Once again, our 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()
Once again, our example

```python
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()
```
Once again, our example

```python
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
```
Once again, our example

```python
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
Once again, our example

```python
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
- Semantically an assertion is a no-op
  - it is never expected to fail because it is supposed to state a fact
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 are 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  - comments tend to get outdated over time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...

- Comment them out before submitting a programming assignment
  - you don’t want your assertions to fail while we are testing your code... 😊
Back to our example

```python
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
Back to our example

```python
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)
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 ====
**Simplified model (ignoring main)**

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

(init)

1. **_init_**
2. **init**
3. **amount = 100000**
4. **T1 loaded 100000**
5. **T2 loaded 100000**
6. **T1 got 90000**
7. **T1 stored 90000**
8. **T1 got 50000**
9. **T2 loaded 100000**
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()
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
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

length of the longest path in turns
#states = 100  diameter = 5

===== Safety violation =====

<table>
<thead>
<tr>
<th>Method</th>
<th>Index</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>[0,40-58]</td>
<td>58 { amount: 100000, done1: False, done2: False }</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>[1-4]</td>
<td>5 { amount: 100000, done1: False, done2: False }</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>[10-17]</td>
<td>17 { amount: 50000, done1: False, done2: True }</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>[5-8]</td>
<td>8 { amount: 90000, done1: True, done2: True }</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>[19-23,25-34,36-37]</td>
<td>37 { amount: 90000, done1: True, done2: True }</td>
<td></td>
</tr>
</tbody>
</table>

>>> Harmony Assertion (file=test.hny, line=11) failed: 90000

Something went wrong in (at least) one path in the graph (assertion failure)
### Harmony Output

`#states = 100 diameter = 5`

```plaintext
==== 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
```
#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
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

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

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

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

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

#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
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

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

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

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

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

#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
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

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

>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
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
Harmony Output

```
#states = 100  diameter =
==== 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

"steps" = list of program counters of machine instructions executed
```
Harmony Machine code

0  Jump 40
1  Frame T1 ()
2  Load amount  T1a: LOAD amount
3  Push 10000  T1b: SUB 10000
def T1():
4  2-ary –  amount -= 10000
5  Store amount  done1 = True  
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
def T2():
13 2-ary /  amount /= 2  
14 Store amount  done2 = True
15 Push True
16 Store done2  T1d: done2 = True
17 Return
18 …
Harmony Machine code

0 Jump 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 stack of T1 into amount  
6 Push True push True onto stack of T1  
7 Store done1 store top of 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 quotient  
14 Store amount store top of stack of T1 into amount  
15 Push True push True onto stack of T2  
16 Store done2 store top of stack of T1 into done2  
17 Return  
18 …
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
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

current state (after turn)
Harmony’s VM **State**

- Three parts:
  - code (which never changes)
  - values of shared variables
  - states of each of the running processes
    - a.k.a. “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 (a.k.a. arguments)
  - Result
    - There is no `return` statement
  - Local variables
    - Declared in `var`, `let`, and `for` statements
Harmony != Python

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

...
I/O in Harmony

Input

- choose expression
  - $x = \text{choose} \{1, 2, 3\}$
  - allows Harmony to know all possible inputs

- const expression
  - const $x = 3$
  - can be overridden with “$-c\ x = 4$” to Harmony

Output

- print $x + y$
- assert $x + y < 10$, $(x, y)$
I/O in Harmony

Input

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

Output

- print $x + y$
- assert $x + y < 10$, $(x, y)$
Non-determinism in Harmony

- Three sources
  - choose expressions
  - thread interleavings
  - interrupts
Limitation: Models must be finite!

- But models are allowed to have cycles
- Executions are allowed to be unbounded
- Harmony checks for the possibility of termination
Back to our problem...

Two threads updating shared variable `amount`

- $T_1$ wants to decrement `amount` by $10K$
- $T_2$ wants to decrement `amount` by 50%

\[
\begin{align*}
\text{amount} & \leftarrow \text{amount} - 10,000; \\
& \ldots \\
\text{amount} & \leftarrow \text{amount} \times 0.5; \\
& \ldots
\end{align*}
\]

How to “serialize” these executions?
Critical Section

Shared memory access: must be serialized

 Goals

- **Mutual exclusion:** at most 1 thread in CS at any time
- **Progress:** all threads wanting to enter CS eventually do
- **Fairness:** equal chances to get into CS (uncommon in practice)
Goals

- **Mutual exclusion**: at most 1 thread in CS at any time
- **Progress**: if any threads want to enter the CS, at least one does

Shared memory access: must be serialized

T₁

. . .
CSEnter()

amount := amount - 10,000;

CSExit()

. . .

T₂

. . .
CSEnter()

amount := amount * 0.5;

CSExit()

. . .
What makes the Critical Section problem hard?

- Mutual exclusion?
- Progress?
- It is the combination!
  - both properties, on their own, are trivial to achieve
  - there is much more to this...
Critical Sections in Harmony

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

spawn T1()
spawn T2()
...

How do we check mutual exclusion?

How do we check progress?
Critical Sections in Harmony

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

spawn T1()
spawn T2()
...

How do we check mutual exclusion?
def thread(self):
    while True
        ...
        # code outside critical section
        ...
        # code to enter the critical section
        cs: assert countLabel(cs) == 1
        ...
        # code to exit the critical section

spawn T1()
spawn T2()
...
Critical Sections in Harmony

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

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

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

spawn T1()
spawn T2()
...

How do we check progress?

If code to enter/exit the critical section cannot terminate, Harmony will complain!
All you need is locks (tatta-rarararaa...)  

- At most one thread can hold the lock  
- Acquire the lock to enter the CS  
- Release the lock when exiting  
- But how does one build a lock?
Try 1: A Naïve Lock

```python
lockTaken = False

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

        # Critical section
        cs: assert countLabel(cs) == 1

        # Leave critical section
        lockTaken = False

    spawn thread(0)
    spawn thread(1)
```
Try 1: A Naïve Lock

```python
lockTaken = False

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

        # Critical section
        cs: assert countLabel(cs) == 1

        # Leave critical section
        lockTaken = False

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

Wait till lock is free, then take it

Release the lock
Try 1: A Naïve Lock

Testing and setting the lock is not atomic!

```python
lockTaken = False

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

    # Critical section
    cs: assert countLabel(cs) == 1

    # Leave critical section
    lockTaken = False

spawn thread/0
spawn thread/1
```

Wait till lock is free, then take it

Release the lock

==== Safety violation ====

```python
__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
Try 2: 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 countLabel(cs) == 1

        # Leave critical section
        flags[self] = False

spawn thread/0
spawn thread/1

Invariant:
Thread i in CS  \Rightarrow  flag[i] = True

Signal you want to enter
If someone in the CS, wait

Signal out of CS

==== 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
Try 3: Turns

```

```

Invariants:
Thread i in CS
=>

```

```

==== Non-terminating State ====

 initiator() [0,28-38]
 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
Peterson's Algorithm: Flags and Turns!

```
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 countLabel(cs) == 1

        # Leave critical section
        flags[self] = False

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

#states = 104 diameter = 5
#components: 37
no issues found
What about a proof?

To understand why it works...

We need to show that, for any execution, all states reached satisfy mutual exclusion

- i.e., that mutual exclusion is an invariant

See the Harmony book for a proof!
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 $x$
Using a lock for a critical section

```python
import synch

const NTHREADS = 2

lock = synch.Lock()

def thread():
    while choose({ False, True }):
        synch.acquire(lock)
        cs: assert countLabel(cs) == 1
        synch.release(lock)

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