0)}
α_0 turn = 1
   {R_2}
   while(in_1 ∧ turn ≠ 0);
   {R_3 : in_0 ∧ (¬in_1 ∨ turn = 0 ∨ at(α_1))}
    CS_0
   {R_3}
    in_0 = false
   {R_1}
   NCS_0
}
```

T0's PC

```
while(!terminate) {
  {S_1 : ¬in_1 ∧ (turn = 1 ∨ turn = 0)}
   in_1 := true
   {S_2 : in_1 ∧ (turn = 1 ∨ turn = 0)}
α_1 turn := 0
   {S_2}
   while(in_0 ∧ turn ≠ 1);
   {S_3 : in_1 ∧ (¬in_0 ∨ turn = 1 ∨ at(α_0))}
    CS_1
   {S_3}
    in_1 = false
   {S_1}
   NCS_1
}
```

T1's PC

Blocking Scenario: T_0 and T_1 at the while loop, before entering critical section

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