Concurrent Programming in Harmony: Critical Sections and Locks

CS 4410
Operating Systems

[Robbert van Renesse]

An Operating System is a Concurrent Program

- The ”kernel contexts” of each of the processes share many data structures
- Further complicated by interrupt handlers that also access those data structures

So I talked with a recruiter last week…

- Not making this up…

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?
- How to construct correct concurrent programs?

Concurrent Programming is Hard

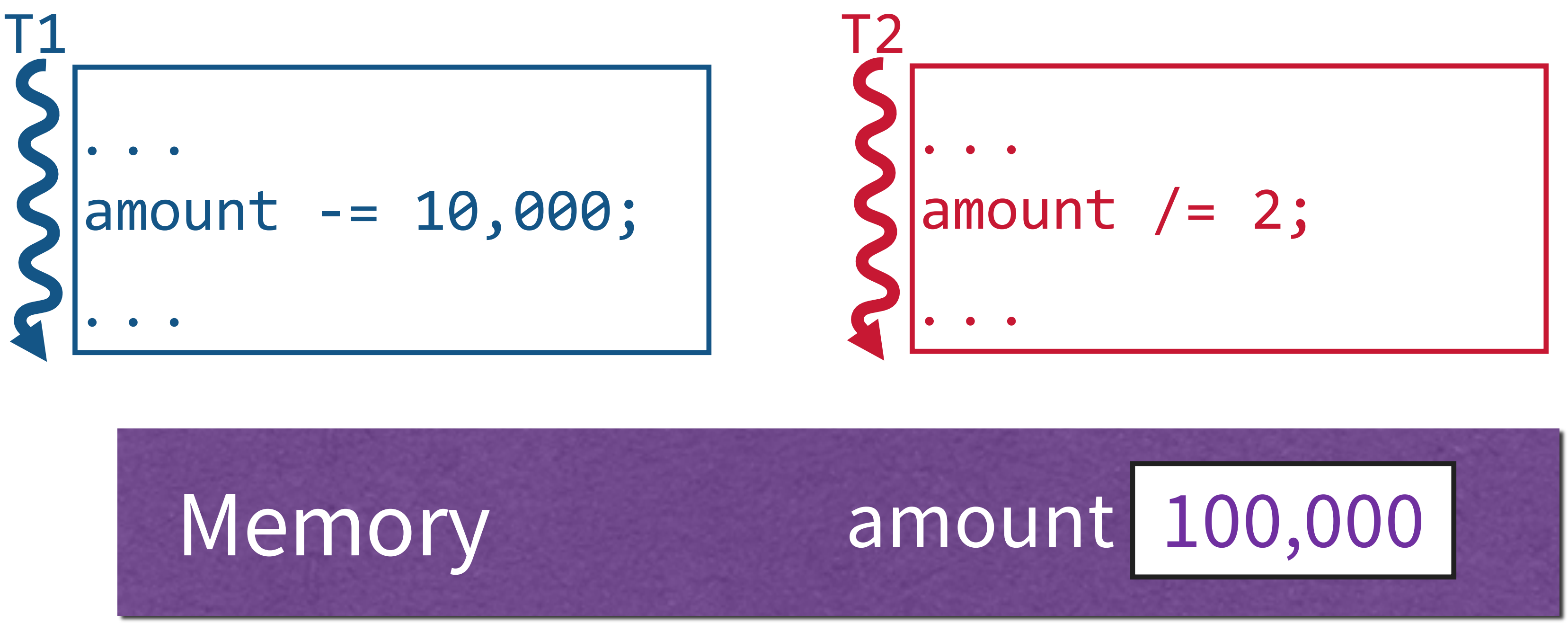
Why?

- 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**

Two Theads, One Variable

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%



What happens when both threads are running?

Two Theads, One Variable

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…

Two Theads, One Variable

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
. . .
```

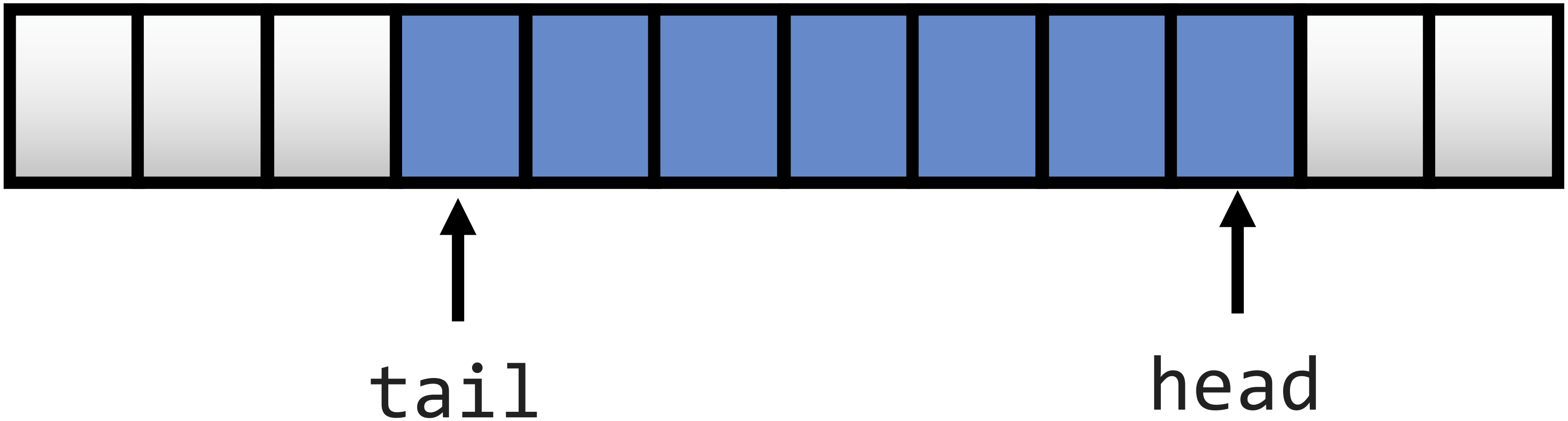
Memory   amount 50,000

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 (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

My experience until now

1. Students develop their concurrent 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
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

My take on the problem

- Handwritten correctness 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 have a steep learning curve

Enter *Harmony*

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

it tries *all* possible executions of a program
until it finds a problem
(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

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();
```

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
```

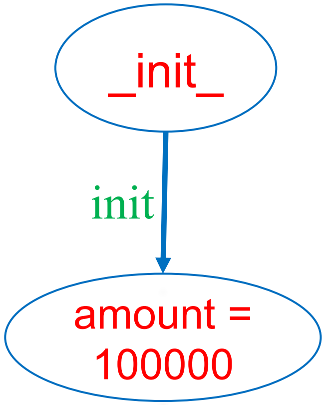
Simplified model (ignoring main)

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

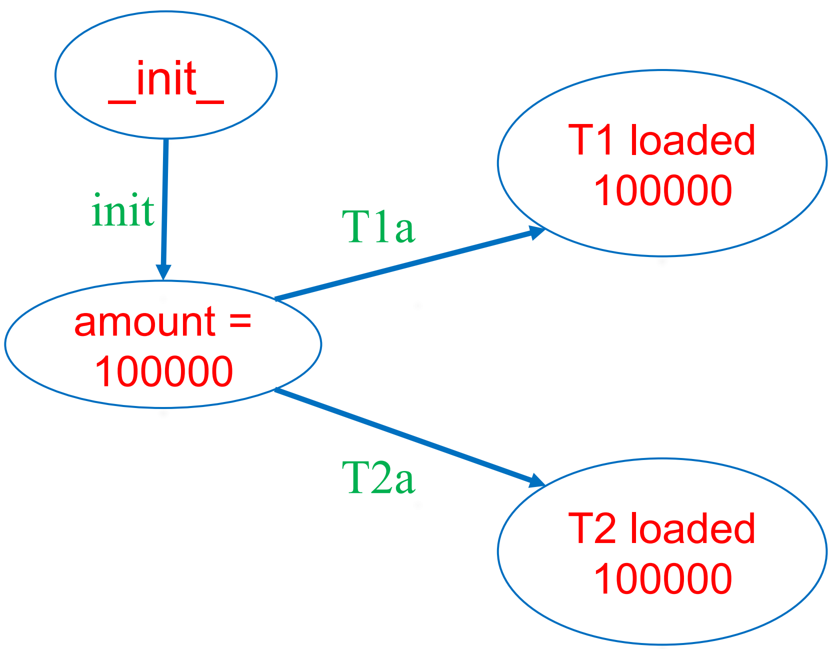
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

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

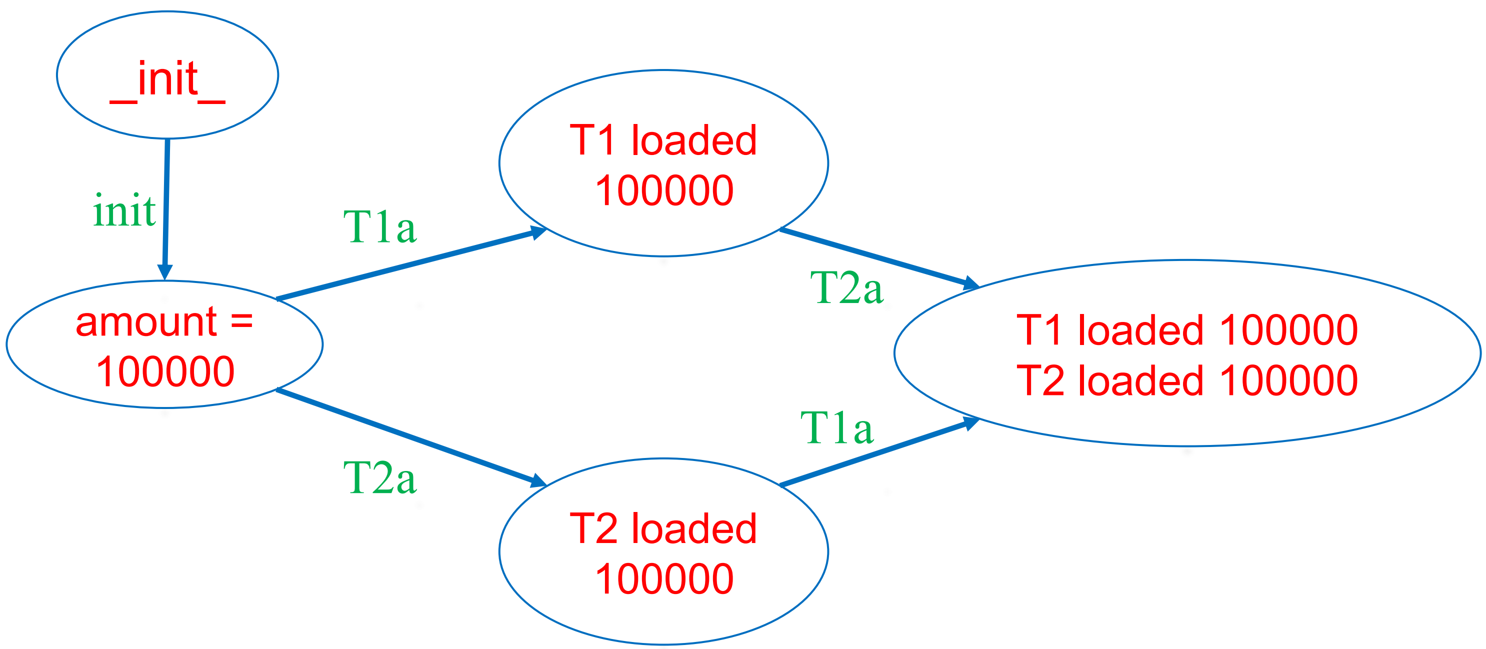
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

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

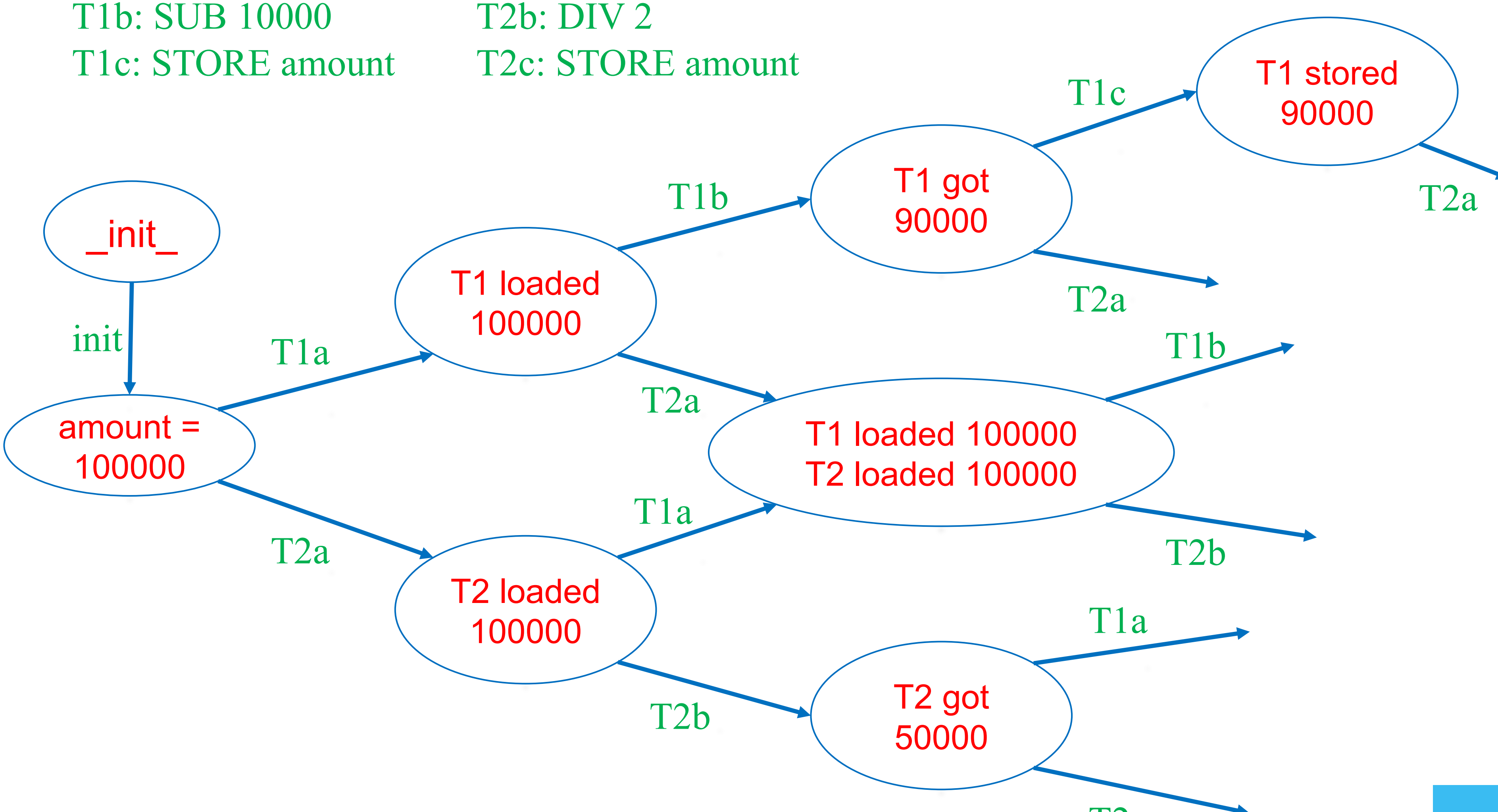
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



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

diameter of 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 ====

path

```
__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 = 100 diameter = 5
              ==== Safety violation ====
_init_        __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 = 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 tag” of a process

__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

"microsteps" =
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
7  Store done1        T1d: done1 = True
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
16 Store done2        T2d: done2 = True
17 Return
18 …
```

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 process 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 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	
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
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 process 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 process 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 process T2
12 Push 2             push 2 onto the stack of process 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 process T2
16 Store done2        store top of the stack of T2 into done2
17 Return
18 …
```

Output

current program counter
(after microsteps)

```
__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 microsteps)

```
__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)

- Name tag
- 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
every statement ends in ;	; at end of statement optional
indentation recommended	indentation required
(…) == […] == …	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
{ } is empty set	set() != { }
few operator precedence rules --- use brackets 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)

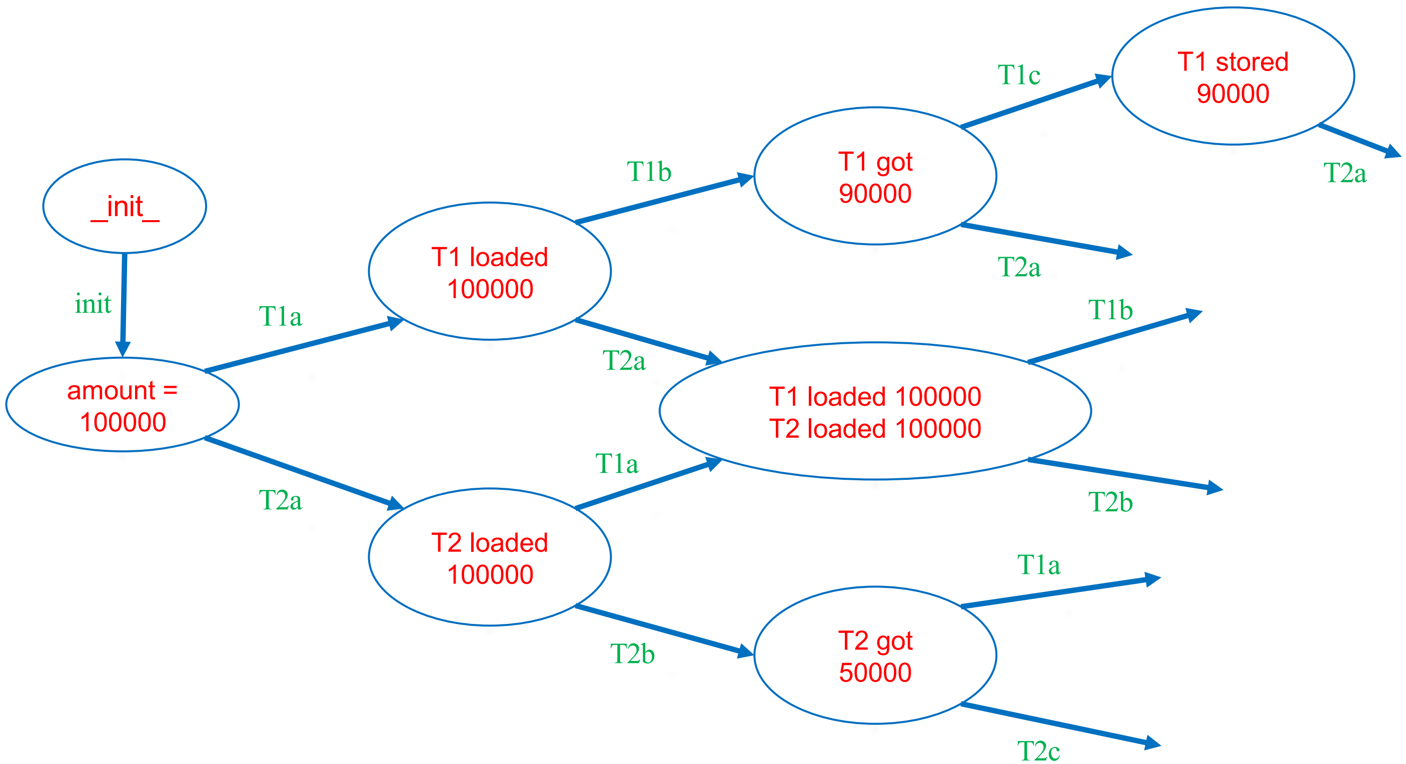
No open(), read(), input(), or print() statements

Non-determinism in Harmony

Two sources:

1. **choose** expressions
2. **process** interleavings

Limitation: models must be finite!



- But models are allowed to have cycles.
- Executions are allowed to be unbounded!
- Harmony does check 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%

T1

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

T2

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

Memory amount 100,000

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: all threads make it into the CS if desired

Fairness: equal chances of getting into CS

… in practice, fairness rarely guaranteed

Critical Sections in Harmony

```
def process(self):
    while True:
        …    # code outside critical section
        …    # code to enter the critical section
        …    # critical section itself
        …    # code to exit the critical section
    ;
;
spawn process(1);
spawn process(2);
…
```

- How do we check mutual exclusion?
- How do we check termination?

Critical Sections in Harmony

```
def process(self):
    while True:
        …   # code outside critical section
        …   # code to enter the critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        …   # code to exit the critical section
    ;
;
spawn process(1);
spawn process(2);
…
```

- How do we check mutual exclusion?
- How do we check progress?

Critical Sections in Harmony

```
def process(self):
    while choose( { False, True } ):
        …    # code outside critical section
        …    # code to enter the critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        …    # code to exit the critical section
    ;
;
spawn process(1);
spawn process(2);
…
```

- How do we check mutual exclusion?
- How do we check progress?
  - *if code to enter/exit the critical section does not 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

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        lockTaken = False;
    ;
;
lockTaken = False;
spawn process(0);
spawn process(1);
```

Figure 5.4: [code/naiveLock.hny] Naïve implementation of a shared lock.

First attempt: a naïve lock

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        lockTaken = False;
    ;
;
lockTaken = False;
spawn process(0);
spawn process(1);
```

wait till lock is free, then take it

Figure 5.4: [code/naiveLock.hny] Naïve implementation of a shared lock.

First attempt: a naïve lock

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        lockTaken = False;
    ;
;
lockTaken = False;
spawn process(0);
spawn process(1);
```

wait till lock is free, then take it

release the lock

Figure 5.4: [code/naiveLock.hny] Naïve implementation of a shared lock.

First attempt: a naïve lock

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

Figure 5.4: [code/naiv

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

Second attempt: *flags*

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
spawn process(0);
spawn process(1);
```

Figure 5.6: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
spawn process(0);
spawn process(1);
```

enter, then wait for other

Figure 5.6: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
spawn process(0);
spawn process(1);
```

enter, then wait for other

leave

Figure 5.6: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, Fal
spawn process(0);
spawn process(1);
```

Figure 5.6: [code/naiv

```
==== Non-terminating State ===
__init__/() [0,36-46]                46 { flags: [False, False] }
process/0 [1-2,3(choose True),4-12] 13 { flags: [True,  False] }
process/1 [1-2,3(choose True),4-12] 13 { flags: [True,  True] }
blocked process: process/1 pc = 13
blocked process: process/0 pc = 13
```

Third attempt: *turn* variable

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        turn = 1 − self;
    ;
;
turn = 0;
spawn process(0);
spawn process(1);
```

Figure 5.8: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

Third attempt: *turn* variable

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        turn = 1 − self;
    ;
;
turn = 0;
spawn process(0);
spawn process(1);
```

wait for your turn

Figure 5.8: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

Third attempt: *turn* variable

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

wait for your turn

```

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        turn = 1 - self;
```

let the other process take a turn

```
    ;
;
turn = 0;
spawn process(0);
spawn process(1);
```

Figure 5.8: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

Third attempt: *turn* variable

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

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        turn = 1 − self;
    ;
;
turn = 0;
spawn process(0);
spawn process(1);
```

Figure 5.8: [code/naiveTu

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

Peterson's Algorithm: *flags & turn*

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

Peterson’s Algorithm: *flags & turn*

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

“you go first”

Peterson's Algorithm: *flags & turn*

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

"you go first"

wait until alone or
it's my turn

Peterson's Algorithm: *flags & turn*

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

"you go first"

wait until alone or it's my turn

leave

Peterson's Algorithm: *flags & turn*

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

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

So, we proved Peterson's Algorithm correct by brute force, enumerating all possible executions.

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*

- 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]

First question: what should IH be?

- Obvious answer: mutual exclusion itself
  - if $P0$ is in the critical section, then $P1$ is not
    - without loss of generality…
  - Formally: $P0@cs \implies \neg P1@cs$

- Unfortunately, this won't work…

State before P1 takes a step:

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        turn = 1 − self;
P1      await (not flags[1 − self]) or (turn == self);

        # critical section is here
P0      @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

flags == [True, True]
turn == 1

State after P1 takes a step:

flags == [True, True]
turn == 1

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

P0 P1

So, is Peterson's Algorithm broken?

No, it'll turn out this prior state cannot be reached from the initial state

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        turn = 1 − self;
P1      await (not flags[1 − self]) or (turn == self);

        # critical section is here
P0      @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

flags == [True, True]
turn == 1

Let’s try another obvious one

- Based on the **await** condition:

$$P0@cs \implies \neg flags[1] \lor turn == 0$$

- Promising because if $P0@cs \land P1@cs$ then

$$\left.\begin{array}{r} P0@cs \implies \neg flags[1] \lor turn == 0 \land \\ P1@cs \implies \neg flags[0] \lor turn == 1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} turn == 0 \ \land \\ \ turn == 1 \end{array}\right.$$

$\implies$ False (therefore mutual exclusion)

- Unfortunately, this is not an invariant…

State before P1 takes a step:

```
      def process(self):
          while choose({ False, True }):
P1            # Enter critical section
              flags[self] = True;
              turn = 1 − self;
              await (not flags[1 − self]) or (turn == self);

              # critical section is here
P0            @cs: assert atLabel.cs == dict{ nametag(): 1 };

              # Leave critical section
              flags[self] = False;
          ;
      ;
      flags = [ False, False ];
      turn = choose({0, 1});
      spawn process(0);
      spawn process(1);
```

flags == [True, False]
turn == 1

$P0@cs \Longrightarrow \neg flags[1] \lor turn == 0$ holds

note: this is a reachable state

State after P1 takes a step:

```
def process(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 == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
spawn process(0);
spawn process(1);
```

P1

P0

flags == [True, True]
turn == 1

$P0@cs \Longrightarrow \neg flags[1] \lor turn == 0$ *violated*

note: this is also a reachable state

But suggests an improved hypothesis

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
P1      @gate: turn = 1 − self;
        await (not flags[1 − self]) or (turn == self);

        # Critical section
P0      @cs: assert (not flags[1 − self]) or (turn == self) or
                 (atLabel.gate == dict{nametags[1 − self] : 1})
```

$$P0@cs \implies \neg flags[1] \vee turn == 0 \vee P1@gate$$

```
        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = choose({0, 1});
nametags = [ dict{ .name: .process, .tag: tag } for tag in {0, 1} ];
spawn process(0);
spawn process(1);
```

Figure 6.3: [code/PetersonInductive.hny] Peterson's Algorithm with Inductive Invariant

Invariance proof

To prove: $P0@cs \implies \neg flags[1] \vee turn == 0 \vee P1@gate$

By induction:

Base case:

- In initial state $\neg P0@cs$
  - false implies anything

Induction Step: assume $P0@cs$ and $P1$ takes a step when

Case 1: $\neg flags[1]$

then after step either $\neg flags[1]$ or $P1@gate$

Case 2: $turn == 0$

then after step still $turn == 0$ ($P1$ never sets turn to 1)

Case 3: $P1@gate$

then after step $turn == 0$

Finally, prove mutual exclusion

$$P0@cs \wedge P1@cs \Longrightarrow$$

$$\begin{cases} \neg flags[1] \vee turn == 0 \vee P1@gate \\ \neg flags[0] \vee turn == 1 \vee P0@gate \end{cases} \wedge$$

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

$$\Longrightarrow False$$

Finally, prove mutual exclusion

$$P0@cs \land P1@cs \Longrightarrow$$

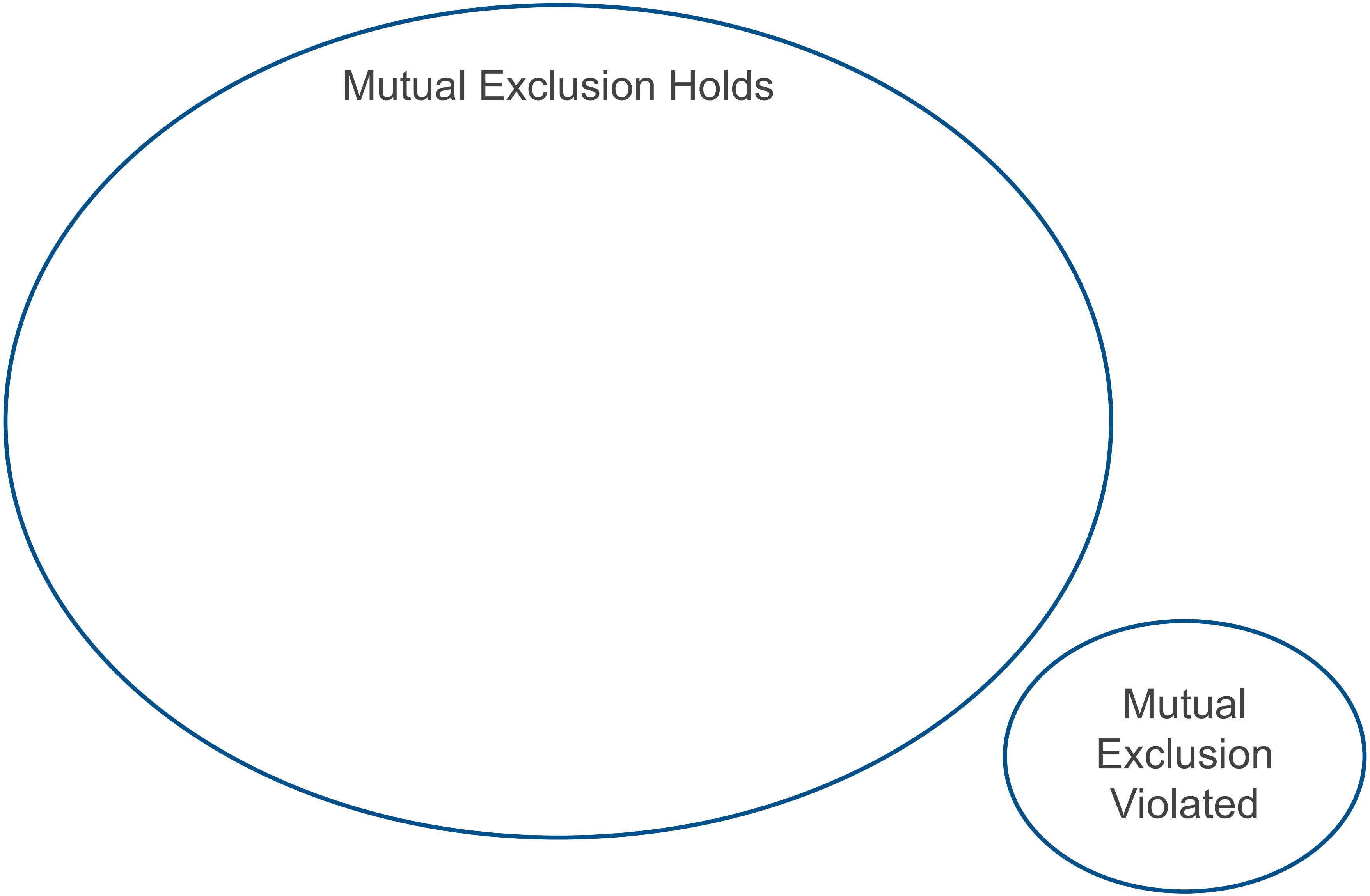
$$\begin{cases} \neg flags[1] \lor turn == 0 \lor P1@gate \\ \neg flags[0] \lor turn == 1 \lor P0@gate \end{cases} \land$$

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

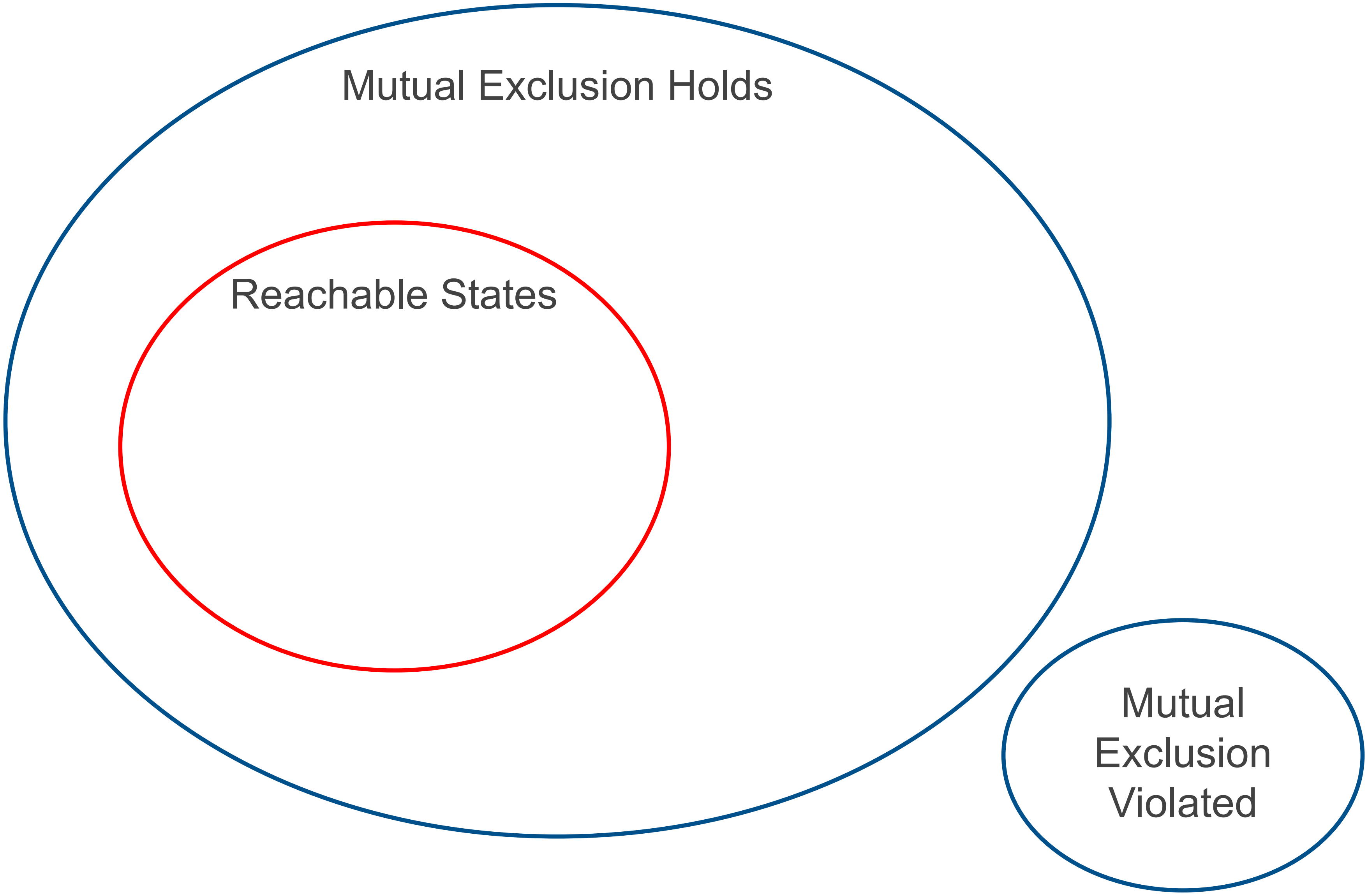
$$\Longrightarrow False$$

QED

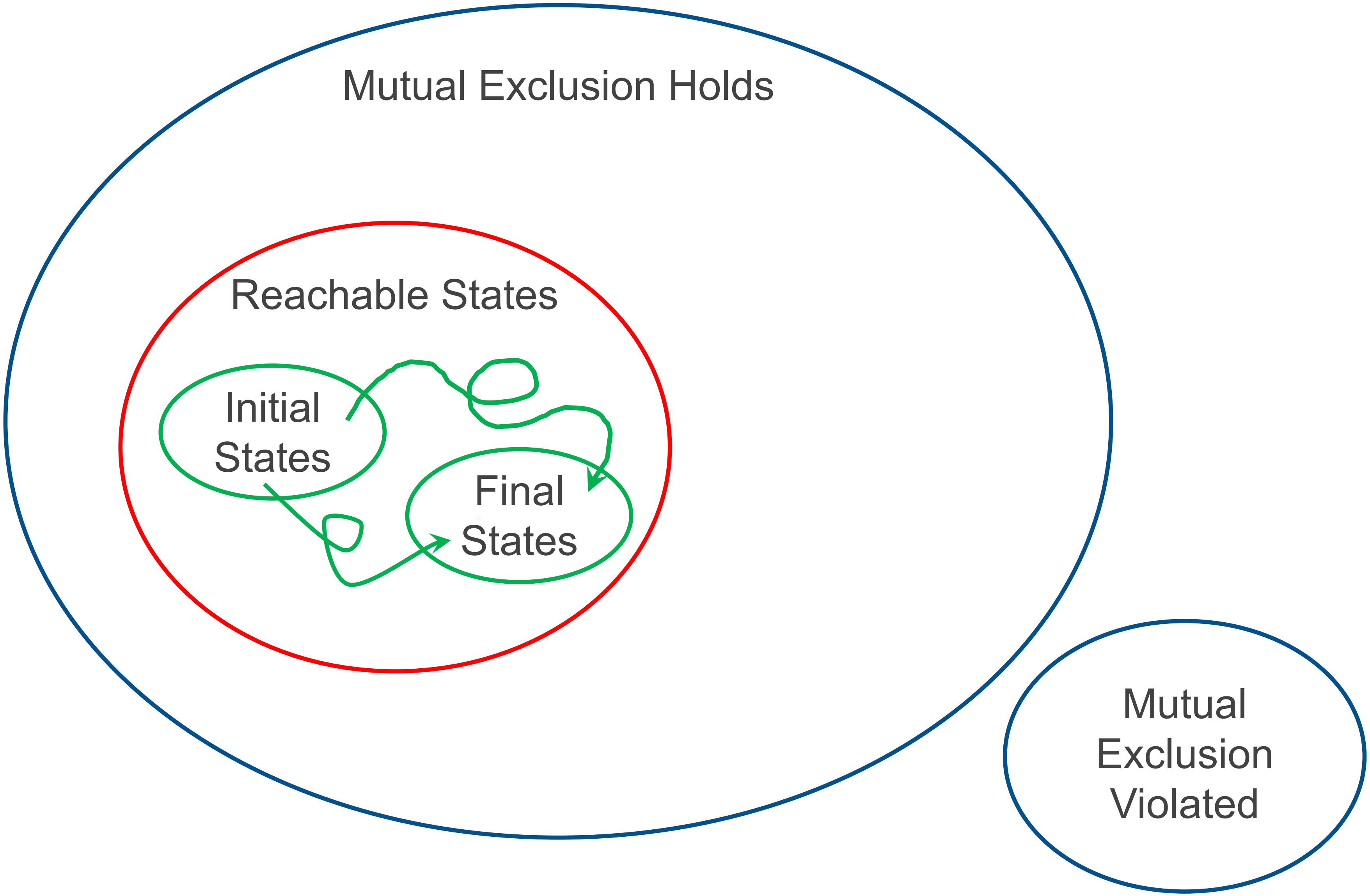
Review in Pictures: State Space



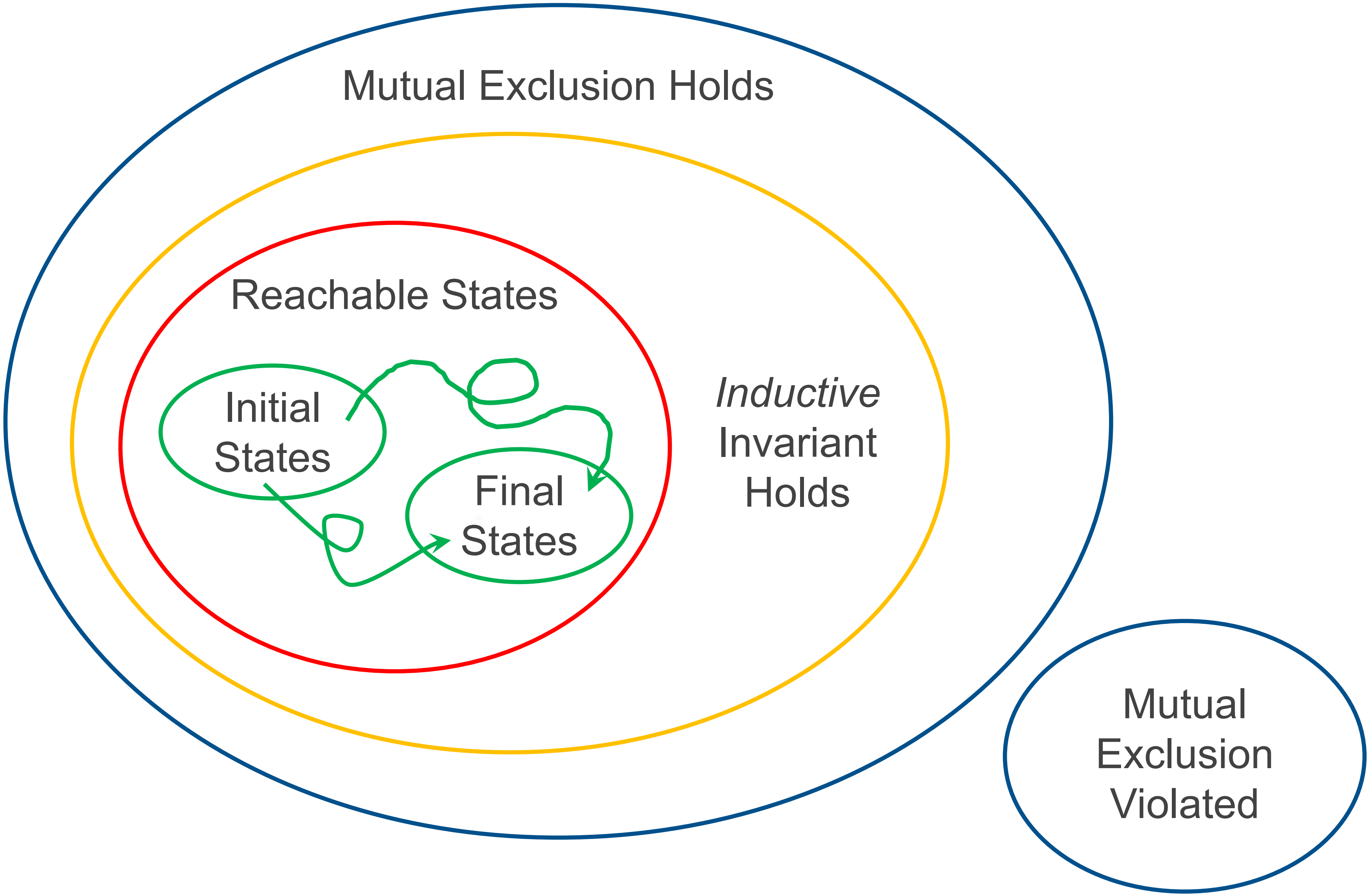
Review in Pictures: State Space



Review in Pictures: State Space



Review in Pictures: State Space



Inductive Invariant

II is an *inductive invariant* if for *any* state S (including unreachable ones!):

- Base case: II holds if S is an initial state
- Induction step: if II holds in S, then II also holds in any states reachable from S in one step

Note, an ordinary invariant only needs to hold in all reachable states

II is useful if it implies an invariant that we are interested in (mutual exclusion in this case)

Peterson’s Reconsidered

- Mutual Exclusion can be implemented with LOAD and STORE instructions to access shared memory
  - 3 STOREs and 1 or more LOADs
- Peterson’s can be generalized to >2 processes
  - even more STOREs and LOADs
- *Too inefficient in practice*

Enter *Interlock Instructions*

- 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*

- 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*

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

- 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

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

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;

def process(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 == dict{ nametag(): 1 })
            ;

        # Leave critical section
        private[self] = True;
        shared = False;
    ;
;
shared = False;
private = [ True, ] * N;
for i in {0..N−1}:
    spawn process(i);
;
```

number of processes

"spinlock"

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

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

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

Two essential invariants

1. $\forall i: process(i)@cs \Rightarrow \neg private[i]$
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 (see book)

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

Checking the second invariant

```
import spinlock;

def checkInvariant():
    let sum = 0:
        if not shared:
            sum = 1;
        ;
        for i in {0..N−1} such that not private[i]:
            sum += 1;
        ;
        result = sum <= 1;
    ;
;
def invariantChecker():
    assert checkInvariant();
;
spawn invariantChecker();
```

Check that at most one of *shared* and *private*[*i*] is False

check it here, atomically

Figure 8.2: [code/spinlockInv.hny] Checking invariants.

assert statements are evaluated atomically

Checking the second invariant

```
import spinlock;

def checkInvariant():
    let sum = 0:
        if not shared:
            sum = 1;
        ;
        for i in {0..N−1} such that not private[i]:
            sum += 1;
        ;
        result = sum <= 1;
    ;
;
def invariantChecker():
    assert checkInvariant();
;
spawn invariantChecker();
```

Riddle: this code checks the invariant only once, and yet it checks the invariant at every state.

How can that be?

Figure 8.2: [code/spinlockInv.hny] Checking invariants.

“Locks”

Best understood as “baton passing”

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

Locks in the “synch” module

```
def tas(lk):
    atomic:
        result = !lk;
        !lk = True;
    ;
;
def Lock():
    result = False;
;
def lock(lk):
    await not tas(lk);
;
def unlock(lk):
    !lk = False;
;
```

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

Figure 9.2: [modules/synch.hny] The **Lock** 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 process
- The invariants become:
  1. $P@cs \Rightarrow P$ holds the lock
  2. at most one process can hold the lock

Using locks from the sync module

import the sync module

```
import synch;

def process(self):
    lock(?countlock);
    count = count + 1;
    unlock(?countlock);
    done[self] = True;
;
def main(self):
    await all(done);
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

Figure 9.3: [code/UpLock.hny] Program of Figure 3.1 fixed with a lock.

Using locks from the sync module

```
import synch;

def process(self):
    lock(?countlock);
    count = count + 1;
    unlock(?countlock);
    done[self] = True;
;
def main(self):
    await all(done);
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

import the sync module

initialize lock

Figure 9.3: [code/UpLock.hny] Program of Figure 3.1 fixed with a lock.

Using locks from the sync module

```
import synch;

def process(self):
    lock(?countlock);
    count = count + 1;
    unlock(?countlock);
    done[self] = True;
;
def main(self):
    await all(done);
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

import the sync module

enter critical section

initialize lock

Figure 9.3: [code/UpLock.hny] Program of Figure 3.1 fixed with a lock.

Using locks from the sync module

```
import synch;

def process(self):
    lock(?countlock);
    count = count + 1;
    unlock(?countlock);
    done[self] = True;
;
def main(self):
    await all(done);
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

import the sync module

enter critical section

?*countlock* is the address of *countlock*

process *self* holds *countlock*

initialize lock

Figure 9.3: [code/UpLock.hny] Program of Figure 3.1 fixed with a lock.

Using locks from the sync module

```
import synch;

def process(self):
    lock(?countlock);
    count = count + 1;
    unlock(?countlock);
    done[self] = True;
;
def main(self):
    await all(done);
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

import the sync module

enter critical section

exit critical section

initialize lock

Figure 9.3: [code/UpLock.hny] Program of Figure 3.1 fixed with a lock.

Spinlocks and Time Sharing

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

Context switching in Harmony

- Harmony allows contexts to be saved and restored

  - *r* = **stop** *list*
    - stops the current process and places its context at the end of the given list
  - **go** *context r*
    - adds a process with the given context to the bag of processes. Process resumes from **stop** expression, returning *r*

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

```
import list;

def Lock():
    result = dict{ .locked: False, .suspended: [ ] };
;
def lock(lk):
    atomic:
        if lk→locked:
            stop lk→suspended;
            assert lk→locked;
        else:
            lk→locked = True;
        ;
    ;
;
def unlock(lk):
    atomic:
        if lk→suspended == [ ]:
            lk→locked = False;
        else:
            go (head(lk→suspended)) ();
            lk→suspended = tail(lk→suspended);
        ;
    ;
;
```

Figure 9.4: [modules/syncS.hny] The Lock interface in the synchS module uses suspension.

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

```
import list;

def Lock():
    result = dict{ .locked: False, .suspended: [ ] };
;
def lock(lk):
    atomic:
        if lk→locked:
            stop lk→suspended;
            assert lk→locked;
        else:
            lk→locked = True;
        ;
    ;
;
def unlock(lk):
    atomic:
        if lk→suspended == [ ]:
            lk→locked = False;
        else:
            go (head(lk→suspended)) ();
            lk→suspended = tail(lk→suspended);
        ;
    ;
;
```

lk→locked is short for (!*lk*).locked (cf. C, C++)

Figure 9.4: [modules/syncS.hny] The `Lock` interface in the `synchS` module uses suspension.

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

```
import list;

def Lock():
    result = dict{ .locked: False, .suspended: [ ] };
;
def lock(lk):
    atomic:
        if lk→locked:
            stop lk→suspended;
            assert lk→locked;
        else:
            lk→locked = True;
        ;
    ;
;
def unlock(lk):
    atomic:
        if lk→suspended == [ ]:
            lk→locked = False;
        else:
            go (head(lk→suspended)) ();
            lk→suspended = tail(lk→suspended);
        ;
    ;
;
```

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

Figure 9.4: [modules/syncS.hny] The Lock interface in the synchS module uses suspension.

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

- “sync” tends to be faster than “syncS”
  - smaller state graph

Atomic section ≠ Critical Section

Atomic Section	Critical Section
only one process can execute	multiple process can execute concurrently, just not within a critical section
rare programming 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 = Qnew(): allocate a new queue
- Qenqueue(q, v): add v to the tail of queue q
- r = Qdequeue(q): returns r = () if q is empty or r = (v,) (a singleton tuple) if v was at the head of the queue

Queue Test Program Example

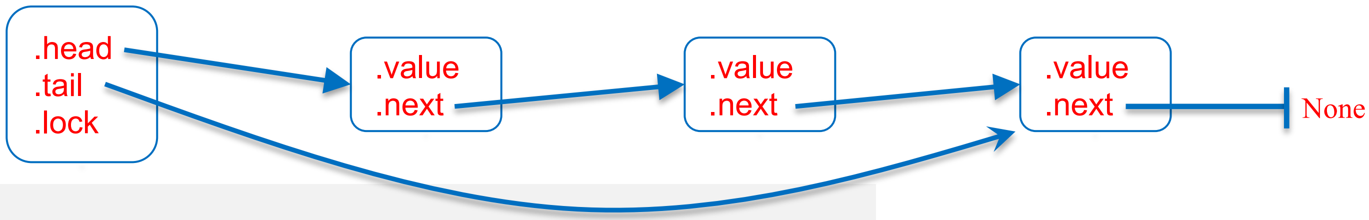
```
import queue;

def sender(q, v):
    Qenqueue(q, v);
;
def receiver(q):
    let done = False:
        while not done:
            let v = Qdequeue(q):
                done = v == ();
                assert done or (v[0] in { 1, 2 });
            ;
        ;
    ;
;

queue = Qnew();
spawn sender(?queue, 1);
spawn sender(?queue, 2);
spawn receiver(?queue);
spawn receiver(?queue);
```

enqueue *v* onto *q*

dequeue until queue *q* is empty

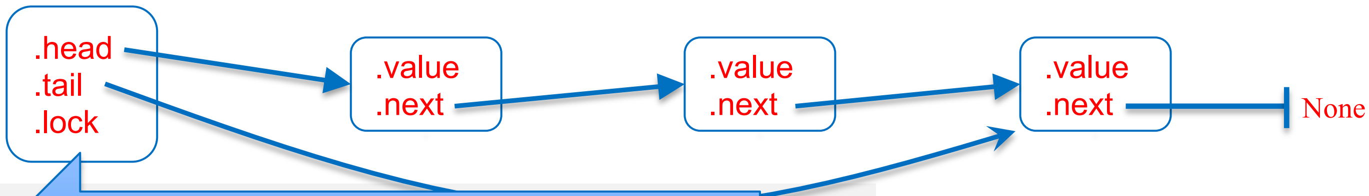
Figure 10.1: [code/queuetest.hny] Test program for a concurrent queue.

Queue implementation, v1




```
import alloc;

def Qnew():
    result = dict{ .head: None, .tail: None, .lock: Lock() };
;
def Qenqueue(q, v):
    let node = malloc(dict{ .value: v, .next: None }):
        lock(?q→lock);
        if q→head == None:
            q→head = q→tail = node;
        else:
            q→tail→next = node;
            q→tail = node;
        ;
        unlock(?q→lock);
    ;
;
```

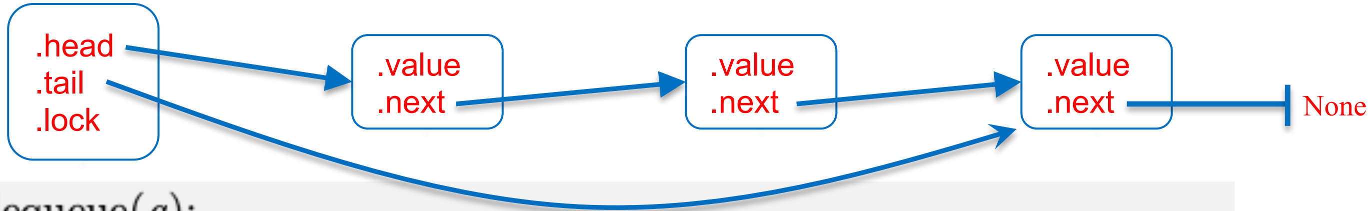
Queue implementation, v1



dynamic memory allocation

```
import alloc;

def Qnew():
    result = dict{ .head: None, .tail: None, .lock: Lock() };
;
def Qenqueue(q, v):
    let node = malloc(dict{ .value: v, .next: None }):
        lock(?q→lock);
        if q→head == None:
            q→head = q→tail = node;
        else:
            q→tail→next = node;
            q→tail = node;
        ;
        unlock(?q→lock);
    ;
;
```

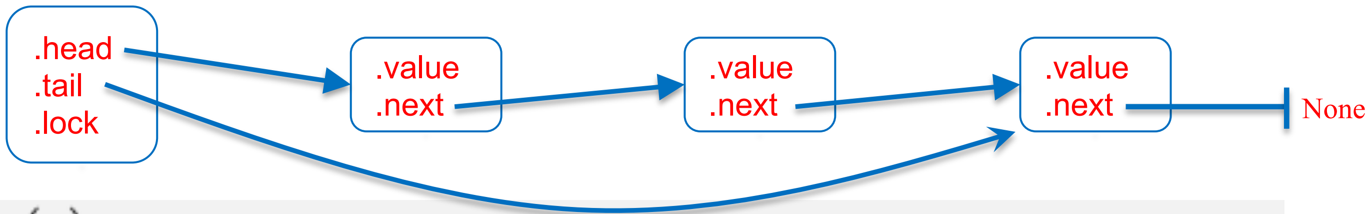
Queue implementation, v1




```
def Qdequeue(q):
    lock(?q→lock);
    let node = q→head:
        if node == None:
            result = ();
        else:
            result = (node→value,);
            q→head = node→next;
            if q→head == None:
                q→tail = None;
            ;
            free(node);
        ;
    ;
    unlock(?q→lock);
;
```

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

Queue implementation, v1




```
def Qdequeue(q):
    lock(?q→lock);
    let node = q→head:
        if node == None:
            result = ();
        else:
            result = (node→value,);
            q→head = node→next;
            free(node);
        ;
    ;
    unlock(?q→lock);
;
```

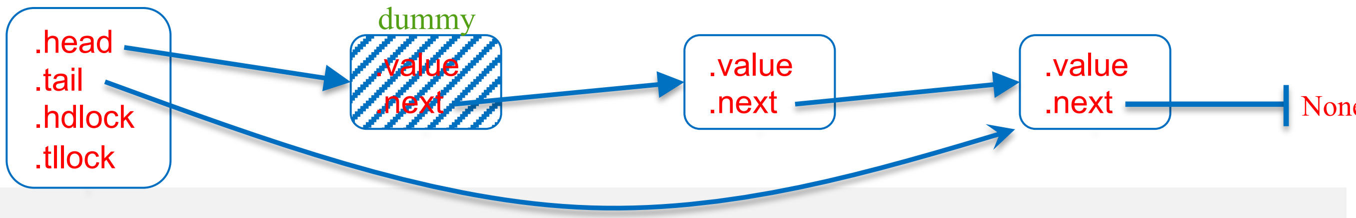
malloc'd memory must be explicitly released (cf. C)

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

How important are concurrent queues?

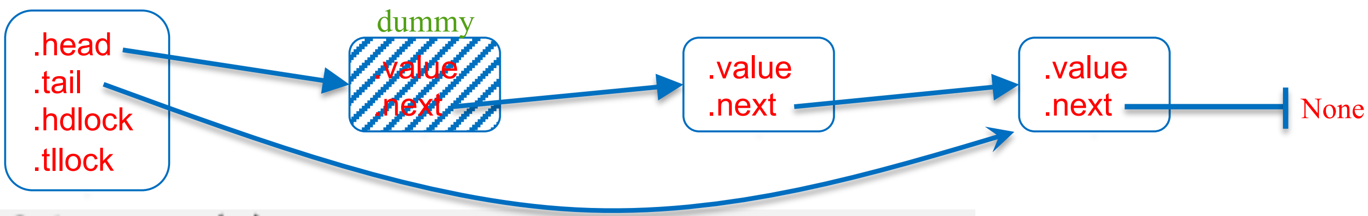
- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

Better concurrent queue: 2 locks




```
import alloc;

def Qnew():
    let dummy = malloc(dict{ .value: (), .next: None }):
        result = dict{ .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() };
    ;
;
def Qenqueue(q, v):
    let node = malloc(dict{ .value: v, .next: None }):
        lock(?q→tllock);
        q→tail→next = node;
        q→tail = node;
        unlock(?q→tllock);
    ;
;
```

Better concurrent queue: 2 locks




```
def Qdequeue(q):
    lock(?q→hdlock);
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = ();
        else:
            free(dummy);
            result = (node→value,);
            q→head = node;
        ;
    ;
    unlock(?q→hdlock);
;
```

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

Figure 10.3: [code/queueMS.hny] A queue with separate locks f

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

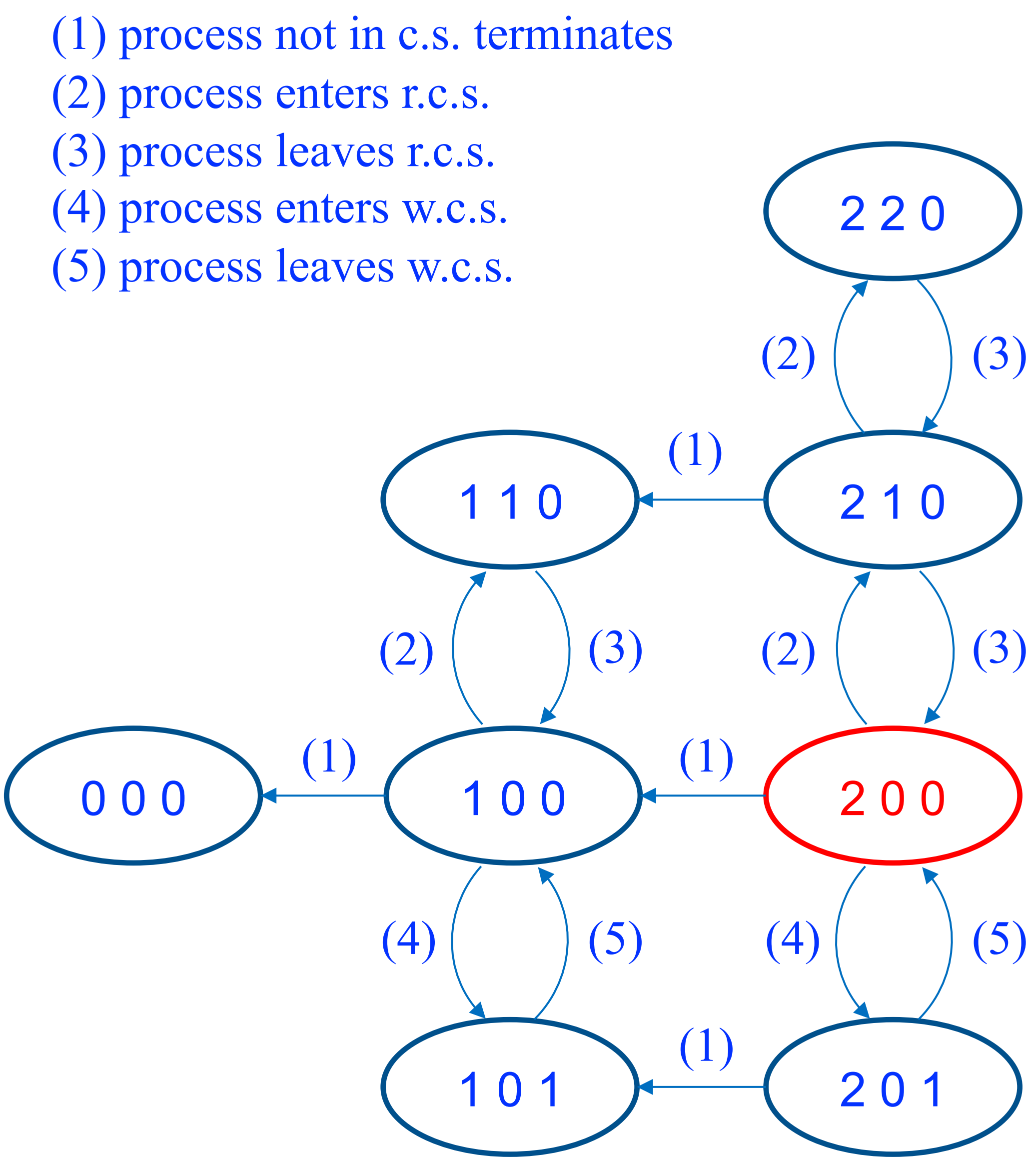
Either:

- multiple readers, or
- a single writer

thus not:

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

Reader/writer lock state diagram



Reader/writer lock interface:

- acquire_rlock()
  - get a read lock. Multiple processes can have the read lock simultaneously, but no process can have a write lock simultaneously
- release_rlock()
  - release a read lock. Other processes may still have the read lock. When the last read lock is released, a write lock may be acquired
- acquire_wlock()
  - acquire the write lock. Only one process can have a write lock, and if so no process can have a read lock
- release_wlock()
  - release the write lock. Allows other processes to either get a read or write lock

R/W lock, Implementation #1

- Uses a single ordinary lock and two integers to count #readers and #writers

```
rwlock = Lock();
nreaders = 0;
nwriters = 0;
```

Figure 11.1: [code/RW.hny] Busy-Waiting Reader/Writer Lock

Invariants:

- if n readers in the critical section, then $nreaders \geq n$
- if n writers in the critical section, then $nwriters \geq n$
- $(nreaders \geq 0 \land nwriters = 0) \lor (nreaders = 0 \land 0 \leq nwriters \leq 1)$

rwlock protects the *nreaders*/*nwriters* variables, not the critical section!

R/W lock, Implementation #1

```
import synch;

def acquire_rlock():
    let blocked = True:
        while blocked:
            lock(?rwlock);
            if nwriters == 0:
                nreaders += 1;
                blocked = False;
            ;
            unlock(?rwlock);
        ;
    ;
;
def release_rlock():
    lock(?rwlock);
    nreaders -= 1;
    unlock(?rwlock);
;
```

R/W lock, Implementation #1

```
import synch;

def acquire_rlock():
    let blocked = True:
        while blocked:
            lock(?rwlock);
            if nwriters == 0:
                nreaders += 1;
                blocked = False;
            ;
            unlock(?rwlock);
        ;
    ;
;
def release_rlock():
    lock(?rwlock);
    nreaders -= 1;
    unlock(?rwlock);
;
```

“busy wait” (i.e., spin) until no writer in the critical section

R/W lock, Implementation #1

```
def acquire_wlock():
    let blocked = True:
        while blocked:
            lock(?rwlock);
            if (nreaders + nwriters) == 0:
                nwriters = 1;
                blocked = False;
            ;
            unlock(?rwlock);
        ;
    ;
;
def release_wlock():
    lock(?rwlock);
    nwriters = 0;
    unlock(?rwlock);
```

"busy wait" until no other process in the critical section

R/W Locks: test for mutual exclusion

```
import RW;

def process():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            acquire_rlock();
            @rcs: assert atLabel.wcs == dict{};
            release_rlock();
        else:                          # .write
            acquire_wlock();
            @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                         (atLabel.rcs == dict{})
                  ;
            release_wlock();
        ;
    ;
;
for i in {1..4}:
    spawn process();
;
```

no writer

1 writer and no readers

Figure 11.2: [code/RWtest.hny] Test code for Figure 11.1.

About *busy waiting*

- ok for multi-core (true) parallelism
- bad for time-sharing (virtual) parallelism

R/W Lock, implementation #2

- Uses two ordinary locks and an integer that counts the number of readers

$$
\begin{aligned}
&rwlock = \text{Lock}();\\
&rlock = \text{Lock}();\\
&nreaders = 0;
\end{aligned}
$$

Figure 12.1: [code/RWlock.hny]

Invariants:

- if n readers in the critical section, then $nreaders \geq n$
- if a writer in the critical section, then $nreaders = 0$
- if writer W in the critical section, then W holds *rwlock*

(if some reader in the critical section, the readers collectively hold *rwlock*)

rlock protects the *nreaders* variable

R/W Lock, implementation #2

```
def acquire_rlock():
    lock(?rlock);
    if nreaders == 0:
        lock(?rwlock);
    ;
    nreaders += 1;
    unlock(?rlock);
;
def release_rlock():
    lock(?rlock);
    nreaders -= 1;
    if nreaders == 0:
        unlock(?rwlock);
    ;
    unlock(?rlock);
;
def acquire_wlock():
    lock(?rwlock);
;
def release_wlock():
    unlock(?rwlock);
;
```

rlock protects the nreaders variable

first reader acquires *rwlock*

last reader releases *rwlock*

writer acquires *rwlock*

writer releases *rwlock*

R/W Lock, implementation #2

nreaders == 0

R1 R2

```
def acquire_rlock():
    lock(?rlock);
    if nreaders == 0:
        lock(?rwlock);
    ;
    nreaders += 1;
    unlock(?rlock);
;
def release_rlock():
    lock(?rlock);
    nreaders -= 1;
    if nreaders == 0:
        unlock(?rwlock);
    ;
    unlock(?rlock);
;
def acquire_wlock():
    lock(?rwlock);
;
def release_wlock():
    unlock(?rwlock);
;
```

W

Reader R1 holds *rlock* and is waiting for *rwlock*

Reader R2 is waiting for *rlock*

Reader R1 holds *rwlock* and released *rlock*

Reader R2 released *rlock*

Reader R1 leaves but *rwlock* is still "held"

Reader R2 released *rwlock*

Writer W is in the critical section and holds *rwlock*

Writer W left the critical section

R/W Lock, implementation #2

```
def acquire_rlock():
    lock(?rlock);
    if nreaders == 0:
        lock(?rwlock);
    ;
    nreaders += 1;
    unlock(?rlock);
;
def release_rlock():
    lock(?rlock);
    nreaders -= 1;
    if nreaders == 0:
        unlock(?rwlock);
    ;
    unlock(?rlock);
;
def acquire_wlock():
    lock(?rwlock);
;
def release_wlock():
    unlock(?rwlock);
;
```

no busy waiting!

both readers and writers "block" when they can't enter the critical section

More testing of reader/writer lock implementations

- Prior test only checks mutual exclusion
- How do you test if the implementation allows multiple readers?
- How do you test if the implementation uses busy waiting or not?

For both, see book