Non-Determinism

(a) [code/prog1.hny] Sequential

1. \( \text{shared} = \text{True} \)
2. 
3. \( \text{def} \ f(): \text{assert} \ \text{shared} \)
4. \( \text{def} \ g(): \text{shared} = \text{False} \)
5. 
6. \( f() \)
7. \( g() \)

(b) [code/prog2.hny] Concurrent

1. \( \text{shared} = \text{True} \)
2. 
3. \( \text{def} \ f(): \text{assert} \ \text{shared} \)
4. \( \text{def} \ g(): \text{shared} = \text{False} \)
5. 
6. \( \text{spawn} \ f() \)
7. \( \text{spawn} \ g() \)

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

Figure 3.1: A sequential and a concurrent program.

(a) [code/prog1.hny] Sequential

shared = True

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

f()
g()

(b) [code/prog2.hny] Concurrent

shared = True

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

spawn f()
spawn g()

**Summary: something went wrong in an execution**

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

* Schedule thread T0: __init__()
  * Line 1: Initialize shared to True
  * **Thread terminated**
* Schedule thread T2: g()
  * Line 4: Set shared to False (was True)
  * **Thread terminated**
* Schedule thread T1: f()
  * Line 3: Harmony assertion failed
Non-Determinism

Two threads updating shared variable \texttt{amount}

- \(T_1\) wants to decrement \texttt{amount} by $10K
- \(T_2\) wants to decrement \texttt{amount} by 50%

\texttt{amount := amount - 10,000;}
\texttt{...}

\texttt{amount := amount * 0.5;}
\texttt{...}

What happens when \(T_1\) and \(T_2\) execute concurrently?
Non-Determinism

Might execute like this:

\[ r_1 := \text{load from amount} \]
\[ r_1 := r_1 - 10,000 \]
\[ \text{store } r_1 \text{ to amount} \]

\[ r_2 := \text{load from amount} \]
\[ r_2 := 0.5 \times r_2 \]
\[ \text{store } r_2 \text{ to amount} \]

Or vice versa: \( T_1 \) and then \( T_2 \)
But might also execute like this:

\[
\begin{align*}
    r1 &:= \text{load from amount} \\
    r1 &:= r1 - 10,000 \\
    \text{store } r1 \text{ to amount}
\end{align*}
\]

\[
\begin{align*}
    r2 &:= \text{load from amount} \\
    r2 &:= 0.5 \times r2 \\
    \text{store } r2 \text{ to amount}
\end{align*}
\]

One update is lost! Wrong – and very hard to debug
Race Conditions

Timing dependent behaviors involving shared state

- Behavior of race condition depends on how threads are scheduled!
  - a concurrent program can generate MANY “schedules” or “interleavings”
    - schedule: a total order of machine instructions
  - bug if any of them generates an undesirable behavior

All possible interleavings should be safe!
Race Conditions:
Hard to Debug

- Only some interleavings may produce a bug
- But bad interleavings may happen very rarely
  - program may run 100s of times without generating an unsafe interleaving
- Small changes to the program may hide bugs
  - “The Case of the Print Statement”
- Compiler and processor hardware can reorder instructions
Students develop their code in Python or C, and test it by running it many times. Testing can only prove the presence of bugs... not their absence!
Dutch Wisdom

True!
But there is testing
and then testing...
They submit their code,
confident that it is
correct...
Dutch Wisdom

and I test the code with my secret and evil methods...*

...and find that most submissions are broken!

*uses homebrew library that randomly samples from possible interleavings ("fuzzing")
I am unhappy, and the students are unhappy!

Why is that?

- Studies show that heavily used code, implemented, reviewed and tested by expert programmers has lots of concurrency bugs.
- Even professors who teach concurrency or write books or papers about concurrency get it wrong sometimes!
Hand-written proofs are just as likely to have bugs as programs... or even more likely, as you can’t test hand-written proofs!

There are no mainstream tools to check concurrent algorithms... those that exist have a steep learning curve.
Dutch Wisdom

Examples of existing tools

Spin

PlusCal

TLA+

```
bool turn, flag[2];
byte ncrit;
active [2] proctype user() // two processes
{
  assrot(_pid == 0 || _pid == 1);
  if (_pid) flag[_pid] = 1;
  turn = _pid;
  if (!flag[1 - _pid]) turn = 1 - _pid;
  ncrit++;
  assert(ncrit == 1)
  ncrit--;
  flag[_pid] = 0;
  goto again
}

--algorithm Peterson

variables flag = i \in {0, 1} \rightarrow FALSE
  \* declares the global variables flag
  \* flag is a 2-element array with initial
  \* fair process (proc \in (0,1))
  \* declares two processes with identical
  \* the keyword fair means that no process
  \* 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
}

VARIABLES flag, turn, pc
vars Δ {flag, turn, pc}
Init Δ ∧ flag = [i \in {0, 1} \rightarrow FALSE]
  ∧ turn = 0
  ∧ pc = [self \in {0, 1} \rightarrow \"a0\"
a3a(self) Δ
  ∧ pc[self] = \"a3a\"
  ∧ IF flag[Not(self)]
    THEN pc' = [pc EXCEPT ![self] = \"a3b\"
    ELSE pc' = [pc EXCEPT ![self] = \"cs\"
  ∧ UNCHANGED (flag, turn)
  \* remaining actions omitted
proc(self) Δ a0(self) \lor \ldots \lor a4(self)
Next Δ \exists self \in {0, 1} : proc(self)
Spec Δ Init \land \square[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**”)
Once again, our example

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

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

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 ⇒ 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
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()
Back to 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()
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()
```

**Summary: something went wrong in an execution**

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

* Schedule thread T0: __init__()  
  * Line 13: Initialize done2 to False  
  * Line 13: Initialize done1 to False  
  * Line 14: Initialize amount to 100000  
  * **Thread terminated**
* Schedule thread T1: T1()  
  * Preempted in T1()  
  * about to store 90000 into amount in line 2  
* Schedule thread T2: T2()  
  * Line 6: Set amount to 50000 (was 100000)  
  * Line 7: Set done2 to True (was False)  
  * **Thread terminated**
* Schedule thread T1: T1()  
  * Line 2: Set amount to 90000 (was 50000)  
  * Line 3: Set done1 to True (was False)  
  * **Thread terminated**
* Schedule thread T3: main()  
  * Line 11: Harmony assertion failed: 90000
Simplified model (ignoring main)

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

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

_init_

**init**

- amount = 100000

**T1a**: LOAD amount

**T1b**: SUB 10000

**T1c**: STORE amount

**T2a**: LOAD amount

**T2b**: DIV 2

**T2c**: STORE amount

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000

T1 loaded 100000

T2 loaded 100000

T1 got 90000

T1 stored 90000

T1 got 50000
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 in the state graph)

[Nemo:~/Documents/orenzo% harmony example.hny
 * Phase 1: convert Java program to bytecode
 * Phase 2: run the model checker (nworkers = 8)
   * 103 states (time 0.00s, mem=0.000GB)
 * Phase 3: analysis
 * **Safety Violation**
 * Phase 4: write results to example.hco
 * Phase 5: loading example.hco

----------------------------------------

**Summary: something went wrong in an execution**

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

* Schedule thread T0: __init__()
  * Line 13: Initialize done2 to False
  * Line 13: Initialize done1 to False
  * Line 14: Initialize amount to 100000
  * **Thread terminated**
* Schedule thread T1: T1()
  * Preempted in T1()
    about to store 90000 into amount in line 2
* Schedule thread T2: T2()
  * Line 6: Set amount to 50000 (was 100000)
  * Line 7: Set done2 to True (was False)
  * **Thread terminated**
* Schedule thread T1: T1()
  * Line 2: Set amount to 90000 (was 50000)
  * Line 3: Set done1 to True (was False)
  * **Thread terminated**
* Schedule thread T3: main()
  * Line 11: Harmony assertion failed: 90000
Harmony Output

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

Shortest path to assertion failure
Harmony’s VM State

Three parts:

- code (which never changes)
- values of shared variables
- states of each of the running threads
  - 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
- Local variables
  - Parameters (a.k.a. arguments)
  - Result
    - There is no `return` statement
  - Local variables
    - Declared in `var`, `let`, and `for` statements
<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</td>
<td>many operator precedence rules</td>
</tr>
<tr>
<td>use parentheses often</td>
<td></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>
<tr>
<td>...</td>
<td>...</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
  - \( \text{const } x = 3 \)
  - 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 \( \text{amount} \)

- \( T_1 \) wants to decrement \( \text{amount} \) by $10K
- \( T_2 \) wants to decrement \( \text{amount} \) by 50%

\[
\text{amount} := \text{amount} - 10,000; \\
\text{...}
\]

\[
\text{...}
\]

\[
\text{amount} := \text{amount} \times 0.5; \\
\text{...}
\]

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 so
- **Fairness**: equal chances to get into CS (uncommon in practice)
Critical Section

Shared memory access: must be serialized

\[ T_1 \]
\[ \ldots \]
\[ \text{CSEnter}() \]
\[ \quad \text{amount} := \text{amount} - 10,000; \]
\[ \text{CSExit}() \]
\[ \ldots \]

\[ T_2 \]
\[ \ldots \]
\[ \text{CSEnter}() \]
\[ \quad \text{amount} := \text{amount} \times 0.5; \]
\[ \text{CSExit}() \]
\[ \ldots \]

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
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...
Prelim Interlude
How many times will the value of result be printed?

First value(s)? Last value(s)?
How many times will the value of result be printed?

First value(s)? Last value(s)?
#include <stdio.h> /* declares printf() */
#include <unistd.h> /* declares fork() */

int main() {
    int i;
    int pid;
    int result = 0;
    for (i=0; i<2; i++) {
        pid = fork();
        result ++;
        printf("result = %d\n", result);
    }
    if (pid == 0) {
        printf("result = %d\n", result);
    }
    return 0;
}
How many times will the value of `result` be printed?

First value(s)? Last value(s)?