Non-Determinism

```
shared = True

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

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

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

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

Non-Determinism

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

(a) [code/prog1.hny] Sequential

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

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

```
[Nemo:~/Documents/harmony] lorenzo% harmony code/prog1.hny
* Phase 1: compile Harmony program to bytecode
* Phase 2: run the model checker (nworkers = 8)
    * 2 states (time 0.00s, mem=0.000GB)
* Phase 3: analysis
    * 2 components (0.00 seconds)
    * Check for data races
    * **No issues found**
* Phase 4: write results to code/prog1.hco
* Phase 5: loading code/prog1.hco
```

```
**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 amount

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

T_1

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

T_2

```
. . .
amount := amount * 0.5;
. . .
```

Memory	amount	100,000

What happens when T_1 and T_2 execute concurrently?

Non-Determinism

Might execute like this:

T_2

```
...
r2 := load from amount
r2 := 0.5 * r2
store r2 to amount
...
```

T_1

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

Memory	amount	40,000

Or viceversa: T_1 and then T_2

amount	45,000

Non-Atomicity

But might also execute like this:

T_1

. . .
r1 := load from amount
r1 := r1 - 10,000
store r1 to amount
. . .

T_2

. . .
r2 := load from amount

. . .

r2 := 0.5 * r2
store r2 to amount
. . .

Memory — amount: 50,000

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!

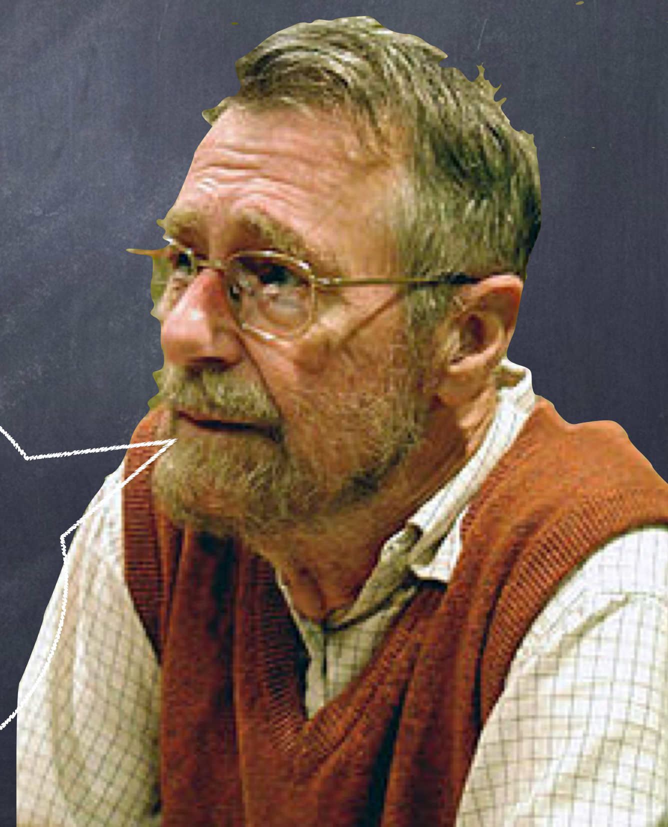
Race Conditions: Hard to Debug

- 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

Dutch Wisdom

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!

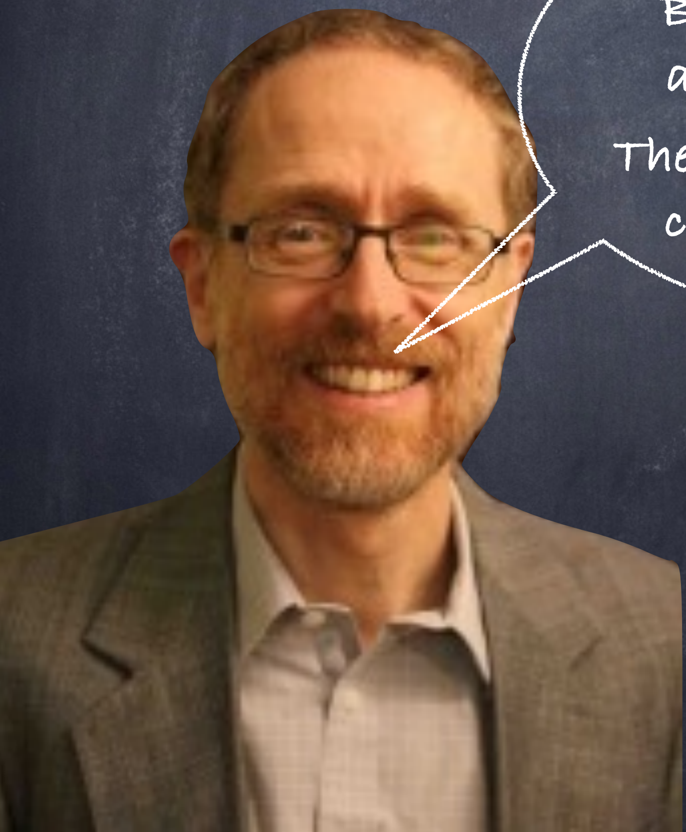


Dutch Wisdom

True!

But there is testing
and then testing...

They submit their code,
confident that it is
correct...



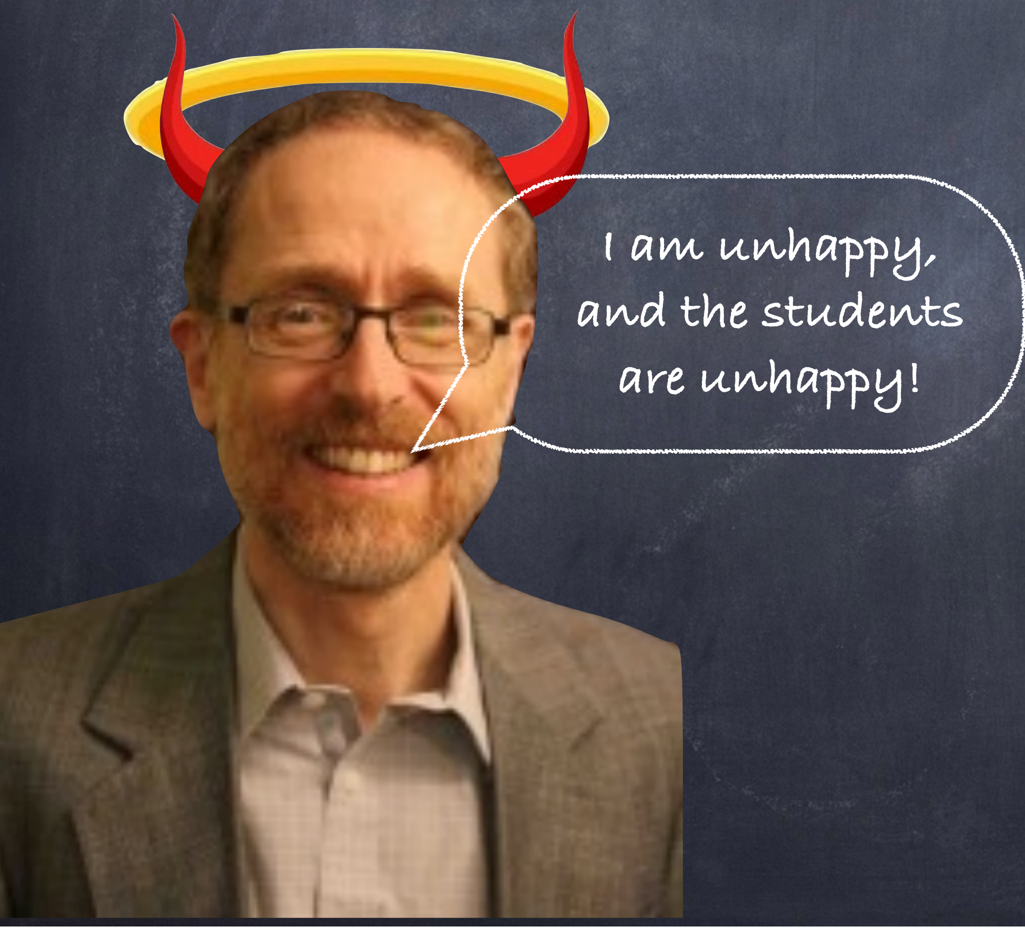
Dutch Wisdom

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")

Dutch Wisdom



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!

Dutch Wisdom

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

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

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

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

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

Spin

```
--algorithm Peterson {
  variables flag = [i \in {0, 1} |-> FALSE
     \* Declares the global variables flag
     \* flag is a 2-element array with ini
  fair process (proc \in {0,1}) {
    \* Declares two processes with identifi
    \* The keyword fair means that no proce
    \* 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) \/ (tur
   cs:  skip ;  \* the critical section
   a5:  flag[self] := FALSE
                                         }
```

PlusCal

TLA+

VARIABLES $flag, turn, pc$

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

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

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

* remaining actions omitted

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

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

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

Enter Harmony

- A new concurrent programming language
  - heavily based on Python syntax to reduce learning curve for many
- A new underlying virtual machine, quite different from any other
  - it tries all possible executions of a program, until it finds a problem (if any)

(this is called "model checking")

Once again, our example

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

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

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

```
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 done 2):
    pass
```

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()
```

Assertion: useful to check properties

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()
```

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

```
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

```
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)

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()
```

```
[Nemo:~/Documents/harmony/mycode] lorenzo% harmony example.hny
* Phase 1: compile Harmony 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 T2: T2()
    * Preempted in T2()
      about to store 50000 into amount in line 6
* Schedule thread T1: T1()
    * Line 2: Set amount to 90000 (was 100000)
    * Line 3: Set done1 to True (was False)
    * **Thread terminated**
* Schedule thread T2: T2()
    * Line 6: Set amount to 50000 (was 90000)
    * Line 7: Set done2 to True (was False)
    * **Thread terminated**
* Schedule thread T3: main()
    * Line 11: Harmony assertion failed: 50000
```

Back to our example

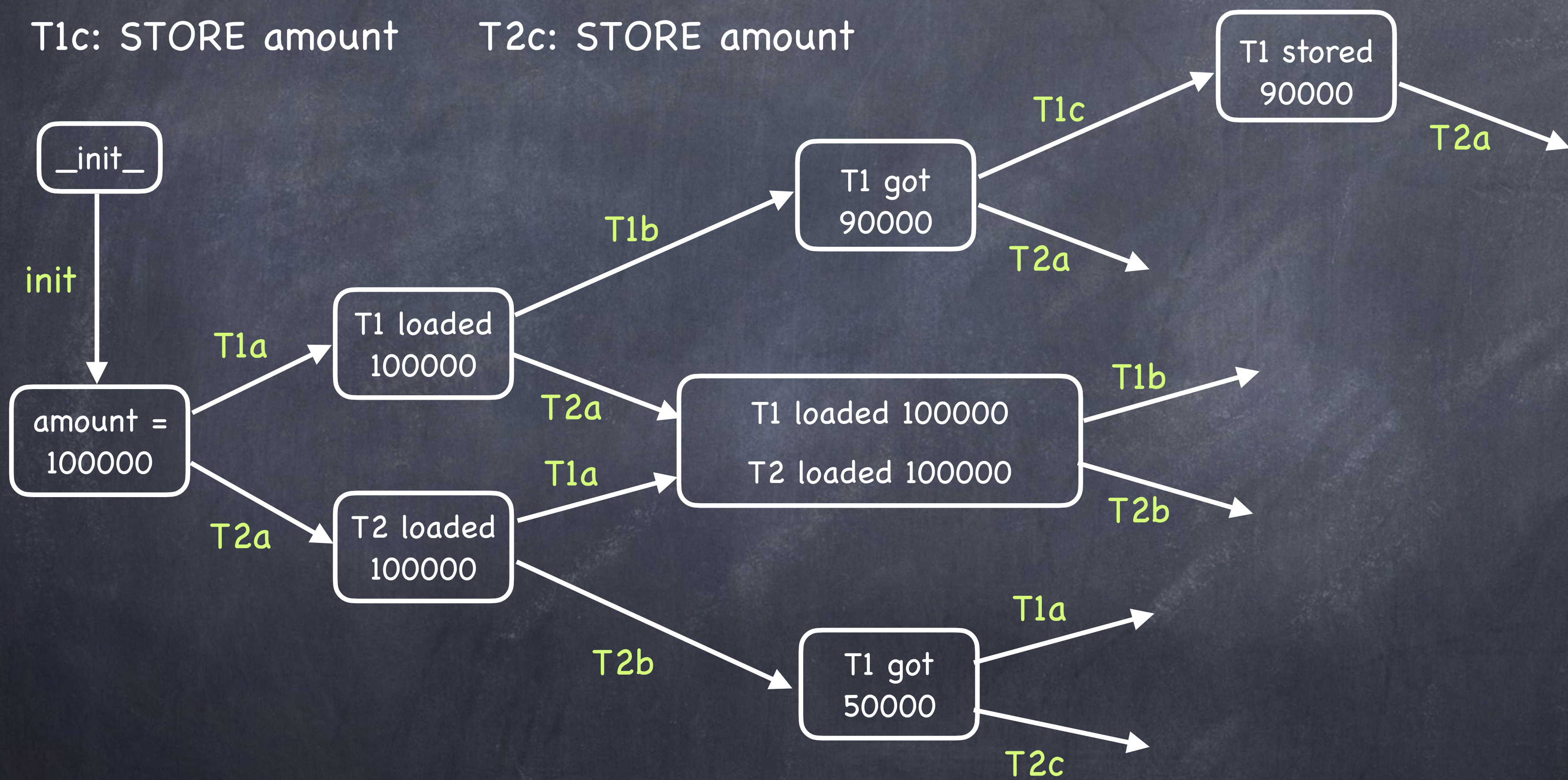
```
def T1():
    amount −= 10000
    done1 = True

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()
```

```
[Nemo:~/Documents/harmony/mycode] lorenzo% harmony example.hny
* Phase 1: compile Harmony 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
```

Simplified model (ignoring main)

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

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



Harmony Output

```
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()
```

```
[Nemo:~/Documents/harmony/mycode] lorenzo% harmony example.hny
* Phase 1: compile Harmony 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

#states in the
state graph

```
[Nemo:~/Docum                   orenzo% harmony example.hny
* Phase 1: co              program to bytecode
* Phase 2:     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

```
[Nemo:~/Documents/harmony/mycode] lorenzo% harmony example.hny
* Phase 1: compile Harmony program to bytecode
* Phase 2: run the model checker (nworkers = 8)
    * 103 states (time 0.00s, mem
* Phase 3: analysis
    * **Safety Violation**
* Phase 4: write results to ex
* 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
```

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

Harmony Output

```
[Nemo:~/Documents/harmony/mycode] lorenzo% harmony example.hny
* Phase 1: compile Harmony program to bytecode
* Phase 2: run the model checker (nworkers = 8)
    * 103 states (time 0.00s, mem=0.000GB)
* Phase 3: analysis

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
```

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

Harmony != Python

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

I/O in Harmony

- Input
  - choose expression
    - $x = \text{choose}(\{1,2,3\})$
    - 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

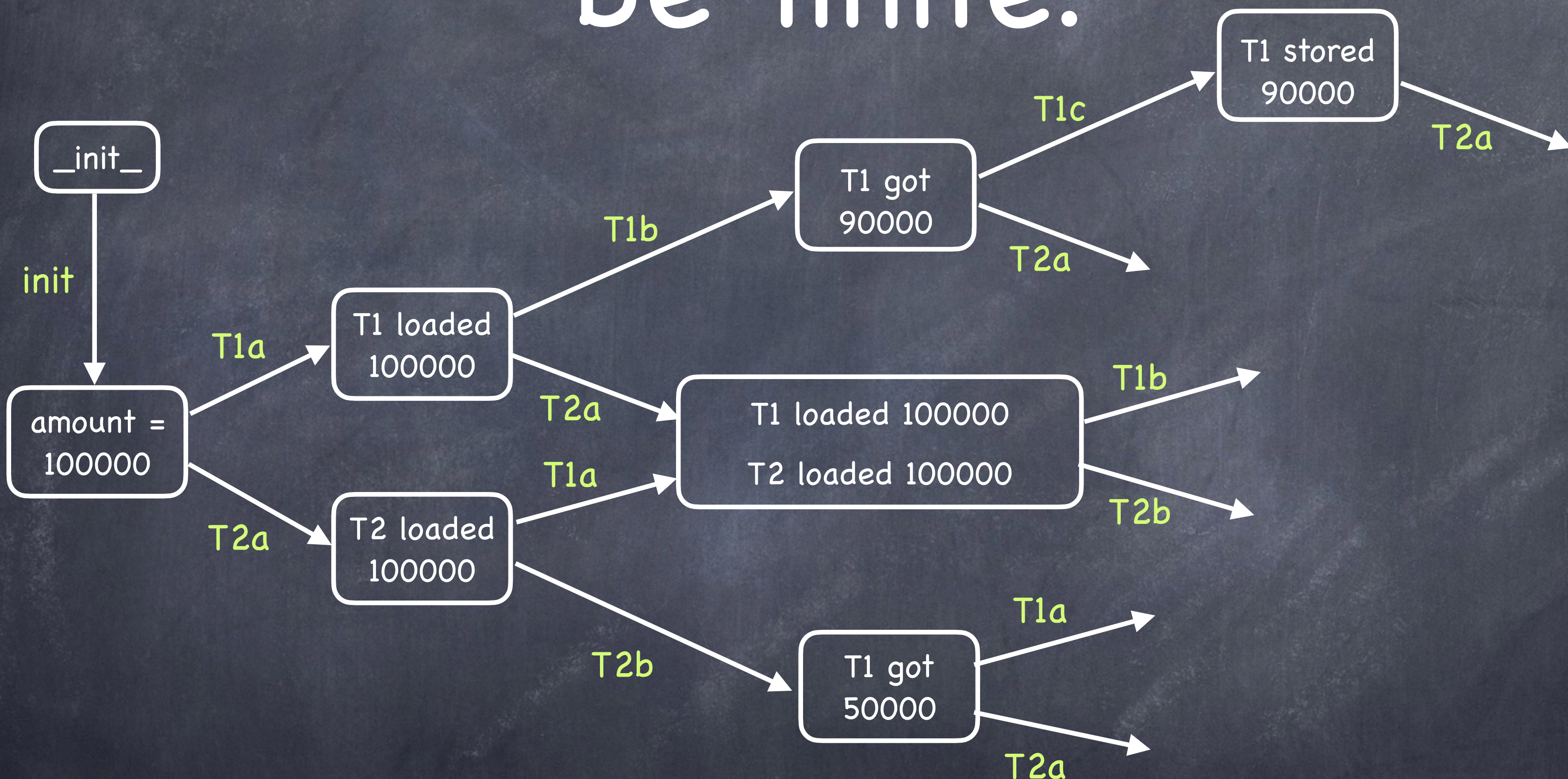
y

- 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%

T_1

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

T_2

```
. . .
amount := amount * 0.5;
. . .
```

Memory	amount	100,000

How to "serialize" these executions?

Critical Section

Shared memory access: must be serialized

T_1

```
...
CSEnter()
    amount := amount - 10,000;
CSExit()
...
```

T_2

```
...
CSEnter()
    amount := amount * 0.5;
CSExit()
...
```

- 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)

Critical Section

Shared memory access: must be serialized

T_1

```
. . .
CSEnter()
   amount := amount - 10,000;
CSExit()
. . .
```

T_2

```
. . .
CSEnter()
     amount := amount * 0.5;
CSExit()
. . .
```

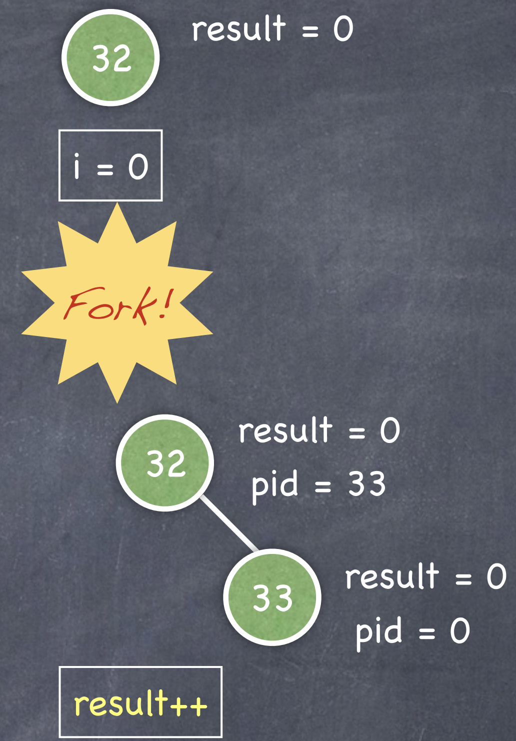
- Goals
  - Mutual exclusion: at most 1 thread in CS at any time
  - Progress: if any threads want to enter the CS, at least one does

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

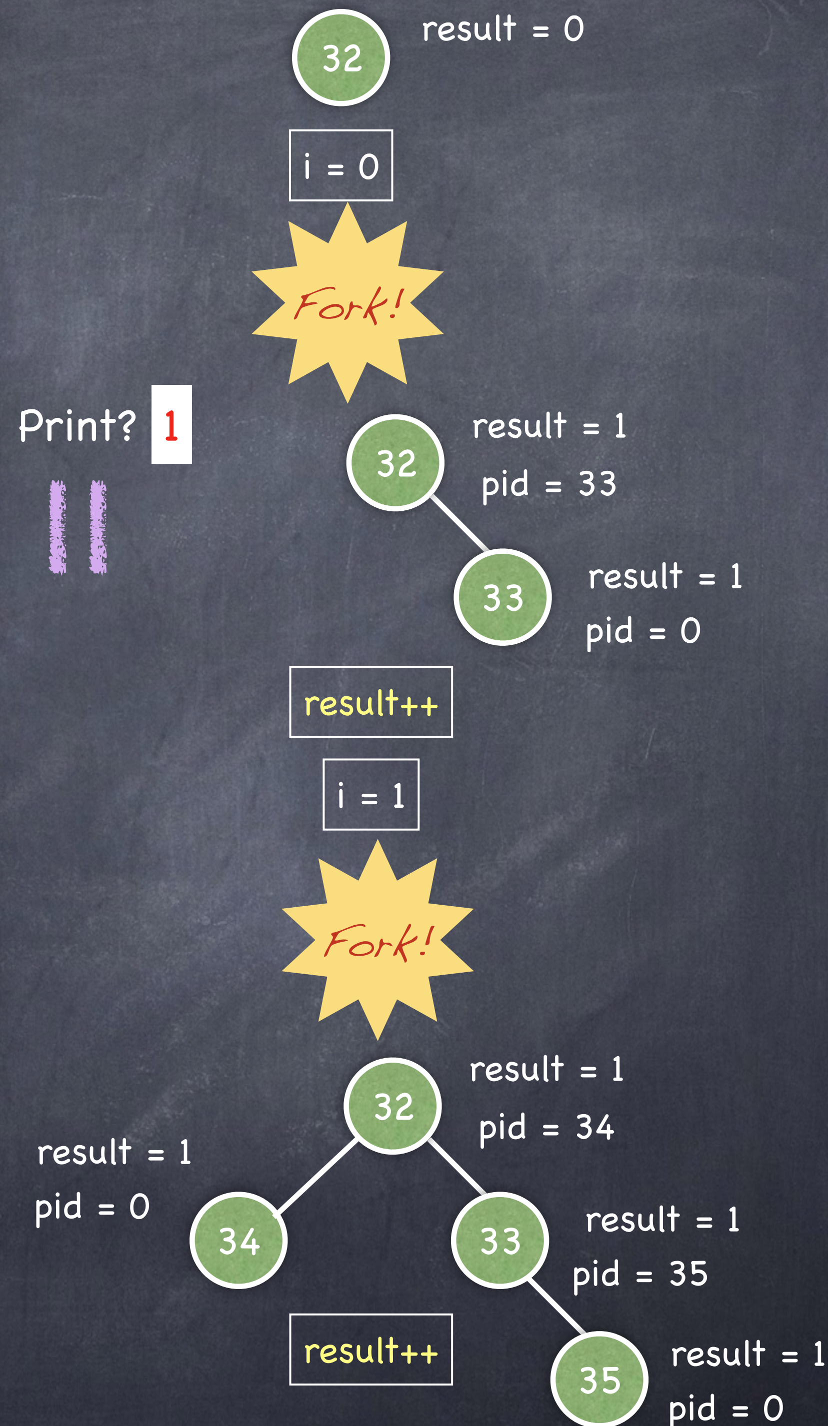
```c
#include <stdio.h>  /* declares printf() */
#include <unistd.h> /* declares fork() */


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)?

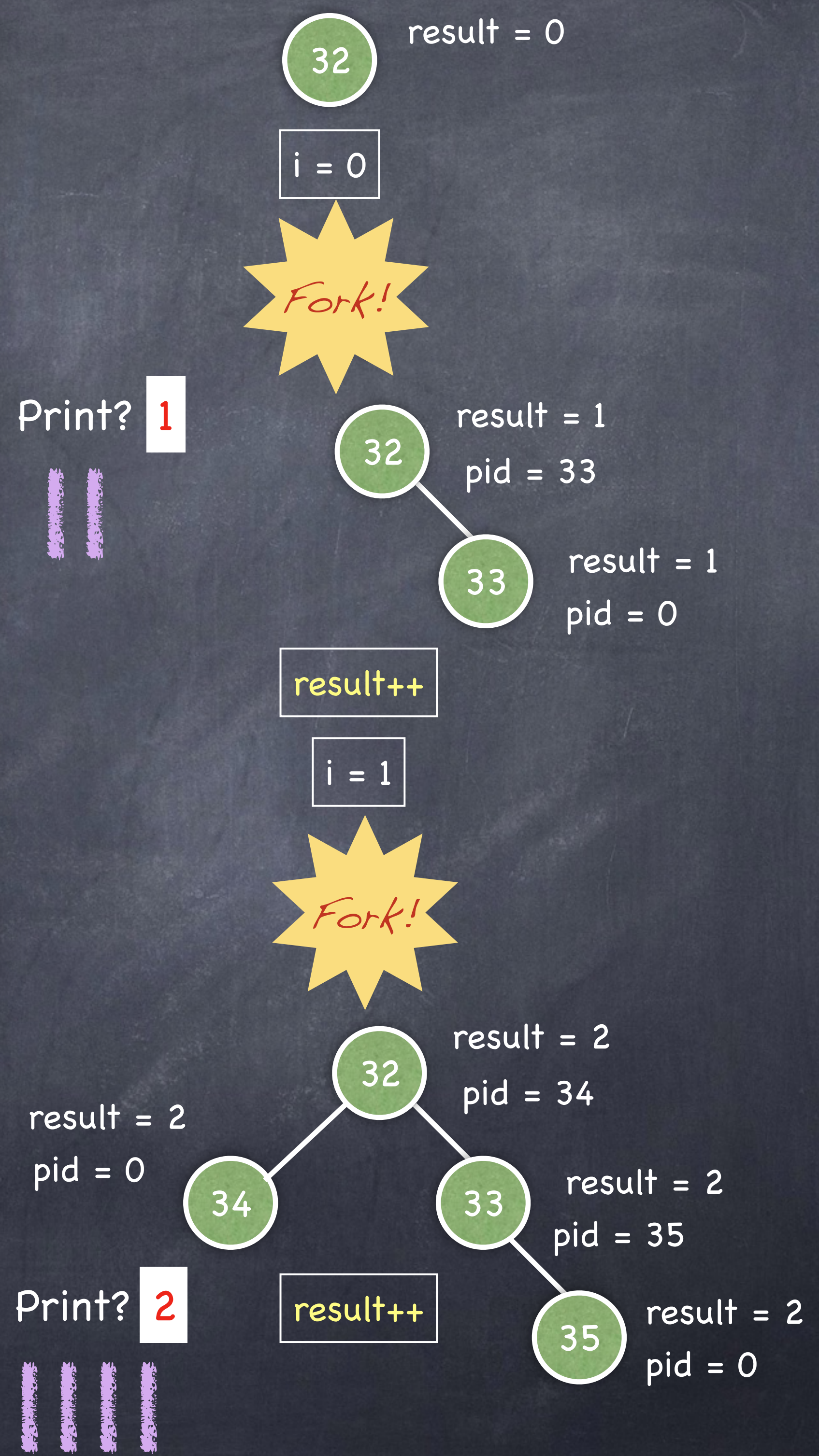
```c
#include <stdio.h>  /* declares printf() */
#include <unistd.h> /* declares fork() */


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)?

```c
#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)?

```c
#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)?