Concurrent Programming: Critical Sections and Locks

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[Robbert van Renesse]

An Operating System is a Concurrent Program

- The ”kernel contexts” of each of the processes share many data structures
  - ready queue, wait queues, file system cache, and much more
- Sharing is further complicated by interrupt handlers that also access those data structures

So I talked with a recruiter . . .

Synchronization Lectures Outline

- What is the problem?
  - no determinism, no atomicity
- What is the solution?
  - some form of locks
- How to implement locks?
  - there are multiple ways
- How to reason about concurrent programs?
  - invariants
- How to construct correct concurrent code?
  - Lorenzo loves cocococo

Concurrent Programming is Hard

Why?

- Concurrent programs are *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 non-atomically
  - **x += 1** compiles to something like
    - **LOAD x**
    - **ADD 1**
    - **STORE x**

Terminology warning

- I use the terms “thread” and “process” interchangeably
    - threads are application processes that share the same address space

Non-Determinism

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

Non-Determinism

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
spawn g()
```

(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

Non-Determinism

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
spawn g()
```

(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

Non-Determinism

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
spawn g()
```

(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
```

Non-Atomicity

2 threads updating a shared variable **amount**

- One thread (you) wants to decrement amount by $10K
- Other thread (IRS) wants to decrement amount by 50%

T1

```
. . .
amount -= 10,000
. . .
```

T2

```
. . .
amount /= 2
. . .
```

Memory amount 100,000

What happens when both threads are running?

Non-Atomicity

Might execute like this:

T2

```
. . .
r2 = load from amount
r2 = r2 / 2
store r2 to amount
. . .
```

T1

```
. . .
r1 = load from amount
r1 = r1 – 10,000
store r1 to amount
. . .
```

Memory amount 40,000

Or vice versa (T1 then T2 → 45,000)…
either way is fine…

Non-Atomicity

Or it might execute like this:

T1

```
. . .
r1 = load from amount
r1 = r1 – 10,000
store r1 to amount
. . .
```

T2

```
. . .
r2 = load from amount

. . .

r2 = r2 / 2
store r2 to amount
. . .
```

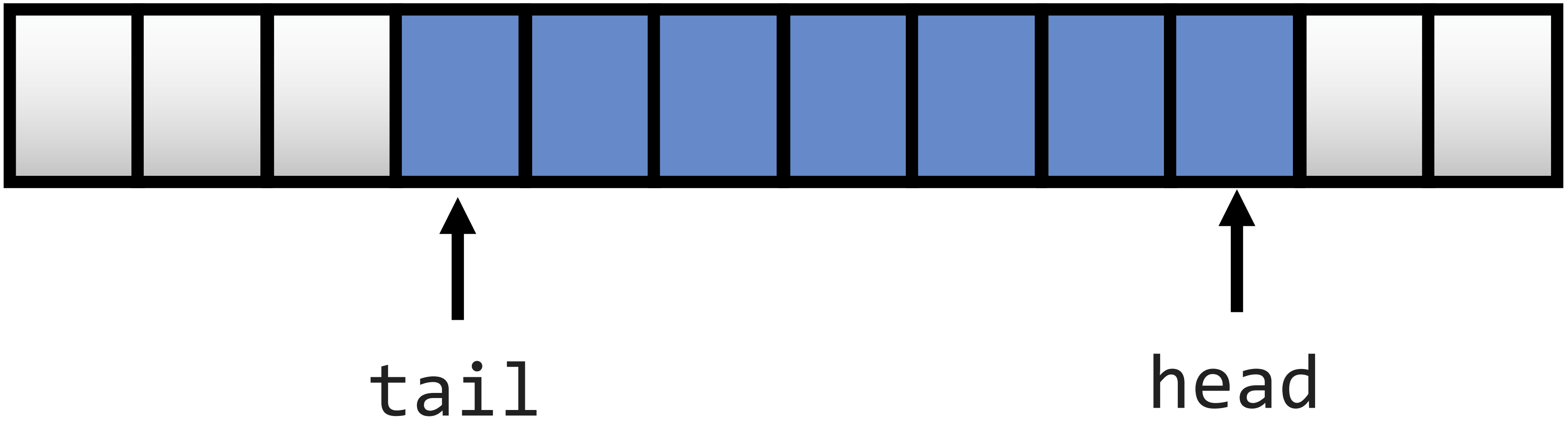
Memory amount 50,000

Lost Update!

Wrong ..and very difficult to debug

Example: Races with Shared Queue

- 2 concurrent enqueue() operations?
- 2 concurrent dequeue() operations?



What could possibly go wrong?

Race Conditions

= *timing dependent error involving shared state*

- Once thread A starts, it needs to “race” to finish
- Whether race condition happens depends on thread schedule
  - Different “schedules” or “interleavings” exist

    (a schedule is a total order on machine instructions)

All possible interleavings should be safe!

Race Conditions are Hard to Debug

- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad
  - But bad interleavings may rarely happen!
  - Works 100x ≠ no race condition
- Timing dependent: small changes hide bugs
  - add print statement → bug no longer seems to happen

My experience until last spring

1. Students develop their code in Python or C
2. They test by running code many times
3. They submit their code, confident that it is correct
4. RVR tests the code with his secret and evil methods
   - uses homebrew library that randomly samples from possible interleavings (“fuzzing”)
5. Finds most submissions are broken
6. RVR unhappy, students unhappy

It's not stupidity

- Several studies show that heavily used code implemented, reviewed, and tested by expert programmers have lots of concurrency bugs
- Even professors who teach concurrency or write books about concurrency get it wrong sometimes
  - *last weekend I found a data race in a well-known concurrent queue algorithm using Harmony*

My take on the problem

- Handwritten proofs just as likely to have bugs as programs
  - or even more likely as you can’t test handwritten proofs
- Lack of mainstream tools to check concurrent algorithms
- Tools that do exist are great but have a steep learning curve

Examples of existing tools

```
bool turn, flag[2];          // the shared variables, booleans
byte ncrit;                  // nr of procs in critical section

active [2] proctype user()  // two processes
{
     assert(_pid == 0 || _pid == 1);
again:
     flag[_pid] = 1;
     turn = _pid;
     (flag[1 - _pid] == 0 || turn == 1 - _pid);

     ncrit++;
     assert(ncrit == 1);
     ncrit--;

     flag[_pid] = 0;
     goto again
}
```

Spin

```
--algorithm Peterson {
  variables flag = [i \in {0, 1} |-> FALSE
     \* Declares the global variables flag
     \* flag is a 2-element array with init
  fair process (proc \in {0,1}) {
    \* Declares two processes with identifie
    \* The keyword fair means that no proces
    \* always take a step.
   a1: while (TRUE) {
         skip ;  \* the noncritical section
   a2:   flag[self] := TRUE ;
   a3:   turn := 1 - self ;
   a4:   await (flag[1-self] = FALSE) \/ (turn
   cs:   skip ;  \* the critical section
   a5:   flag[self] := FALSE
                                               }
```

PlusCal

TLA+

VARIABLES $flag, turn, pc$

$vars \triangleq \langle flag, turn, pc \rangle$

$Init \triangleq \land\ flag = [i \in \{0, 1\} \mapsto \text{FALSE}]$
$\quad\quad\quad \land\ turn = 0$
$\quad\quad\quad \land\ pc = [self \in \{0, 1\} \mapsto \text{“a0”}]$

$a3a(self) \triangleq$
$\quad \land\ pc[self] = \text{“a3a”}$
$\quad \land\ \text{IF } flag[Not(self)]$
$\quad\quad \text{THEN } pc' = [pc \text{ EXCEPT } ![self] = \text{“a3b”}]$
$\quad\quad \text{ELSE } pc' = [pc \text{ EXCEPT } ![self] = \text{“cs”}]$
$\quad \land\ \text{UNCHANGED } \langle flag, turn \rangle$

* remaining actions omitted

$proc(self) \triangleq a0(self) \lor \ldots \lor a4(self)$

$Next \triangleq \exists\, self \in \{0, 1\} : proc(self)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

Enter *Harmony*

- A new concurrent programming language
  - heavily based on Python syntax to reduce learning curve for many
- A new underlying virtual machine
  - quite different from any other:

it tries *all* possible executions of a program until it finds a problem, if any

(this is called "model checking")

Example (same as before)

```
def T1():
    amount -= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True
```

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Equivalent to:

```
while not (done1 and done2):
    pass
```

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Assertion: useful to check properties

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Output amount if assertion fails

An important note on assertions

- An assertion is not part of your algorithm
- If it fails, the thread that is running *is still running and will continue to run*
- Homework Hint: Harmony reports the state at the time an assertion fails, but that does not mean that the threads have finished or terminated
- Semantically, an assertion is a no-op

That said…

- Assertions are super-useful
  - *@label*: **assert** P is a type of *invariant*:

$$pc = label \;\Rightarrow P$$

- Use them liberally
  - In C, Java, …, they're automatically removed in production code
  - Or automatically optimized out if you have a really good compiler
- They are great for testing
- They are *executable documentation*
  - comments tend to get outdated over time

That said…

That said…

Comment them out before you submit a programming assignment

- you don’t want your assertions to fail while we are testing your code ☺

Back to example

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Initialize shared variables

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Spawn three processes (threads)

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5  { amount: 100000, done1: False, done2: False }
T2/() [10-17].        17 { amount: 50000,   done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,   done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

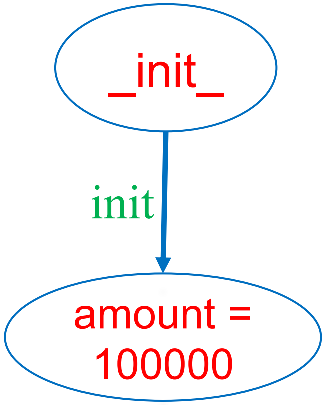
Simplified model (ignoring main)

T1a: LOAD amount	T2a: LOAD amount
T1b: SUB 10000	T2b: DIV 2
T1c: STORE amount	T2c: STORE amount

Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

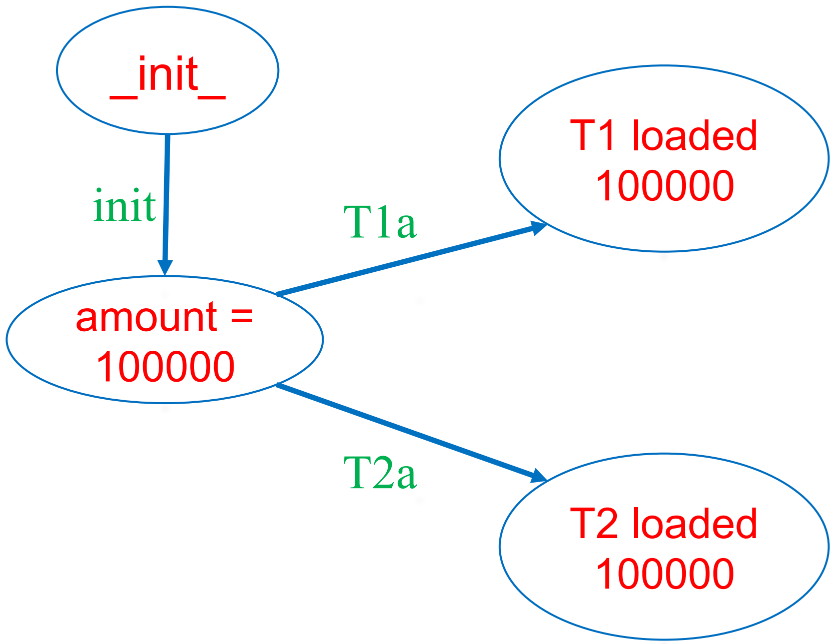
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

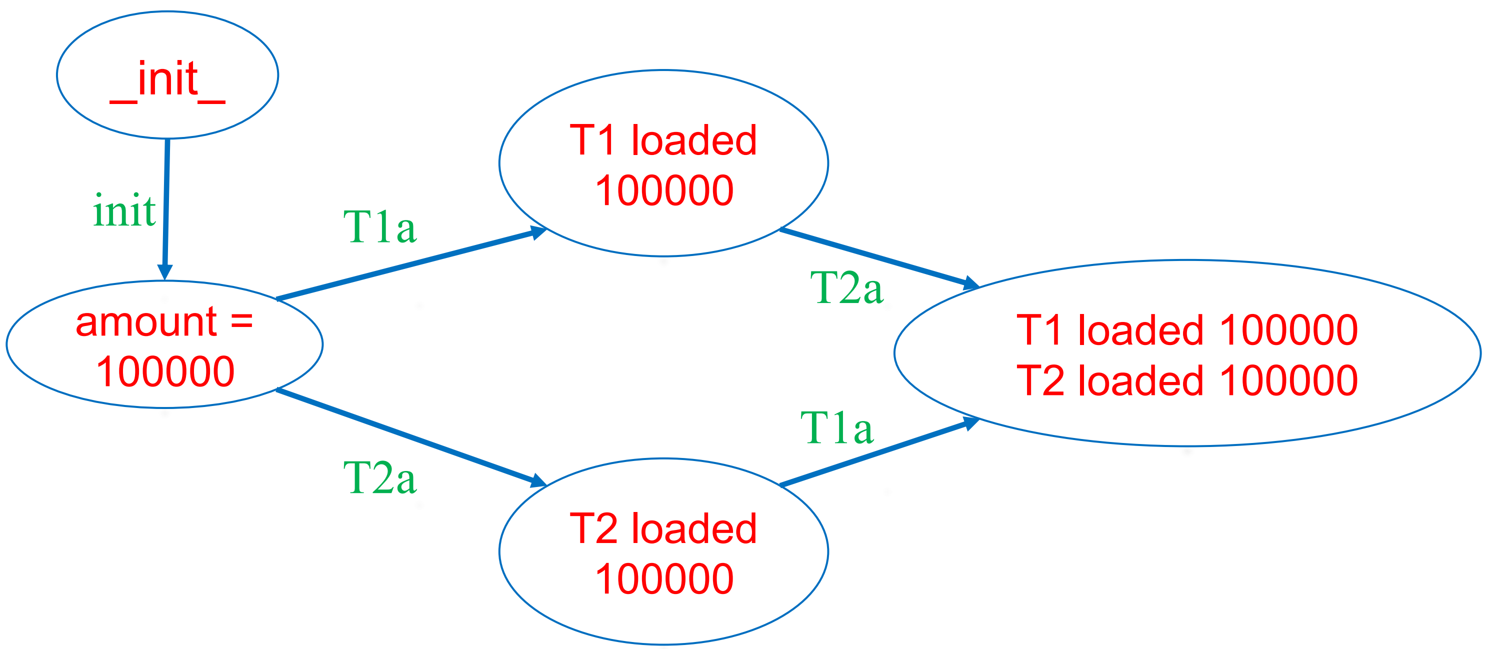
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

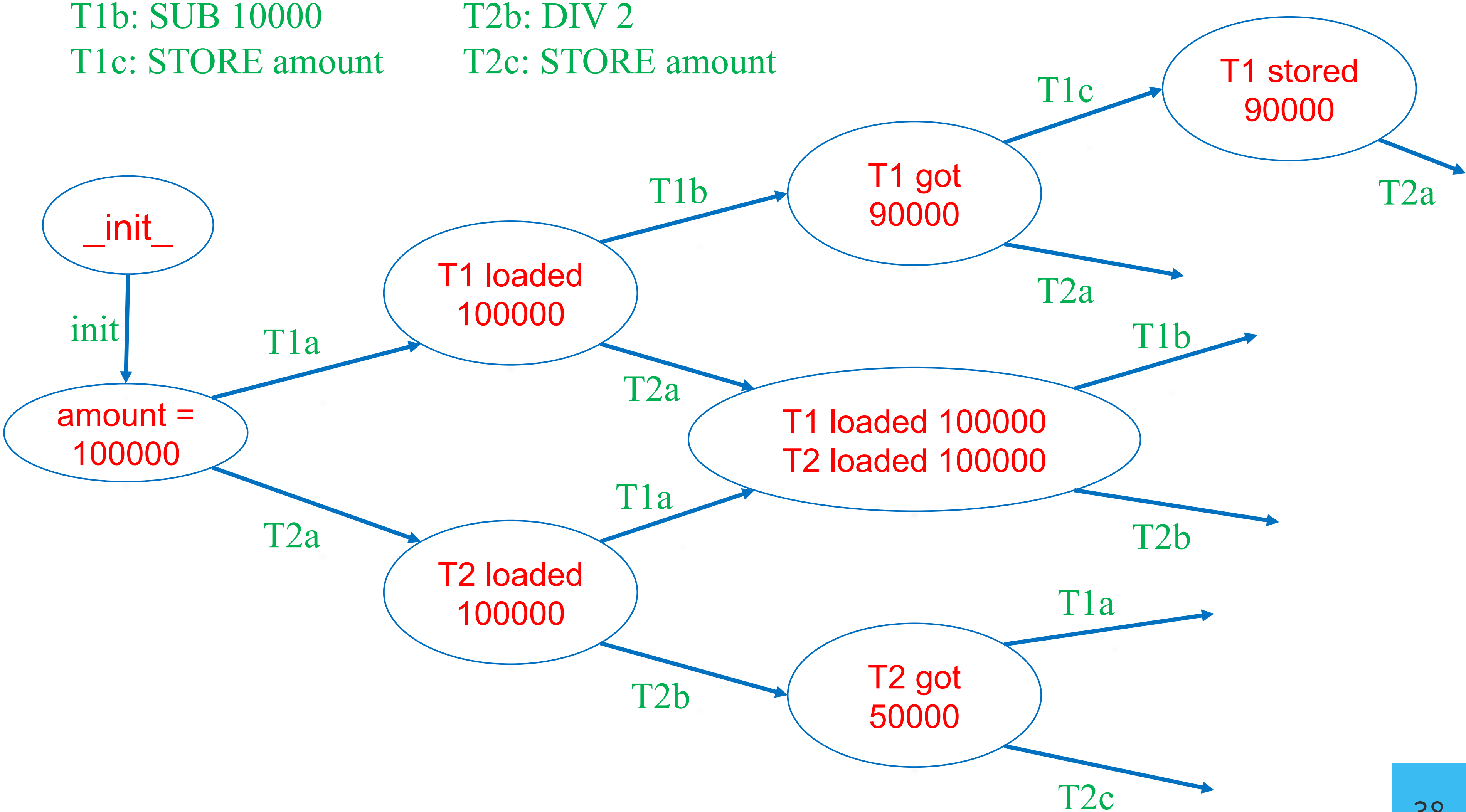
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Harmony Output

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]                 5  { amount: 100000, done1: False, done2: False }
T2/() [10-17].            17 { amount: 50000,   done1: False, done2: True   }
T1/() [5-8]                  8 { amount: 90000,   done1: True,  done2: True   }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

#states in the state graph

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]             5  { amount: 100000, done1: False, done2: False }
T2/() [10-17].         17 { amount: 50000,   done1: False, done2: True  }
T1/() [5-8]             8 { amount: 90000,   done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

something went wrong in (at least) one path in the graph (*assertion failure*)

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

shortest path to
assertion failure

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]              5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]           17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]              8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

turns

Output

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]              5  { amount: 100000, done1: False, done2: False }
T2/() [10-17]           17  { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]              8  { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

init

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]               5  { amount: 100000, done1: False, done2: False }
        T2/() [10-17]            17 { amount: 50000,   done1: False, done2: True  }
        T1/() [5-8]               8 { amount: 90000,   done1: True,  done2: True  }
        main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2abc   T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
        T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
        main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2abc   T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1c     T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
        main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]               5 { amount: 100000, done1: False, done2: False }
T2abc   T2/() [10-17]            17 { amount: 50000,  done1: False, done2: True  }
T1c     T1/() [5-8]               8 { amount: 90000,  done1: True,  done2: True  }
main    main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

	#states = 100 diameter = 5
	==== Safety violation ====
init	__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab	T1/() [1-4] 5 { amount: 100000, done1: False, done2: False }
T2abc	T2/() [10-17] 17 { amount: 50000, done1: False, done2: True }
T1c	T1/() [5-8] 8 { amount: 90000, done1: True, done2: True }
main	main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
	>>> Harmony Assertion (file=test.hny, line=11) failed: 90000

Output

name of a thread

__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4] 5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]. 17 { amount: 50000, done1: False, done2: True }
T1/() [5-8] 8 { amount: 90000, done1: True, done2: True }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }

Output

“steps” =
list of program counters
of machine instructions
executed

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]              5  { amount: 100000, done1: False, done2: False }
T2/() [10-17]           17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]              8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

Harmony Machine Code

0 Jump 40	
1 Frame T1 ()	
2 Load amount	T1a: LOAD amount
3 Push 10000	T1b: SUB 10000
4 2-ary –	
5 Store amount	T1c: STORE amount
6 Push True	T1d: done1 = True
7 Store done1	
8 Return	
9 Jump 40	
10 Frame T2 ()	
11 Load amount	T2a: LOAD amount
12 Push 2	T2b: DIV 2
13 2-ary /	
14 Store amount	T2c: STORE amount
15 Push True	T2d: done2 = True
16 Store done2	
17 Return	
18 …	

```
def T1():
    amount −= 10000
    done1 = True
```

```
def T2():
    amount /= 2
    done2 = True
```

Harmony Machine Code

```
0  Jump 40                PC := 40
1  Frame T1 ()
2  Load amount
3  Push 10000
4  2-ary –
5  Store amount
6  Push True
7  Store done1
8  Return
9  Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

```
0  Jump 40                     PC := 40
1  Frame T1 ()
2  Load amount                 push amount onto the stack of thread T1
3  Push 10000
4  2-ary –
5  Store amount
6  Push True
7  Store done1
8  Return
9  Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

0 Jump 40	PC := 40
1 Frame T1 ()	
2 Load amount	push amount onto the stack of thread T1
3 Push 10000	push 10000 onto the stack of thread T1
4 2-ary –	replace top two elements of stack with difference
5 Store amount	
6 Push True	
7 Store done1	
8 Return	
9 Jump 40	
10 Frame T2 ()	
11 Load amount	
12 Push 2	
13 2-ary /	
14 Store amount	
15 Push True	
16 Store done2	
17 Return	
18 …	

Harmony Machine Code

0 Jump 40 — PC := 40

1 Frame T1 ()
2 Load amount — push amount onto the stack of thread T1
3 Push 10000 — push 10000 onto the stack of thread T1
4 2-ary – — replace top two elements of stack with difference
5 Store amount — store top of the stack of T1 into amount
6 Push True
7 Store done1
8 Return

9 Jump 40

10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return

18 …

Harmony Machine Code

```
 0 Jump 40            PC := 40
 1 Frame T1 ()
 2 Load amount        push amount onto the stack of process T1
 3 Push 10000         push 10000 onto the stack of process T1
 4 2-ary –            replace top two elements of stack with difference
 5 Store amount       store top of the stack of T1 into amount
 6 Push True          push True onto the stack of thread T1
 7 Store done1        store top of the stack of T1 into done1
 8 Return
 9 Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

0 Jump 40	PC := 40
1 Frame T1 ()	
2 Load amount	push amount onto the stack of process T1
3 Push 10000	push 10000 onto the stack of process T1
4 2-ary –	replace top two elements of stack with difference
5 Store amount	store top of the stack of T1 into amount
6 Push True	push True onto the stack of thread T1
7 Store done1	store top of the stack of T1 into done1
8 Return	
9 Jump 40	
10 Frame T2 ()	
11 Load amount	push amount onto the stack of thread T2
12 Push 2	push 2 onto the stack of thread T2
13 2-ary /	replace top two elements of stack with division
14 Store amount	store top of the stack of T2 into amount
15 Push True	push True onto the stack of thread T2
16 Store done2	store top of the stack of T2 into done2
17 Return	
18 …	

Output

current program counter
(after turn)

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

Output

current state
(after turn)

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5  { amount: 100000, done1: False, done2: False }
T2/() [10-17]          17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8  { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

Harmony Virtual Machine *State*

Three parts:

1. code (which never changes)
2. values of the shared variables
3. states of each of the running processes
   – “contexts”

State represents one vertex in the graph model

Context (state of a process)

- Method name and parameters
- PC (program counter)
- stack (+ implicit stack pointer)
- local variables
  - parameters (aka arguments)
  - “result”
    - there is no **return** statement
  - local variables
    - declared in **let** and **for** statements

Harmony != Python

Harmony	Python
tries all possible executions	executes just one
(…) == […] == …	1 != [1] != (1)
1, == [1,] == (1,) != (1) == [1] == 1	[1,] == [1] != (1) == 1 != (1,)
f(1) == f 1 == f[1]	f 1 and f[1] are illegal (if f is method)
{ } is empty set	{ } is empty dictionary
few operator precedence rules --- use parentheses often	many operator precedence rules
variables global unless declared otherwise	depends... Sometimes must be explicitly declared global
no **return**, **break**, **continue**	various flow control escapes
no classes	object-oriented
…	…

I/O in Harmony?

- Input:
  - **choose** expression
    - x = **choose**({ 1, 2, 3 })
    - allows Harmony to know all possible inputs
  - **const** expression
    - **const** x = 3
    - can be overridden with "-c x=4" flag to harmony
  - Output:
    - **assert** x + y < 10
    - **assert** x + y < 10, (x, y)

I/O in Harmony?

- Input:
  - **choose** expression
    - x = **choose**({ 1, 2, 3 })
    - allows Harm ... s
  - **cons**
    - ... en with “-c x=4” flag to harmony
  - ... t:
    - **assert** x + y < 10
    - **assert** x + y < 10, (x, y)

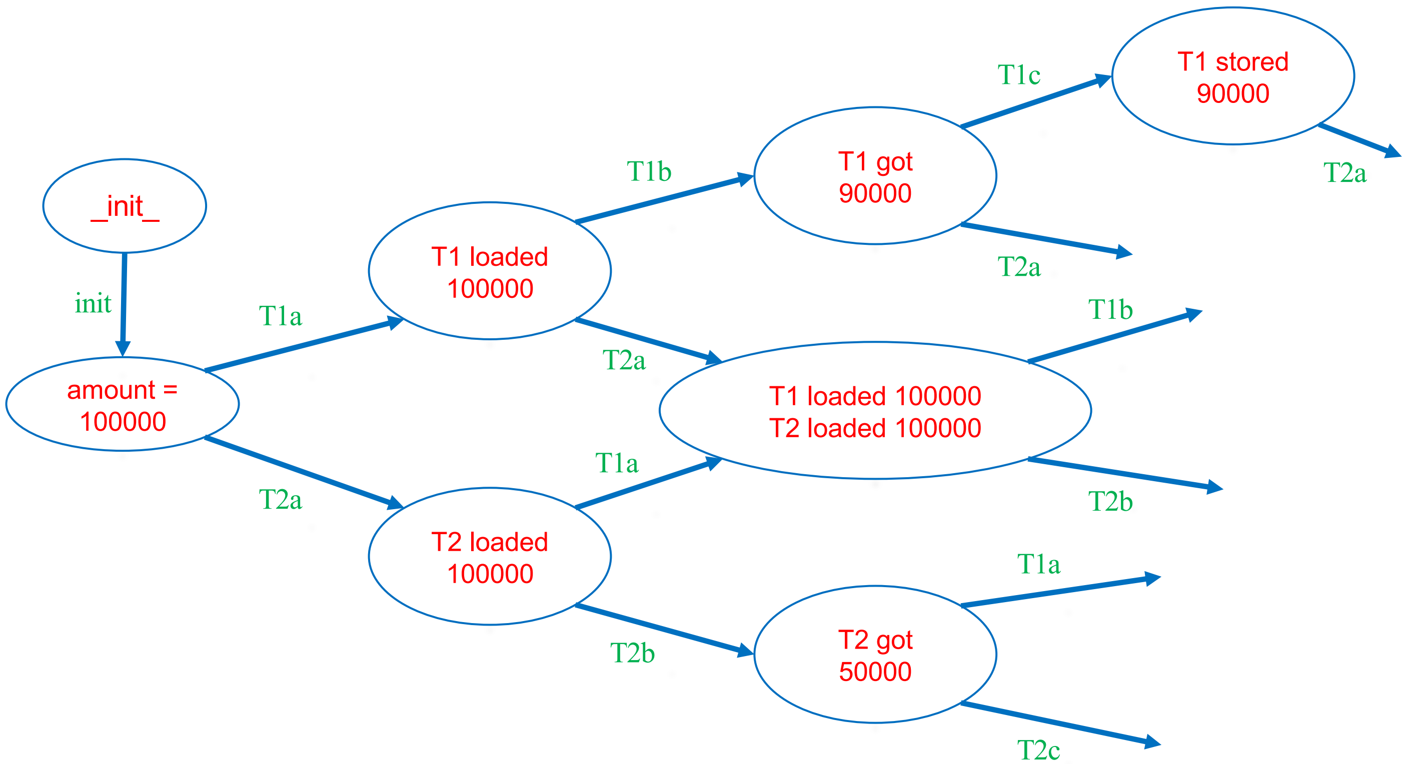
No open(), read(), input(), or print() statements

Non-determinism in Harmony

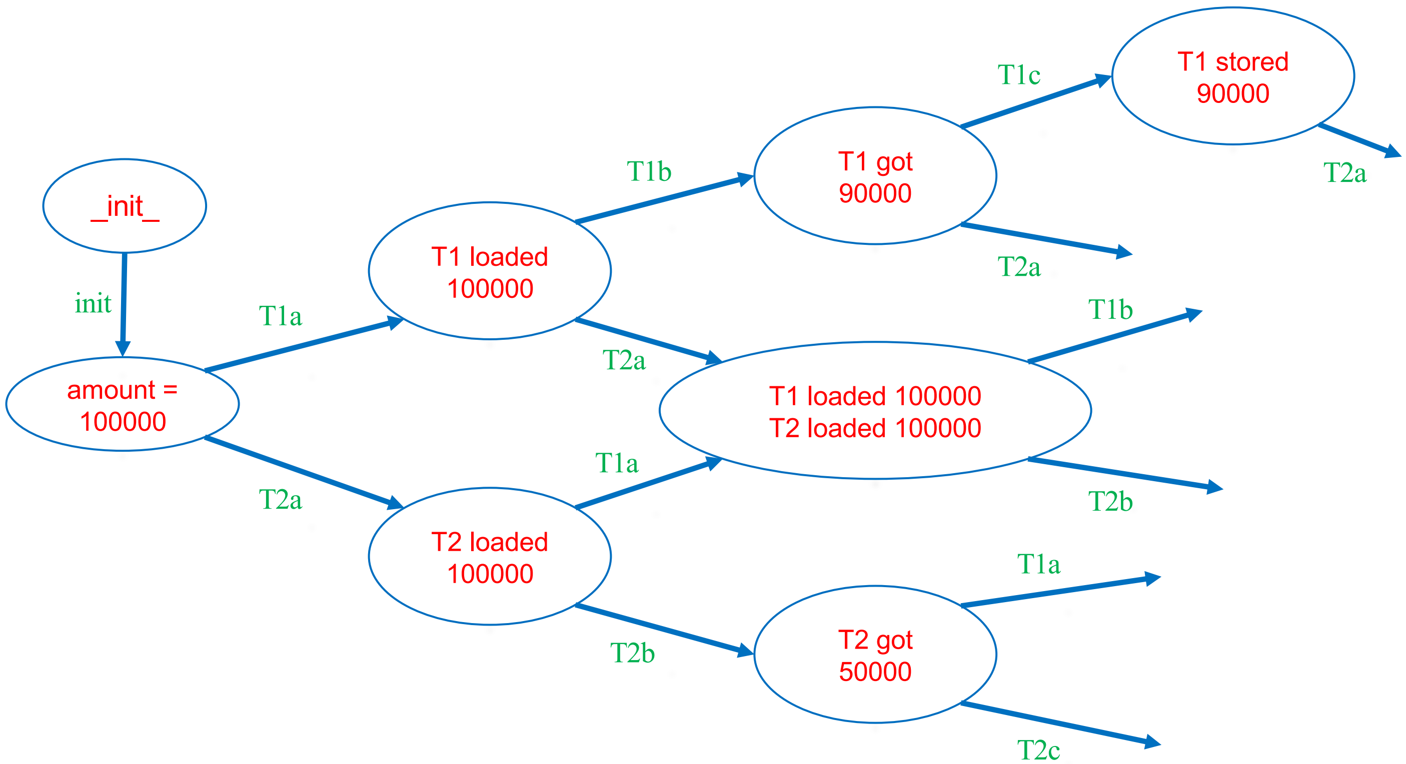
Three sources:

1. **choose** expressions
2. thread interleavings
3. Interrupts

Limitation: models must be finite!



Limitation: models must be finite!

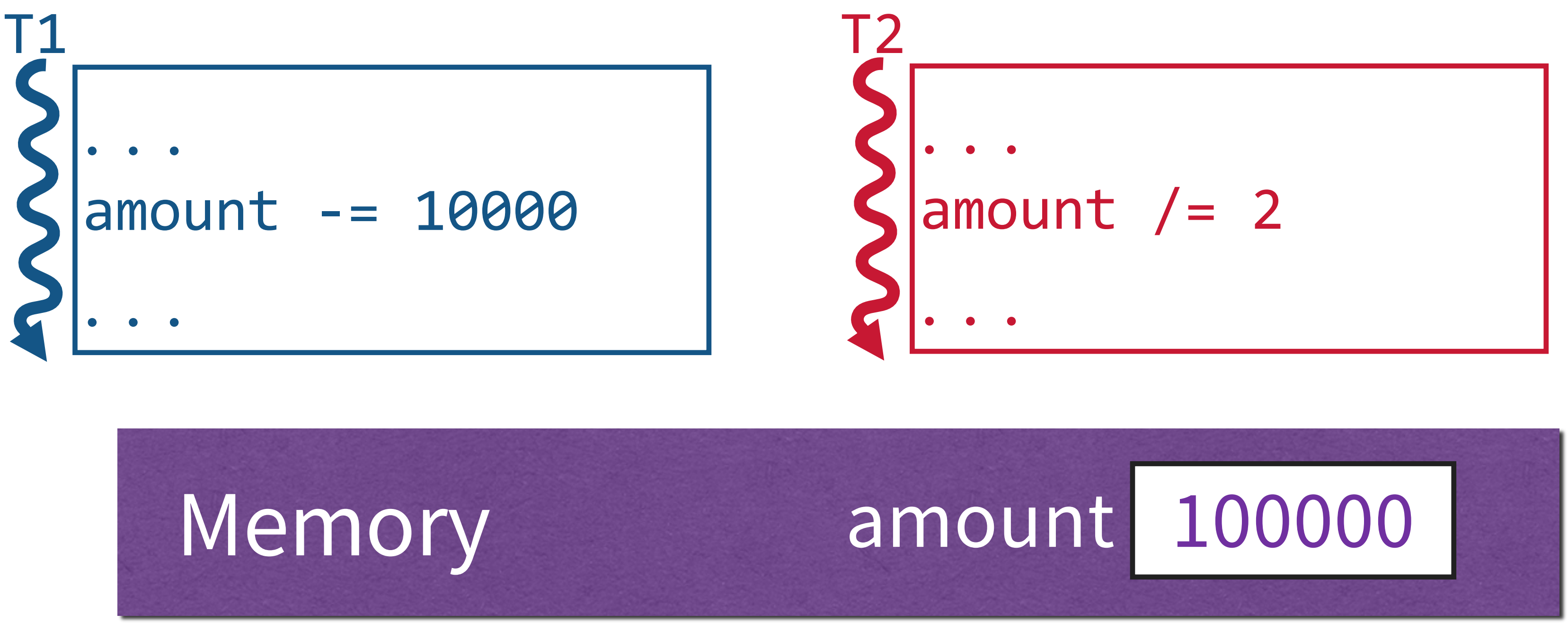


- But models are allowed to have cycles.
- Executions are allowed to be unbounded!
- Harmony checks for *possibility* of termination

Back to our problem…

2 threads updating a shared variable **amount**

- One thread wants to decrement amount by $10K
- Other thread wants to decrement amount by 50%



How to “serialize” these executions?

Critical Section

Must be serialized due to shared memory access

T1

```
. . .
CSEnter()
  amount -= 10000
CSExit()
. . .
```

T2

```
. . .
CSEnter()
  amount /= 2
CSExit()
. . .
```

<u>Goals</u>

Mutual Exclusion: 1 thread in a critical section at time

Progress: all threads make it into the CS if desired

Fairness: equal chances of getting into CS

… in practice, fairness rarely guaranteed

Critical Section

Must be serialized due to shared memory access

T1

```
. . .
CSEnter()
  Critical section
CSExit()
. . .
```

T2

```
. . .
CSEnter()
  Critical section
CSExit()
. . .
```

<u>Goals</u>

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:
  - either one is trivial to achieve by itself

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 thread(1)
spawn thread(2)
…
```

- 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
        @cs: assert atLabel(cs) == { (thread, self): 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

```
def thread(self):
    while choose( { False, True } ):
        …    # code outside critical section
        …    # code to enter the critical section
        @cs: assert atLabel(cs) == { (thread, self): 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:

How does one build a lock?

- Harmony is a concurrent programming language. *Really, doesn't Harmony have locks?*

You have to program them!

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

```
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)
```

wait till lock is free, then take it

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)
```

wait till lock is free, then take it

release the lock

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

First attempt: 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 atLabel(cs) == { (thread, self): 1 }
11
12          # Leave
13          lockTake
14
15  spawn thread(
16  spawn thread(
```

```
==== Safety violation ====
__init__/() [0,26-36]                          36 { lockTaken: False }
thread/0 [1-2,3(choose True),4-7]       8 { lockTaken: False }
thread/1 [1-2,3(choose True),4-8]       9 { lockTaken: True }
thread/0 [8-19]                                19 { lockTaken: True }
>>> Harmony Assertion (file=code/naiveLock.hny, line=10) failed
```

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

Second attempt: *flags*

```
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*

```
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)
```

enter, then wait for other

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

Second attempt: *flags*

```
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)
```

enter, then wait for other

leave

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

Second attempt: *flags*

```
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*

```
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/n

```
==== Non-terminating State ===
__init__/() [0,36-46]                       46 { flags: [False, False] }
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
```

Third attempt: *turn* variable

```
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

```
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)
```

after you...

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

Third attempt: *turn* variable

```
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)
```

after you...

wait for your turn

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

Third attempt: *turn* variable

```
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)
```

```
==== Non-terminating State ===
__init__/() [0,28-38]                                       38 { turn: 0 }
thread/0 [1-2,3(choose True),4-26,2,3(choose True),4]        5 { turn: 1 }
thread/1 [1-2,3(choose False),4,27]                         27 { turn: 1 }
blocked thread: thread/0 pc = 5
```

Figure 5.7: [co

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] Peterson's Algorithm

Peterson's Algorithm: *flags & turn*

latest version of Harmony only

```
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*

```
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                "you go first"
        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*

```
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

Figure 6.1: [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.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)
```

#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 might understand 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*

invariant = predicate that holds in every reachable state

What is an invariant?

A property that holds in all reachable states

(and possibly in some unreachable states as well)

What is a property?

A property is a set of states

often succinctly described using a predicate

(all states that satisfy the predicate and no others)

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"?
  - or how do we characterize a *reachable state*?
- Instead, it's much easier if we proved a so-called "inductive invariant":
  - Base Case:
    - show that IH holds in the initial state(s)
  - 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
  - if $T0$ is in the critical section, then $T1$ is not
    - without loss of generality…
  - 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
T1      await (not flags[1 − self]) or (turn == self)

        # critical section is here
T0      @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:

```
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)
```

T0

T1

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)

```
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)
```

T1

T0

flags = [True, True]
turn = 1

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Useful and obvious but insufficient invariant

```
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)
```

$$Tx@cs \Rightarrow flags[x]$$

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

What else do we expect to hold @cs?

```
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:

$$T0@cs \implies \neg flags[1] \vee turn = 0$$

- Promising because if $T0@cs \wedge T1@cs$ then

$$\left.\begin{array}{r} T0@cs \implies \neg flags[1] \vee turn = 0 \wedge \\ T1@cs \implies \neg flags[0] \vee turn = 1 \end{array}\right\} \Rightarrow \left\{\begin{array}{c} turn = 0 \ \wedge \\ turn = 1 \end{array}\right.$$

$\implies$ False (therefore mutual exclusion)

- Unfortunately, this is not an invariant…

Another obvious IH to try

- Based on the **await** condition:

$$T0@cs \implies \neg flags[1] \lor turn = 0$$

Easy to check with Harmony

Just run it with the following:

@cs: assert (not flags[1 - self]) or (turn == self)

- Unfortunately, this is not an invariant…

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 s
        flags[self] = False

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

T1

T0

flags = [True, False]
turn = 1

$T0@cs \implies \neg flags[1] \vee turn = 0$ holds

note: this is a reachable state

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

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 s
        flags[self] = False

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

T1

T0

flags = [True, True]
turn = 1

$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0$ violated

note: this is also a reachable state

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

But suggests an improved hypothesis

$$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \vee T1@gate$$

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

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
T1      @gate: turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # Critical section
T0      @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

$$T0@cs \implies \neg flags[1] \lor turn = 0 \lor T1@gate$$

```
flags = [ False, False ]
```

Also easy to check with Harmony

Proves that it is invariant, but not necessarily an *inductive invariant*

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

        # Leave critical section
        flags[self] = False
```

Inductive Invariance Proof

Let I be the induction hypothesis:

$$I \triangleq T0@cs \Longrightarrow \neg flags[1] \vee turn == 0 \vee T1@gate$$

I clearly holds in the initial state because $\neg T0@cs$ (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 $T0$ or $T1$ takes a step

Tricky Case 1:

$\neg T0@cs$ and $T0$ takes a step so that $T0@cs$

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

```
11        await (not flags[1 − self]) or (turn == self)
12
13        # critical section is here
14        @cs: assert atLabel(cs) == { (thread, self): 1 }
```

But then $\neg flags[1] \vee turn = 0$ still holds after the step

So $T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \vee T1@gate$

Tricky Case 2:

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

This must mean that before the step

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

So 3 cases to consider:

$\#$ *Enter critical section*
$flags[self] = \texttt{True}$
@gate: $turn = 1 - self$

- $\neg flags[1] \Rightarrow flags[1]$
  ➔ 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 $T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \vee T1@gate$ ✔

Finally, prove mutual exclusion

$$T0@cs \wedge T1@cs \Longrightarrow$$

$$\begin{cases} \neg flags[1] \vee turn = 0 \vee T1@gate \\ \neg flags[0] \vee turn = 1 \vee T0@gate \end{cases} \wedge$$

$$\Longrightarrow turn = 0 \wedge \mathrm{turn} = 1$$

$$\Longrightarrow False$$

Finally, prove mutual exclusion

$$T0@cs \wedge T1@cs \Longrightarrow$$

$$\begin{cases} \neg flags[1] \vee turn = 0 \vee T1@gate \\ \neg flags[0] \vee turn = 1 \vee T0@gate \end{cases} \wedge$$

$$\Longrightarrow turn = 0 \wedge \mathrm{turn} = 1$$

$$\Longrightarrow False$$

QED

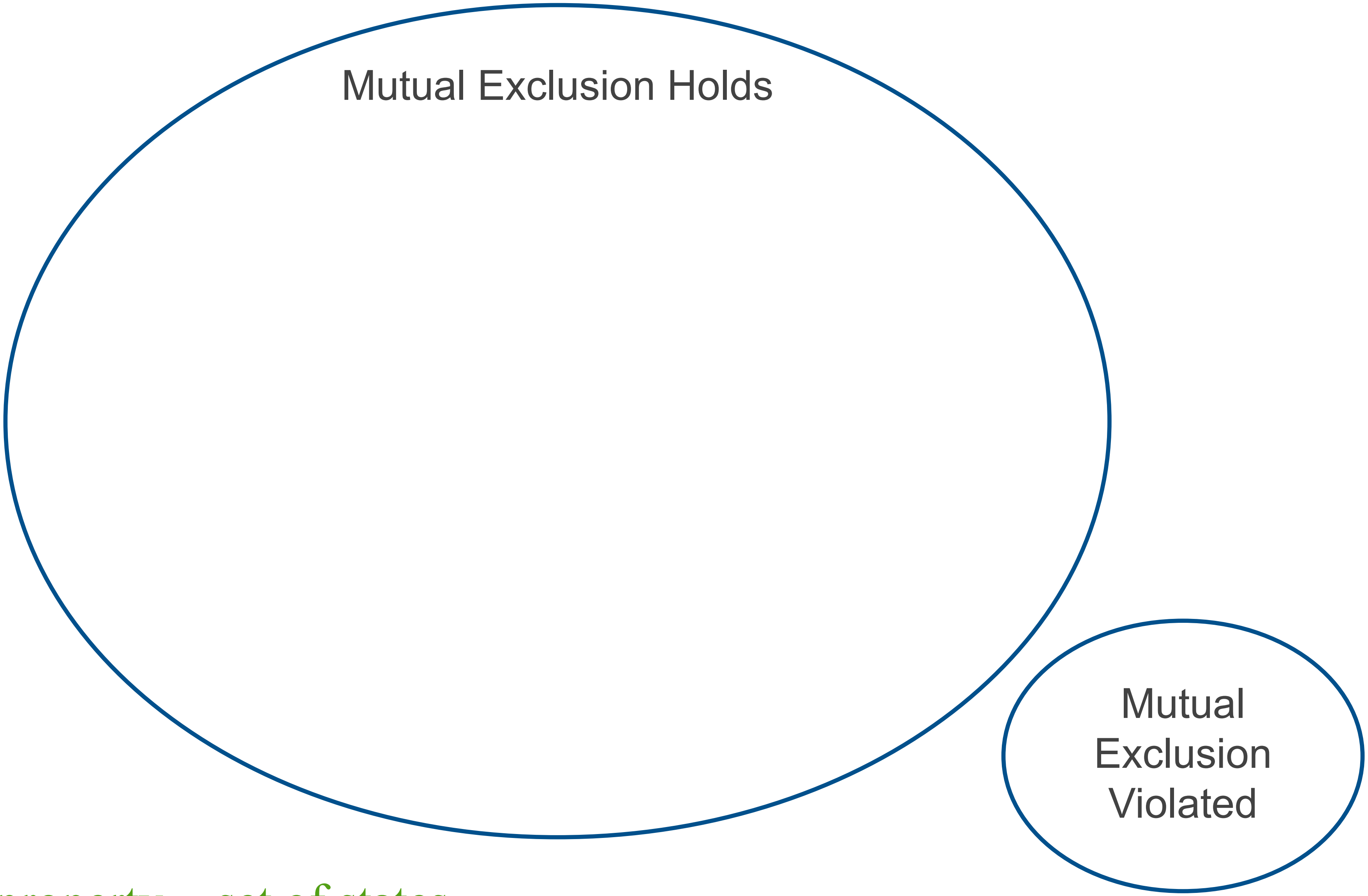
Now we can see why this state cannot be reached!

```
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
T1 ⇒    await (not flags[1 − self]) or (turn == self)

        # critical section is here
T0 ⇒    @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

flags = [True, True]
turn = 1

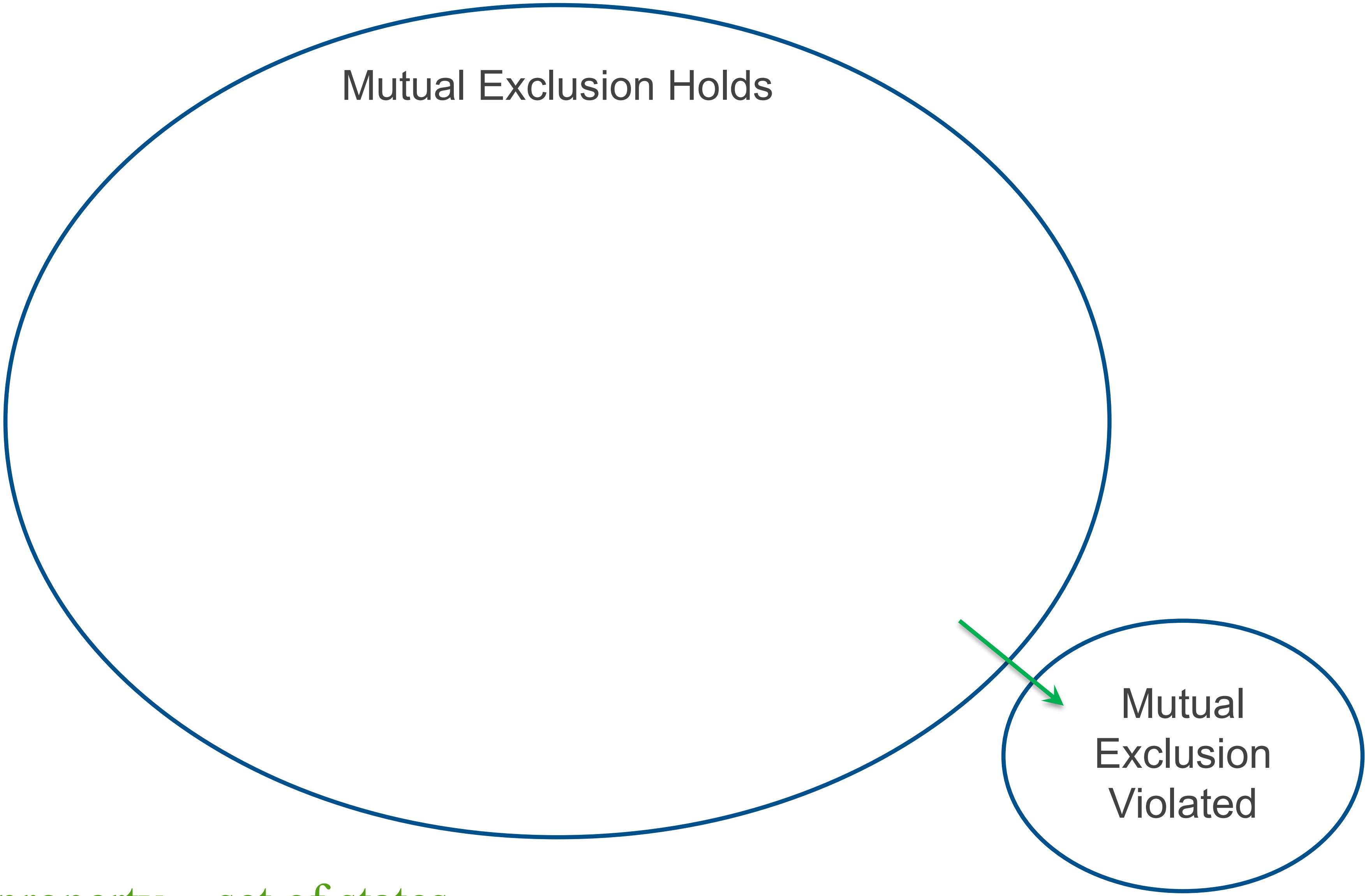
$$T0@cs \Longrightarrow \neg flags[1] \lor turn = 0 \lor T1@gate \quad \times$$

Review in Pictures: State Space



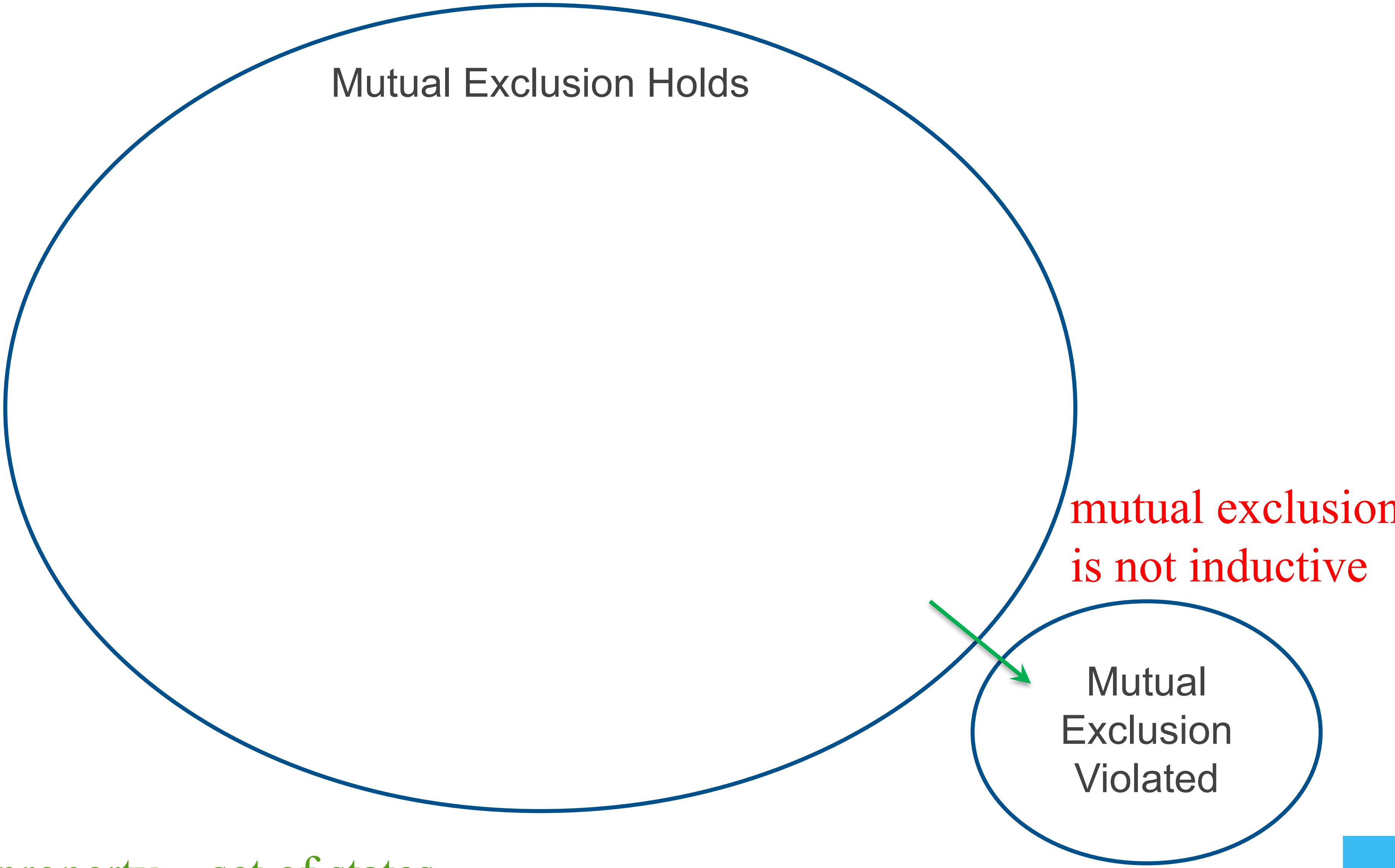
property = set of states

Review in Pictures: State Space



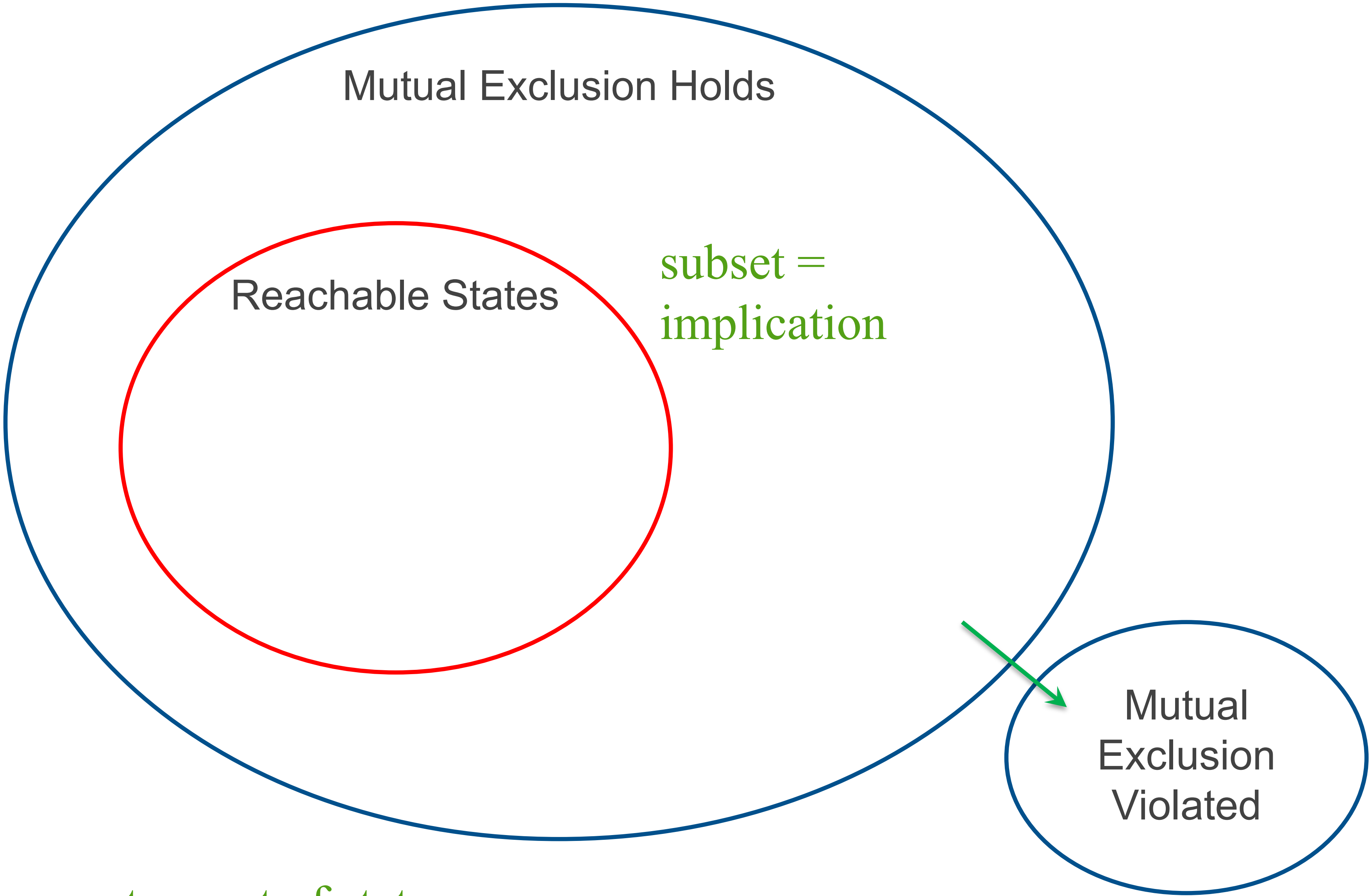
property = set of states

Review in Pictures: State Space



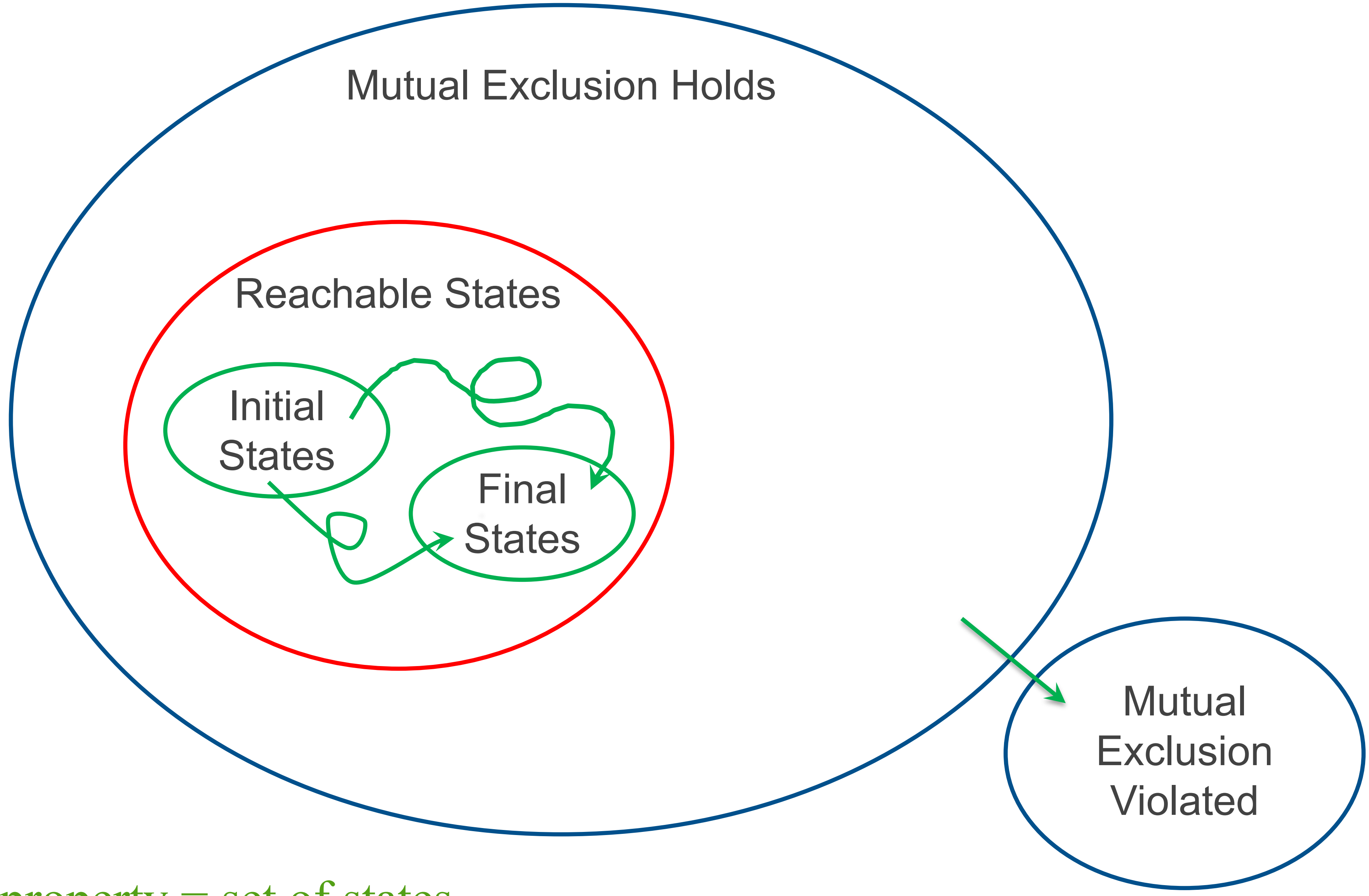
property = set of states

Review in Pictures: State Space



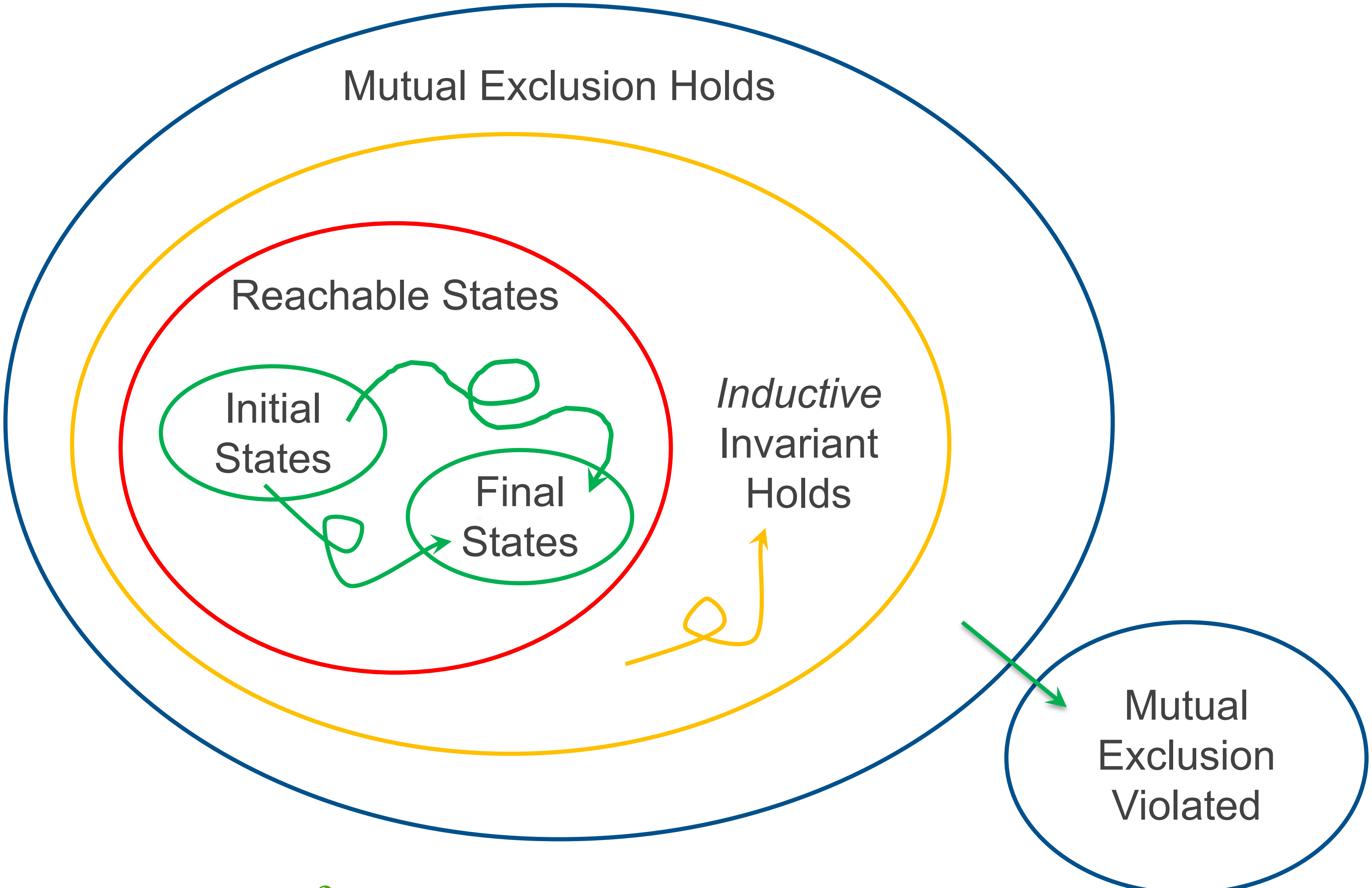
property = set of states

Review in Pictures: State Space



property = set of states

Review in Pictures: State Space



property = set of states

Peterson’s Reconsidered

- Mutual Exclusion can be implemented with atomic LOAD and STORE instructions to access shared memory
  - multiple STOREs and LOADs
- Peterson’s can be generalized to >2 processes
  - even more STOREs and LOADs

Too inefficient in practice

Peterson's Reconsidered *More*

- Assumes that LOAD and STORE instructions are *atomic*
- This is a good assumption in Harmony
- But not on a real processor
- Also not guaranteed by C, Java, Python, …

So Peterson's doesn't even work on modern machines

Peterson's Reconsidered More

- Assumes that LO
  instructions are
- This is a good as ny
- But not on a rea
- Also not guarant on,
  …

So Peterson's does dern machines

Data Race

- When two threads access the same variable
- And at least one is a STORE
- Then the semantics of the outcome is *undefined*

For example

- Shared variable *x* contains 3
- Thread A stores 4 into *x*
- Thread B loads *x*
  - With atomic load/store operations, B will read either 3 or 4
  - With modern CPUs, the value that B reads is undefined

Enter *Interlock Instructions*

(ECE people got us into this mess; this is how they get us out)

- Machine instructions that do multiple shared memory accesses atomically

- E.g., TestAndSet *s*, *p*
  - sets *p* to the (old) value of *s*
  - sets *s* to True
    - i.e., LOAD *s*, STORE *p*, STORE *s*

- Entire operation is *atomic*
  - other machine instructions cannot interleave

Enter *Interlock Instructions*

(ECE people got us into this mess; this is how they get us out)

- Machine instructions that do multiple shared memory accesses atomically

- E.g., TestAndSet *s*, *p*
  - sets *p* to the (old) value of *s*
  - sets *s* to True
    - i.e., LOAD *s*, STORE *p*, STORE *s*

```
def tas(s, p):
    atomic:
        !p = !s
        !s = True
```

- Entire operation is *atomic*
  - other machine instructions cannot interleave

Harmony interlude: *pointers*

- If x is a shared variable, ?x is the address of x
- If p is a shared variable and p == ?x, then we say that p is a *pointer* to x
- Finally, !p refers to the value of x

Harmony interlude: *pointers*

- If x is a shared variable, ?x is the address of x
- If p is a shared variable and p == ?x, then we say that p is a *pointer* to x
- Finally, !p refers to the value of x

Where?
There!

Harmony interlude: *pointers*

- If x is a shared variable, ?x is the address of x
- If p is a shared variable and p == ?x, then we say that p is a *pointer* to x
- Finally, !p refers to the value of x

$$
\begin{array}{l}
\texttt{def tas}(s,\ p)\texttt{:} \\
\quad \texttt{atomic:} \\
\qquad !p = !s \\
\qquad !s = \texttt{True}
\end{array}
$$

s and *p* are pointers, thus tas(*s*, *p*) can be used with any two shared variables:

tas(?*x*, ?*y*) or tas(?*q*, ?*r*)

Critical Sections with TAS

```
const N = 3

shared = False
private = [ True, ] * N
def thread(self):
    while choose({ False, True }):
        # Enter critical section
        while private[self]:
            tas(?shared, ?private[self])

        # Critical section
        @cs: assert (not private[self]) and (atLabel(cs) == { (thread, self): 1 })

        # Leave critical section
        private[self] = True
        shared = False

for i in {0..N−1}:
    spawn thread(i)
```

number of processes

private[*i*] belongs to process(*i*)

Figure 8.1: [code/spinlock.hny] Mutual Exclusion using a "spinlock" based on test-and-set.

Critical Sections with TAS

```
const N = 3
```

number of processes

```
shared = False
private = [ True, ] * N
```

private[*i*] belongs to process(*i*)

```
def thread(self):
    while choose({ False, True }):
        # Enter critical section
        while private[self]:
            tas(?shared, ?private[self])
```

"spinlock"

```
        # Critical section
        @cs: assert (not private[self]) and (atLabel(cs) == { (thread, self): 1 })

        # Leave critical section
        private[self] = True
        shared = False

for i in {0..N−1}:
    spawn thread(i)
```

Figure 8.1: [code/spinlock.hny] Mutual Exclusion using a "spinlock" based on test-and-set.

Critical Sections with TAS

```
const N = 3

shared = False
private = [ True, ] * N
def thread(self):
    while choose({ False, True }):
        # Enter critical section
        while private[self]:
            tas(?shared, ?private[self])

        # Critical section
        @cs: assert (not private[self]) and (atLabel(cs) == { (thread, self): 1 })

        # Leave critical section
        private[self] = True
        shared = False

for i in {0..N−1}:
    spawn thread(i)
```

number of processes

private[*i*] belongs to process(*i*)

“spinlock”

$\text{thread}(\mathit{self})@\mathit{cs} \Longrightarrow \neg \mathit{private}[\mathit{self}]$

Figure 8.1: [code/spinlock.hny] Mutual Exclusion using a “spinlock” based on test-and-set.

Two essential invariants

1. $\forall i: thread(i)@cs \Rightarrow \neg private[i]$
2. at most 1 of $shared$ and $private[i]$ is False

1. Obvious
2. Easy proof by induction

   both can also be checked by Harmony

Two essential invariants

1. $\forall i\colon thread(i)@cs \Rightarrow \neg private[i]$
2. at most 1 of $shared$ and $private[i]$ is False

1. Obvious
2. Easy proof by induction

both can also be checked by Harmony

$\texttt{invariant len}(x\ \texttt{for}\ x\ \texttt{in}\ [shared,] + private\ \texttt{where not}\ x) <= 1$

Two essential invariants

1. $\forall i: thread(i)@cs \Rightarrow \neg private[i]$
2. at most 1 of $shared$ and $private[i]$ is False

1. Obvious
2. Easy proof by induction

both can also be checked by Harmony

$\texttt{invariant len}(x \texttt{ for } x \texttt{ in } [shared,] + private \texttt{ where not } x) <= 1$

If at most one $private[i]$ can be False, then at most one $thread(i)$ can be $@cs$

“Locks”

Best understood as “baton passing”

- At most one thread, or *shared*, can “hold” False

Locks in the “synch” module

```
def tas(lk):
    atomic:
        result = !lk
        !lk = True

def BinSema(acquired):
    result = acquired

def Lock():
    result = BinSema(False)

def acquire(binsema):
    await not tas(binsema)

def release(binsema):
    assert binsema
    !binsema = False

def held(binsema):
    result = !binsema
```

Observation: *private*[*i*] does not need to be a shared variable. Just return the old value

Figure 9.2: [modules/synch.hny] The binary semaphore interface and implementation in the **synch** module.

“Ghost” state

- No longer have *private*[*i*]
- Instead:
  - We say that a lock is *held* or *owned* by a thread
- The invariants become:

1. $T@cs \Rightarrow T$ holds the lock
2. at most one thread can hold the lock

"Ghost" state

- No longer have *private*[*i*]
- Instead:
  - We say that a lock is *held* or *owned* by a thread
- The invariants become:
  1. $T@cs \Rightarrow T$ holds the lock
  2. at most one thread can hold the lock

(Harmony, like other systems, does not keep track of which thread holds which lock)

Using locks from the sync module

```
from synch import Lock, acquire, release

sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def thread(self):
    acquire(?countlock)
    count = count + 1
    release(?countlock)
    done[self] = True
    await done[1 − self]
    assert count == 2

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

import the sync module

Figure 9.3: [code/UpLock.hny] Program of Figure 3.2 fixed with a lock.

Using locks from the sync module

```
from synch import Lock, acquire, release

sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def thread(self):
    acquire(?countlock)
    count = count + 1
    release(?countlock)
    done[self] = True
    await done[1 - self]
    assert count == 2

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

initialize lock

Figure 9.3: [code/UpLock.hny] Program of Figure 3.2 fixed with a lock.

Using locks from the sync module

```
from synch import Lock, acquire, release

sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def thread(self):
    acquire(?countlock)
    count = count + 1
    release(?countlock)
    done[self] = True
    await done[1 − self]
    assert count == 2

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

enter critical section

exit critical section

Figure 9.3: [code/UpLock.hny] Program of Figure 3.2 fixed with a lock.

Using locks from the sync module

```
from synch import Lock, acquire, release

sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def thread(self):
    acquire(?countlock)
    count = count + 1
    release(?countlock)
    done[self] = True
    await done[1 - self]
    assert count == 2

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

?*countlock* is the address of *countlock*

thread *self* holds *countlock*

Figure 9.3: [code/UpLock.hny] Program of Figure 3.2 fixed with a lock.

Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But how about when it involves two threads on the same core:
    - when there is no pre-emption?
    - when there is pre-emption?

Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But how about when it involves two threads on the same core:
  - when there is no pre-emption?
    - can cause all threads to get stuck while one is trying to obtain a lock spinlock
  - when there is pre-emption?

Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But how about when it involves two threads on the same core:
  - when there is no pre-emption?
    - can cause all threads to get stuck while one is trying to obtain a lock spinlock
  - when there is pre-emption?
    - can cause delays and waste of CPU cycles while a thread is trying to obtain a spinlock

Context switching in Harmony

- Harmony allows contexts to be saved and restored (i.e., context switch)
  - *r* = **stop** *v*
    - stops the current thread and stores context in *v*
  - **go** *context r*
    - adds a thread with the given context to the bag of threads. Thread resumes from **stop** expression, returning *r*

Locks using **stop** and **go**

```
def BinSema(acquired):
    result = { .acquired: acquired, .suspended: [ ] }

def Lock():
    result = BinSema(False)

def acquire(binsema):
    atomic:
        if binsema→acquired:
            stop binsema→suspended[len binsema→suspended]
            assert binsema→acquired
        else:
            binsema→acquired = True

def release(binsema):
    atomic:
        assert binsema→acquired
        if binsema→suspended == [ ]:
            binsema→acquired = False
        else:
            go (list.head(binsema→suspended)) ()
            binsema→suspended = list.tail(binsema→suspended)
```

.acquired: boolean
.suspended: queue of contexts

Locks using **stop** and **go**

```
def BinSema(acquired):
    result = { .acquired: acquired, .suspended: [ ] }

def Lock():
    result = BinSema(False)

def acquire(binsema):
    atomic:
        if binsema→acquired:
            stop binsema→suspended[len binsema→suspended]
            assert binsema→acquired
        else:
            binsema→acquired = True

def release(binsema):
    atomic:
        assert binsema→acquired
        if binsema→suspended == [ ]:
            binsema→acquired = False
        else:
            go (list.head(binsema→suspended)) ()
            binsema→suspended = list.tail(binsema→suspended)
```

binsema→acquired is short for
(!*binsema*).acquired (cf. C, C++)

Locks using **stop** and **go**

```
def BinSema(acquired):
    result = { .acquired: acquired, .suspended: [ ] }

def Lock():
    result = BinSema(False)
```

Similar to a Linux "futex": if there is no contention (hopefully the common case) acquire() and release() are cheap. If there is contention, they involve a context switch.

```

def release(binsema):
    atomic:
        assert binsema→acquired
        if binsema→suspended == [ ]:
            binsema→acquired = False
        else:
            go (list.head(binsema→suspended)) ()
            binsema→suspended = list.tail(binsema→suspended)
```

Choosing modules in Harmony

- "synch" is the (default) module that has the TAS version of lock
- "synchS" is the module that has the **stop**/**go** version of lock
- you can select which one you want:

harmony -m synch=synchS x.hny

- "synch" tends to be faster than "synchS"
  - smaller state graph

Atomic section ≠ Critical Section

Atomic Section	Critical Section
only one thread can execute	multiple threads can execute concurrently, just not within a critical section
rare programming language paradigm	ubiquitous: locks available in many mainstream programming languages
good for implementing interlock instructions	good for building concurrent data structures

Building a Concurrent Queue

- *q* = queue.new(): allocate a new queue
- queue.put(*q*, *v*): add *v* to the tail of queue *q*
- *v* = queue.get(*q*): returns None if *q* is empty or *v* if *v* was at the head of the queue

Queue Test Program Example

```
import queue

def sender(q, v):
    queue.put(q, v)

def receiver(q):
    let done = False:
        while not done:
            let v = queue.get(q):
                done = v == None
                assert done or (v in { 1, 2 })

testq = queue.Queue()
spawn sender(?testq, 1)
spawn sender(?testq, 2)
spawn receiver(?testq)
spawn receiver(?testq)
```

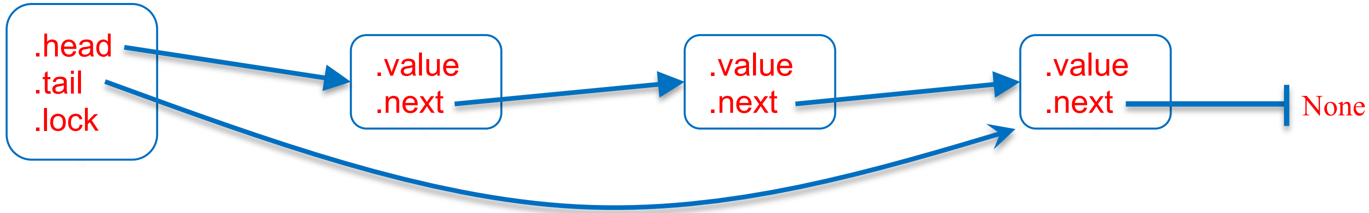
enqueue *v* onto *q*

dequeue until queue *q* is empty

create queue

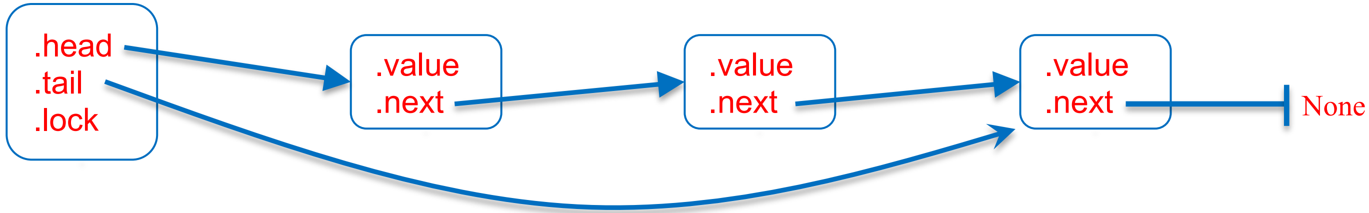
Figure 10.1: [code/queuetest.hny] Test program for a concurrent queue.

Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

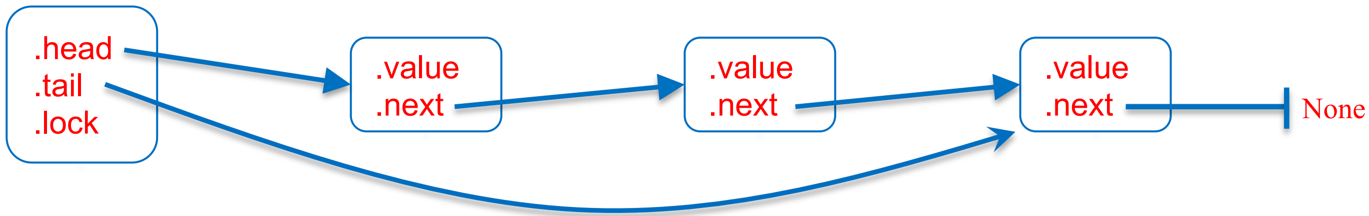
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

dynamic memory allocation

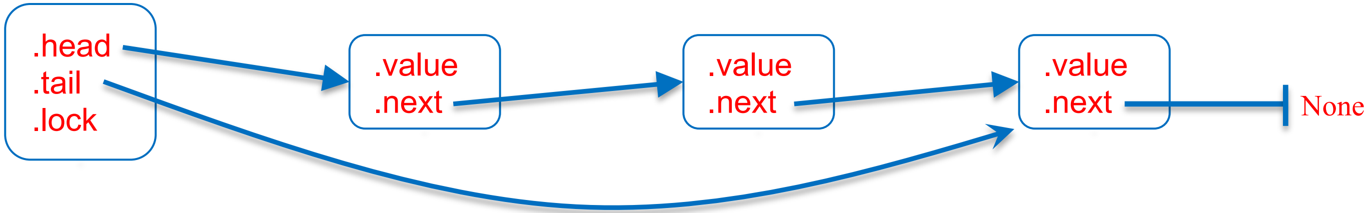
Queue implementation, v1



create empty queue

```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

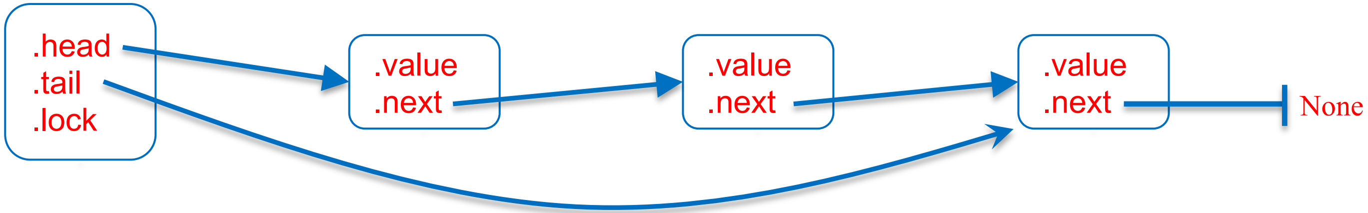
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

allocate node

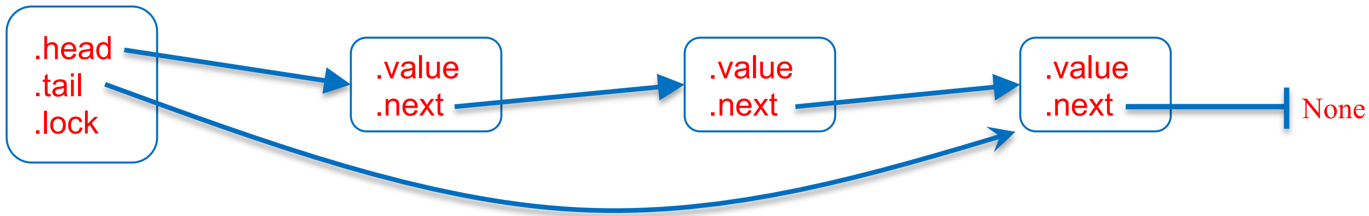
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

Queue implementation, v1

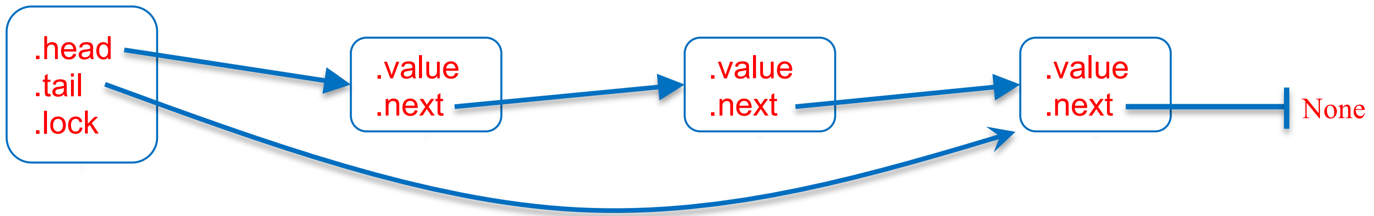



```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

the hard stuff

Queue implementation, v1



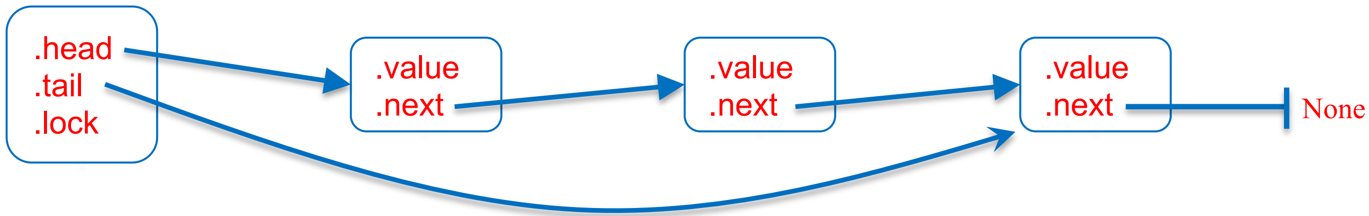

```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

the hard stuff

release lock

Queue implementation, v1




```
def get(q):
    acquire(?q→lock)
    let node = q→head:
        if node == None:
            result = None
        else:
            result = node→value
            q→head = node→next
            if q→head == None:
                q→tail = None
            free(node)
    release(?q→lock)
```

malloc'd memory must be explicitly released (cf. C)

Figure 10.2: [code/queue.hny]A basic concurrent queue data structure.

How important are concurrent queues?

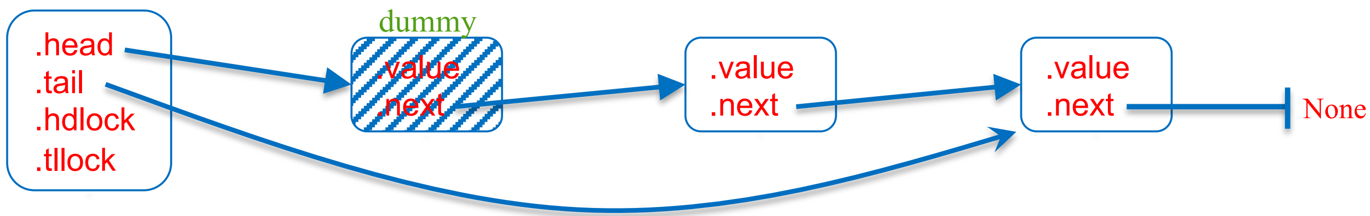
- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - ...

How important are concurrent queues?

- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

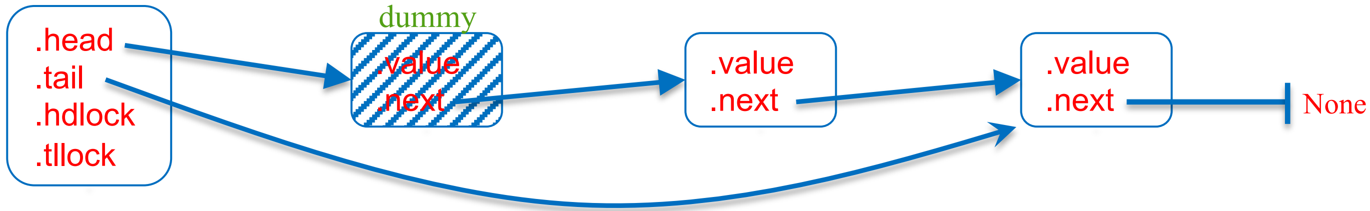
Good performance is critical!

Concurrent queue v2: 2 locks




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    let dummy = malloc({ .value: (), .next: None }):
        result = { .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→tllock)
        q→tail→next = node
        q→tail = node
        release(?q→tllock)
```

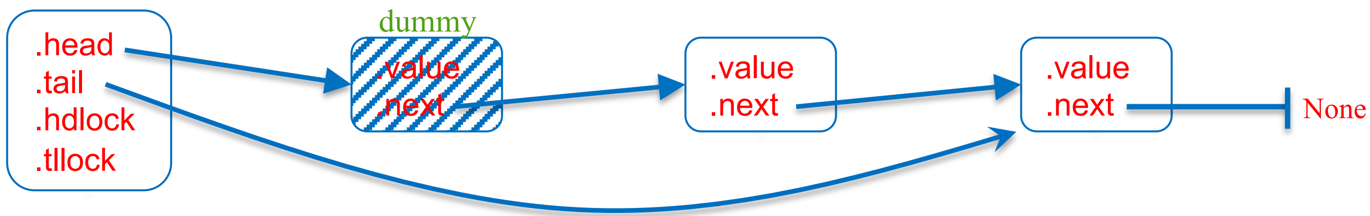
Concurrent queue v2: 2 locks




```
def get(q):
    acquire(?q→hdlock)
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
            free(dummy)
```

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.

Concurrent queue v2: 2 locks

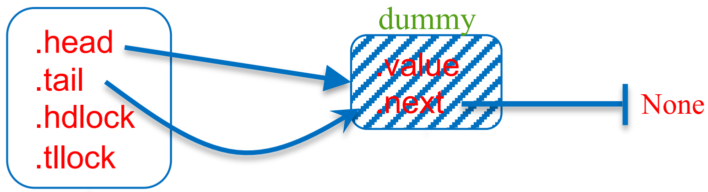



```
def get(q):
    acquire(?q→hdlock)
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
            free(dummy)
```

No contention for concurrent enqueue and dequeue operations!
➔ more concurrency ➔ faster

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.

Concurrent queue v2: 2 locks




```
def get(q):
    acquire(?q→hdlock)
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
            free(dummy)
```

But also incorrect for today's hardware because of a data race…

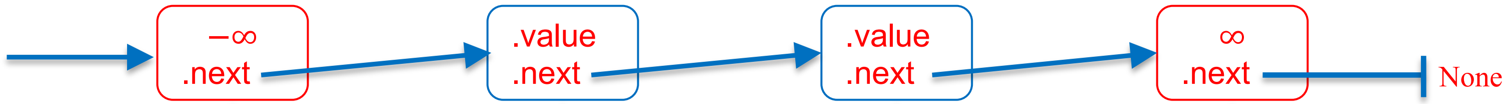
put and get concurrently access *dummy*→next when queue is empty

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.

Global vs. Local Locks

- The two-lock queue is an example of a data structure with *finer-grained locking*
- A global lock is easy, but limits concurrency
- Fine-grained or local locking can improve concurrency, but tends to be trickier to get right

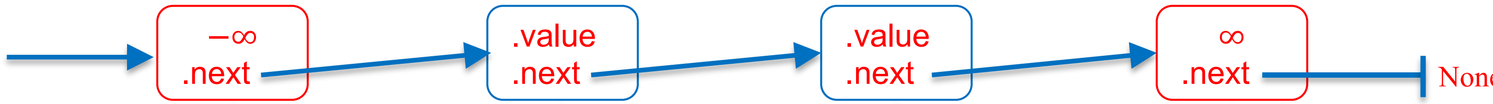
Sorted Integer Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):      # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    let before = lst:
        acquire(?before→lock)
        let after = before→next:
            acquire(?after→lock)
            while after→value < v:
                release(?before→lock)
                before = after
                after = before→next
                acquire(?after→lock)
            result = (before, after)

def LinkedList():
    result = _node(−inf, _node(inf, None))
```

create empty list

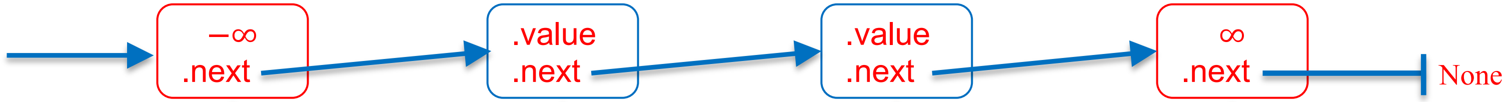
Sorted Integer Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):     # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    let before = lst:
        acquire(?before→lock)
        let after = before→next:
            acquire(?after→lock)
            while after→value < v:
                release(?before→lock)
                before = after
                after = before→next
                acquire(?after→lock)
            result = (before, after)

def LinkedList():
    result = _node(−inf, _node(inf, None))
```

Helper routine to find and lock two consecutive nodes *before* and *after* such that $before \rightarrow value < v \leq after \rightarrow value$

Sorted Integer Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):     # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    let before = lst:
        acquire(?before→lock)
        let after = before→next:
            acquire(?after→lock)
            while after→value < v:
                release(?before→lock)
                before = after
                after = before→next
                acquire(?after→lock)
            result = (before, after)

def LinkedList():
    result = _node(−inf, _node(inf, None))
```

Helper routine to find and lock two consecutive nodes *before* and *after* such that
$before \rightarrow value < v \leq after \rightarrow value$

Hand-over hand locking

(good for data structures without cycles)

Sorted Integer Linked List with Lock per Node

```
def insert(lst, v):
    let before, after = _find(lst, v):
        if after→value != v:
            before→next = _node(v, after)
        release(?after→lock)
        release(?before→lock)

def remove(lst, v):
    let before, after = _find(lst, v):
        if after→value == v:
            before→next = after→next
            release(?after→lock)
            free(after)
        else:
            release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
        result = after→value == v
        release(?after→lock)
        release(?before→lock)
```

Figure 10.4: [code/linkedlist.hny] Linked list with fine-grained locking.

Sorted Integer Linked List with Lock per Node

```
def insert(lst, v):
    let before, after = _find(lst, v):
        if after→value != v:
            before→next = _node(v, after)
        release(?after→lock)
        release(?before→lock)

def remove(lst, v):
    let before, after = _find(lst, v):
        if after→value == v:
            before→next = after→next
            release(?after→lock)
            free(after)
        else:
            release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
        result = after→value == v
        release(?after→lock)
        release(?before→lock)
```

Multiple threads can access the list simultaneously, but they can't *overtake* one another

Figure 10.4: [code/linkedlist.hny] Linked list with fine-grained locking.

How to get more concurrency?

Idea: allow multiple read-only operations to execute concurrently

- In many cases, reads are much more frequent than writes

➔ reader/writer lock

Either:

- multiple readers, or
- a single writer

thus not:

- *a reader and a writer, nor*
- *multiple writers*

Reader/writer lock interface and invariants:

- RW.read_acquire()
  - get a read lock. Multiple threads can have the read lock simultaneously, but no thread can have a write lock simultaneously
- RW.read_release()
  - release a read lock. Other threads may still have the read lock. When the last read lock is released, a write lock may be acquired
- RW.write_acquire()
  - acquire the write lock. Only one thread can have a write lock, and if so no thread can have a read lock
- RW.write_release()
  - release the write lock. Allows other threads to either get a read or write lock

R/W Locks: test for mutual exclusion

```
import RW

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            RW.read_acquire(?rw)
            @rcs: assert atLabel(wcs) == ()
            RW.read_release(?rw)
        else:                        # .write
            RW.write_acquire(?rw)
            @wcs: assert (atLabel(wcs) == { (thread, ()): 1 }) and
                        (atLabel(rcs) == ())
            RW.write_release(?rw)

for i in {1..3}:
    spawn thread()
```

no writer

1 writer and no readers

Figure 11.1: [code/RWtest.hny] Test code for reader/writer locks.

Conditional Waiting

Conditional Waiting

- So far we've shown how threads can wait for one another to avoid multiple threads in the critical section
- Sometimes there are other reasons:
  - Wait until queue is non-empty
  - Wait until there are no readers (or writers) in a reader/writer lock
  - …

Busy Waiting: not a good way

- Wait until queue is non-empty:

$$
\begin{aligned}
&\mathit{done} = \textbf{False}\\
&\textbf{while not}\ \mathit{done}:\\
&\qquad \mathit{next} = \text{queue.get}(q)\\
&\qquad \mathit{done} = \mathit{next}\ !\!= \textbf{None}
\end{aligned}
$$

Busy Waiting: not a good way

- Wait until queue is non-empty:

    done = **False**
    while not *done*:
        next = queue.get(*q*)
        done = *next* != **None**

- *wastes CPU cycles*
- *creates unnecessary contention*