Concurrent Programming:
Critical Sections & Locks

An OS is a concurrent program

- The "kernel contexts" of each of the processes share many data structures
  - ready queue, wait queues, file system cache, and much more
- Interrupt handlers also access those data structures!
- Need to learn how to share

Lectures Outline - I

- What is the problem?
  - no determinism, no atomicity
- What is the solution?
  - some form of lock
- How to implement locks?
  - there are multiple ways

Lectures Outline - II

- How to specify concurrent problems?
  - atomic operations
- How to construct concurrent code?
  - invariants
- How to test concurrent programs?
  - comparing behaviors

Concurrent Programming is Hard

- Concurrent programs are non-deterministic
  - run twice with same input, get different answers
  - one time it works, another it fails
- Program statements are executed non-atomically
  - $x += 1$ compiles to something like
    - LOAD x
    - ADD 1
    - STORE x

Non-Determinism

```
shared = True

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

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

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

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

Non-Determinism

```
shared = True

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

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

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

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed

Non-Determinism

```
shared = True

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

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

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

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed

Non-Determinism

```
shared = True

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

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

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

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed

Non-Atomicity

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-Atomicity

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
  - timing dependent
- Compiler and processor hardware can reorder instructions

Dutch Wisdom

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!

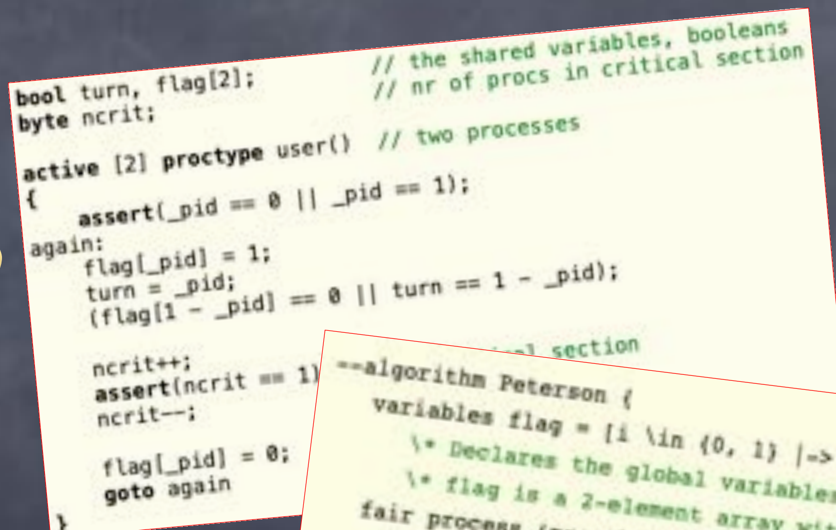
Dutch Wisdom

Handwritten proofs are just as likely to have bugs as programs… or even more likely, as you can't test unwritten proofs!

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

Dutch Wisdom

Examples of existing tools



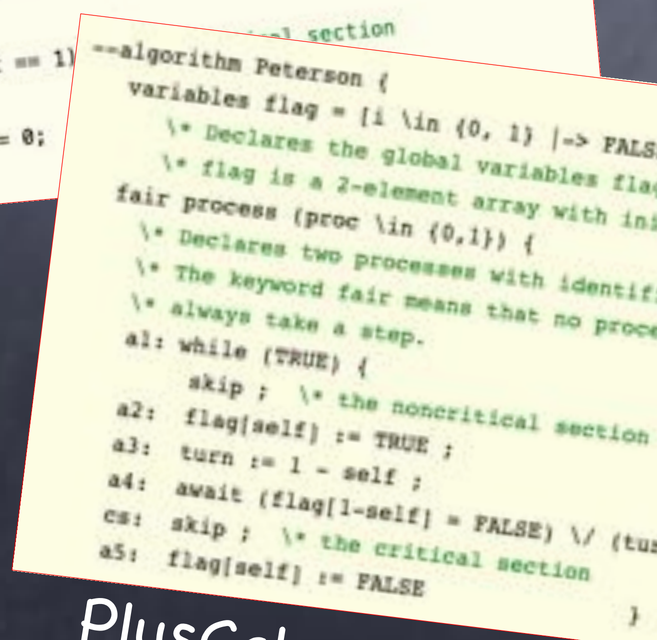
```
bool turn, flag[2];          // the shared variables, booleans
byte ncrit;                  // nr of procs in critical section

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
}
```

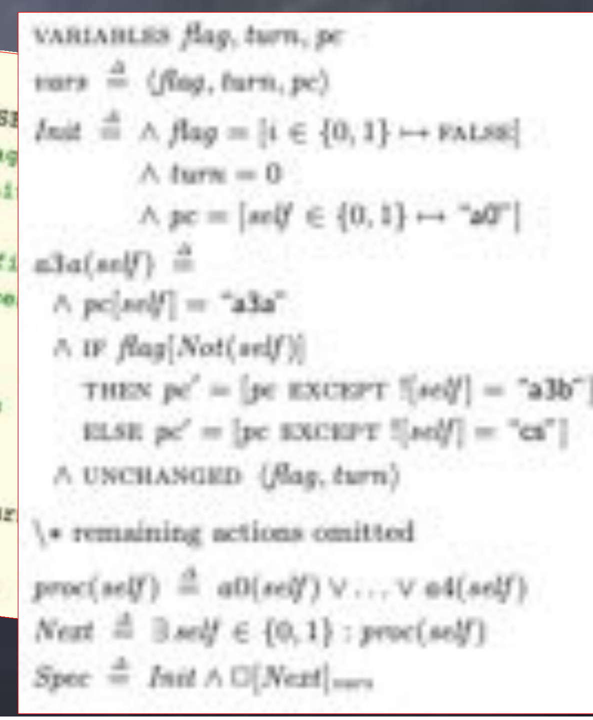
Spin



```
--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 ≜ ⟨flag, turn, pc⟩
Init ≜ ∧ flag = [i ∈ {0, 1} ↦ FALSE]
       ∧ turn = 0
       ∧ pc = [self ∈ {0, 1} ↦ "a0"]
a3a(self) ≜
  ∧ 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) ≜ a0(self) ∨ . . . ∨ a4(self)
Next ≜ ∃ self ∈ {0, 1} : proc(self)
Spec ≜ Init ∧ □[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")