Concurrent Programming: Critical Sections

CS 6410

[Robbert van Renesse]
Concurrent Programming is Hard

Why?

- Concurrent programs are \textit{non-deterministic}
  - run them twice with same input, get two different answers
  - or worse, one time it works and the second time it fails
- Program statements are executed \textit{non-atomically}
  - $x += 1$ compiles to something like
    - LOAD $x$
    - ADD 1
    - STORE $x$
Enter *Harmony*

- A new concurrent programming language
  - heavily based on Python syntax to reduce learning curve for many
- A new underlying virtual machine

It tries *all* possible executions of a program (or rather, explores all possible reachable states) until it finds a problem, if any

(this is called “model checking” )
Non-Determinism

(a) [code/prog1.hny] Sequential

```
1 shared = True
2
def f(): assert shared
3 def g(): shared = False
4
5 f()
6 g()
```

(b) [code/prog2.hny] Concurrent

```
1 shared = True
2
def f(): assert shared
3 def g(): shared = False
4
5 spawn f()
6 spawn g()
```

Figure 3.1: A sequential and a concurrent program.
Non-Determinism

(a) [code/prog1.hny] Sequential

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed
Critical Section

Must be serialized due to shared memory access

Goals

Mutual Exclusion: 1 thread in a critical section at time

Progress: at least one thread makes it into the CS if desired and no other thread is there

Fairness: equal chances of getting into CS

... in practice, fairness rarely guaranteed or needed
Mutual Exclusion and Progress

• Need both:
  o either one is trivial to achieve by itself
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 thread(1)
spawn thread(2)
…

• How do we check mutual exclusion?
• How do we check progress?
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 thread(1)
spawn thread(2)
...

• How do we check mutual exclusion?
• How do we check progress?
Critical Sections in Harmony

```python
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 thread(1)
    spawn thread(2)
    ...
```

- How do we check mutual exclusion?
- How do we check progress?
  - *if code to enter/exit the critical section cannot terminate, Harmony with balk*
Sounds like you need a lock…

• True, but this is an O.S. class!
• The question is:

\textit{How does one build a lock?}

• Harmony is a concurrent programming language. \textit{Really, doesn’t Harmony have locks?}

\textit{You have to program them!}
First attempt: a naïve lock

```python
lockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

    spawn thread(0)
    spawn thread(1)
```

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.
First attempt: a naïve lock

```
lockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

spawn thread(0)
spawn thread(1)
```

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.
First attempt: a naïve lock

```python
clockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

    spawn thread(0)
    spawn thread(1)
```

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.
First attempt: a naïve lock

```
lockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

spawn thread()
spawn thread()
```

Figure 5.3: [code/naiveLock.hny] Naive implementation of a shared lock.
Second attempt: flags

```python
flags = [False, False]

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        # Critical section
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.
Second attempt: \textit{flags}

\begin{verbatim}
flags = [False, False]

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        # Critical section
        @cs: assert atLabel(cs) == {thread, self): 1}

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
\end{verbatim}

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.
Second attempt: *flags*

```python
flags = [False, False]

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        # Critical section
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.
Second attempt: flags

```python
def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        # Critical section
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.
Second attempt: flags

```python
flags = [False, False]

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
    await not flags[1 - self]

    # Critical section
    @cs: assert atLabel(cs) == {thread, self): 1}

    # Leave critical section
    flags[self] = False

spawn thread(0) __init__() [0,36-46] 46 {flags: [False, False]}
spawn thread(1) thread/0 [1-2,3(choose True),4-12] 13 {flags: [True, False]}
thread/1 [1-2,3(choose True),4-12] 13 {flags: [True, True]}

blocked thread: thread/1 pc = 13
blocked thread: thread/0 pc = 13
```

Figure 5.5: [code/notebook]
Third attempt: \textit{turn} variable

```python

```turn = 0

```def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

        spawn thread(0)
        spawn thread(1)

```

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.
Third attempt: *turn* variable

```python
turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

        spawn thread(0)
        spawn thread(1)
```

Figure 5.7: [code/naiveTurn.hny](code/naiveTurn.hny) Naïve use of turn variable to solve mutual exclusion.
Third attempt: \textit{turn} variable


def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

        spawn thread(0)
        spawn thread(1)

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.
Third attempt: \textit{turn} variable

\begin{verbatim}
313
314        def thread(self):
315            while choose({ False, True }):
316                # Enter critical section
317                turn = 1 - self
318                await turn == self
319
320                # Critical section
321                @cs: assert atLabel(cs) == { (thread, self): 1 }
322
323                # Leave critical section
324
325                spawn thread(0)
326                spawn thread(1)
\end{verbatim}

\textbf{Figure 5.7: [code]}

\begin{itemize}
\item \texttt{__init__}() [0,28-38] 38 \{ turn: 0 \}
\item \texttt{thread/0} [1-2,3(choose True),4-26,2,3(choose True),4] 5 \{ turn: 1 \}
\item \texttt{thread/1} [1-2,3(choose False),4,27] 27 \{ turn: 1 \}
\item blocked thread: \texttt{thread/0 pc = 5}
\end{itemize}
Peterson’s Algorithm: flags & turn

```python
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
Peterson’s Algorithm: flags & turn

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

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
Peterson’s Algorithm: flags & turn

```python
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 atLabel(cs) == { (thread, self): 1 }  
15
16         # Leave critical section
17         flags[self] = False
18
19         spawn thread(0)
20         spawn thread(1)
```

Figure 6.1: [code/Peterson.lny] Peterson’s Algorithm
Peterson’s Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

Figure 6.1: [code/Peterson.hny](code/Peterson.hny) Peterson’s Algorithm
Peterson’s Algorithm: flags & turn

```
sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

        spawn thread(0)
        spawn thread(1)
```

“you go first”
wait until alone or it’s my turn
leave

Figure 6.1: [code/Peterson.lhn] Peterson’s Algorithm
Peterson’s Algorithm: flags & turn

```python
sequential flags, turn

flags = [False, False]

turn = choose({0, 1})

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)

spawn thread(1)
```

#states = 104 diameter = 5
#components: 37
no issues found

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
So, we proved Peterson’s Algorithm correct by brute force, enumerating all possible executions. We now know *that* it works.

*But how does one prove it by deduction? so one understands *why* it works…*
What and how?

- Need to show that, for any execution, all states reached satisfy mutual exclusion
  - in other words, mutual exclusion is *invariant*

\[ \text{invariant} = \text{predicate that holds in every reachable state} \]
How to prove an invariant?

• Need to show that, for any execution, all states reached satisfy the invariant

• Sounds similar to sorting:
  ○ Need to show that, for any list of numbers, the resulting list is ordered

• Let’s try proof by induction on the length of an execution
Proof by induction

You want to prove that some *Induction Hypothesis* $IH(n)$ holds for any $n$:

- **Base Case:**
  - show that $IH(0)$ holds

- **Induction Step:**
  - show that if $IH(i)$ holds, then so does $IH(i+1)$
Proof by induction in our case

To show that some IH holds for an *execution* E of any number of *steps*:

- **Base Case:**
  - show that IH holds in the initial state(s)

- **Induction Step:**
  - show that if IH holds in a state produced by E, then for any possible next step s, IH also holds in the state produced by E + [s]
But there’s a problem

• How do we characterize a “state produced by E”?  
  o or how do we characterize a reachable state?
• Instead, it’s much easier if we proved a so-called “inductive invariant”:
  o Base Case:
    - show that IH holds in the initial state(s)
  o Induction Step:
    - show that if IH holds in any state, then for any possible next step, IH also holds in the resulting state
First question: what should IH be?

• Obvious answer: mutual exclusion itself
  o if $T0$ is in the critical section, then $T1$ is not
    – without loss of generality…
  o Formally: $T0@cs \implies \neg T1@cs$

• Unfortunately, this won’t work…
State before T1 takes a step:

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

flags = [ True, True ]
turn = 1

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
State after T1 takes a step:

```python
sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

flags = [ True, True ]
turn = 1

mutual exclusion violated

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
So, is Peterson’s Algorithm broken?
No, it’ll turn out this prior state cannot be reached from the initial state (see later)

```python
sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

flags = [ True, True ]
turn = 1

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
Useful and obvious but insufficient invariant

```python
sequential flags, turn
flags = [False, False]
turn = choose({0, 1})

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
What else do we expect to hold @cs?

```python
sequential flags, turn
flags = [False, False]
turn = choose({0, 1})

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson’s Algorithm
Another obvious IH to try

• Based on the **await** condition:
  \[T0@cs \implies \neg flags[1] \lor turn = 0\]

• Promising because if \(T0@cs \land T1@cs\) then
  \[T0@cs \implies \neg flags[1] \lor turn = 0 \land \begin{cases} turn = 0 \land \\
  T1@cs \implies \neg flags[0] \lor turn = 1 \end{cases} \implies \begin{cases} turn = 1 \land \\
  \end{cases}\]
  \(\implies \text{False} \) (therefore mutual exclusion)

• Unfortunately, this is not an invariant…
• Based on the `await` condition:
  \[ T0@cs \implies \neg flags[1] \lor turn = 0 \]

• Promising because if \( T0@cs \land T1@cs \) then
  \[ T0@cs \implies \neg flags[1] \lor turn = 0 \land T1@cs \implies \neg flags[0] \lor turn = 1 \]
  \[ \implies \text{false} \] (therefore mutual exclusion)

• Unfortunately, this is not an invariant...

Easy to check with Harmony
Just run it with the following:
\[ @cs: \text{assert (not flags[1 - self]) or (turn == self)} \]
State before T1 takes a step:

```python
def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
```

flags = [True, False]
turn = 1

Note: this is a reachable state.
State after T1 takes a step:

```
sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

flags = [ True, True ]
turn = 1

```
T0@cs ⟹ ¬flags[1] ∨ turn = 0 violated
note: this is also a reachable state
```
But suggests an improved hypothesis

\[ T0@cs \Rightarrow \neg \text{flags}[1] \lor \text{turn} = 0 \lor T1@gate \]

```python
flags = [False, False]
turn = choose({0, 1})

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        @gate: turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # Critical section
        @cs: assert (not flags[1 - self]) or (turn == self) or (atLabel(gate) == {(thread, 1 - self): 1})

        # Leave critical section
        flags[self] = False
```
But suggests an improved hypothesis

\[ T_0@cs \implies \neg \text{flags}[1] \lor \text{turn} = 0 \lor T_1@gate \]

\text{flags} = [\text{False, False}]

Also easy to check with Harmony

Proves that it is invariant, but not necessarily an \textit{inductive invariant}

\texttt{@cs: assert (not flags[1 - self]) or (turn == self) or (atLabel(gate) == {(thread, 1 - self): 1})}

\texttt{# Leave critical section flags[self] = False}
Inductive Invariance Proof

Let $I$ be the induction hypothesis:

$$I \equiv T_{0}@cs \implies \neg flags[1] \lor turn == 0 \lor T_{1}@gate$$

$I$ clearly holds in the initial state because $\neg T_{0}@cs$ (false implies anything)

We are going to show: if $I$ holds in a state (reachable or not), then $I$ also holds in any state after either $T_{0}$ or $T_{1}$ takes a step.
Tricky Case 1:

\( \neg T0@cs \) and \( T0 \) takes a step so that \( T0@cs \)

This must mean that \( \neg flags[1] \lor turn = 0 \)
before the step (see code line 11)

But then \( \neg flags[1] \lor turn = 0 \) still holds after the step

So \( T0@cs \Rightarrow \neg flags[1] \lor turn = 0 \lor T1@gate \)
Tricky Case 2:

$T0@cs$ and $T1$ takes a step

This must mean that before the step

$\neg flags[1] \lor turn = 0 \lor T1@gate$ (by IH).

So, 3 cases to consider:

- $\neg flags[1] \Rightarrow flags[1]$
  - this means $T1@gate$ after the step
- $turn = 0 \Rightarrow turn = 1$
  - can’t happen (only $T0$ sets $turn$ to 1)
- $T1@gate \Rightarrow \neg T1@gate$
  - this means $turn = 0$ after step

So, $T0@cs \Rightarrow \neg flags[1] \lor turn = 0 \lor T1@gate$
Finally, prove mutual exclusion

\[ T0@cs \land T1@cs \implies \]
\[ \begin{cases} 
  \neg flags[1] \lor turn = 0 \lor T1@gate \\
  \neg flags[0] \lor turn = 1 \lor T0@gate 
\end{cases} \land \\
\implies turn = 0 \land turn = 1 \\
\implies False \]
Finally, prove mutual exclusion:

\[ T0@cs \land T1@cs \implies \]
\[ \neg flags[1] \lor \text{turn} = 0 \lor T1@gate \land \]
\[ \neg flags[0] \lor \text{turn} = 1 \lor T0@gate \implies \text{turn} = 0 \land \text{turn} = 1 \implies False \]

QED
Now we can see why this state cannot be reached!

```python
def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)
        # critical section is here
        @cs: assert atLabel(cs) == {thread, self): 1 }
        # Leave critical section
        flags[self] = False
        spawn thread(0)
        spawn thread(1)
```

flags = [True, True]
turn = 1

$T_0@cs \implies \neg flags[1] \lor turn = 0 \lor T_1@gate \times$
property = set of states
property = set of states
Review in Pictures: State Space

Mutual Exclusion Holds

property = set of states

Mutual Exclusion Violated

mutual exclusion is not inductive
Reachable States

property = set of states

subset = implication

Mutual Exclusion Holds

Mutual Exclusion Violated
property = set of states
property = set of states
Swapping lines 9 and 10?

```python
    sequential flags, turn

    flags = [ False, False ]
    turn = choose({0, 1})

    def thread(self):
        while choose({ False, True }):
            # Enter critical section
            flags[self] = True
            turn = 1 - self
            await (not flags[1 - self]) or (turn == self)

            # critical section is here
            @cs: assert atLabel(cs) == { (thread, self): 1 }

            # Leave critical section
            flags[self] = False

        spawn thread(0)
        spawn thread(1)
```

Figure 6.1: [code/Peterson.lhsy] Peterson’s Algorithm