PROVING THINGS ABOUT CONCURRENT PROGRAMS

Lecture 21 – CS2110 – Fall 2009

Overview

- Last time we looked at techniques for proving things about recursive algorithms
  - We saw that in general, recursion matches with the notion of an inductive proof
- How can one reason about a concurrent algorithm?
  - We still want proofs of correctness
  - Techniques aren’t identical but we do use induction

Safety and Liveness

- When a program uses multiple threads, we need to worry about many things
  - Are concurrent memory accesses correctly synchronized?
  - Do the threads "interfere" with one-another?
  - Can a deadlock arise?
  - What if some single thread gets blocked but the others continue to run?
  - Could an infinite loop arise in which threads get stuck running, but making no progress?

Safety and Liveness

- Leslie Lamport suggested that we think about the question in terms of *safety* and *liveness*
  - A program is *safe* if nothing bad happens. The guarantee that concurrently accessed memory will be locked first is a *safety property.*
    - The property is also called **mutual exclusion**
  - A program is live if good things eventually happen. The guarantee that all threads get to make progress is a *liveness property*

Proper synchronization

- Consider a program with multiple threads in it
  - Perhaps threads T1 and T2
  - They share some objects

- First, we need to ask if the shared objects are **thread safe**
  - Every access protected by synchronized() { … }

Critical section example

Thread A: Swap(X[i], Y[j])

- Suppose i=3, j=7
  1. tmp = X[i];
  2. X[i] = X[j];
  3. X[j] = tmp;

Thread B : Swap(X[i], Y[j])

- …. same indicies
  4. tmp = X[i];
  5. X[i] = X[j];
  6. X[j] = tmp;

- Two swaps on the same items… so at the end we should be back where we started, right?

Critical section example

Thread A: Swap(X[i], Y[j])

- Suppose i=3, j=7

1. tmp = X[i];
2. X[i] = X[j];
3. X[j] = tmp;

Thread B : Swap(X[i], Y[j])

- …. same indicies

4. tmp = X[i];
5. X[i] = X[j];
6. X[j] = tmp;

- What if thread B runs (entirely) in between the last two lines of thread A?

Critical section example

Thread A: Swap(X[i], Y[j])

- Suppose i=3, j=7
  1. tmp = X[i];
  2. X[i] = X[j];
  3. X[j] = tmp;

Thread B : Swap(X[i], Y[j])

- …. same indicies
  4. tmp = X[i];
  5. X[i] = X[j];
  6. X[j] = tmp;

- We end up with X[i] = X[j] and X[j]'s old value is lost!
- With other values for i,j and other execution orderings can lose X[j] or cause other kinds of problems

Hardware needs synchronization too!

- As we saw last week, the hardware itself may malfunction if we omit synchronization!
  - Modern CPUs sometimes reorder operations to execute them faster, usually because some slow event (like fetching something from memory) occurs, and leaves the CPU with time to kill
  - So it might look ahead and find some stuff that can safely be done a bit early

Hardware needs synchronization too!

- Without synchronization locks, if a thread updates objects the thread itself always sees the exact updates in the order they were done
- But other threads on other cores could see them out of order and could see some updates but not others

Interleavings

- Suppose that a program correctly locks all accesses to shared objects
- Would it now be safe?
- Issue that arises involves *interleavings*

Interleavings

- Suppose threads A and B are executing
- A updates Object X, and then B changes X
  - Was this order “enforced by the program” or could it be an accident of thread scheduling?
- Ideally, when threads interact we would like to control ordering so that it will be predictable

Determinism

- A program is *deterministic* if it produces the identical results every time it is run with identical input
  - This is desirable

- A program is *non deterministic* if the same inputs sometimes result in different outcomes
  - This is confusing and can signal problems

Linearizability

- Concept was proposed by Wing and Herlihy
  - Start with your concurrent program
  - But prove that it behaves just like some non-concurrent program that does the same operations in some “linear” order
    - Idea behind proof: if the effect of two executions is the same, then we can treat them as equivalent
- Program is concurrent yet acts deterministic
- Not all programs are linearizable

We also worry about Deadlock

- Deadlock occurs if two or more threads are unable to execute because each is waiting for the other to do something, and both are blocked

- This is typically a buggy situation and hence we also need to prove that our concurrent code can’t deadlock

Deadlock

- Recall from last week

- Deadlock depends on four conditions
  - A wait-for cycle
  - Locks that are held until the thread finishes what it wants to do, not released
  - No preemption of locks
  - Mutual exclusion

Example: Deadlock avoidance

- Suppose that threads acquire locks in some standard order. *Thm: deadlock cannot occur!*
  - Slightly oversimplified proof: A deadlock means that there is some cycle of threads A, B…. T each waiting for the next to take some action.
  - Consider thread A and assume A holds lock X_a.
    - A is waiting on B: A wants a lock X_b and B holds that lock.
    - Now look at B: it holds X_b and wants X_c.
    - We eventually get to thread T that holds X_t and wants X_a
    - But per our rules $X_a < X_b < \dots X_t < X_a$: a contradiction! QED
  - Notice that this is similar to an inductive argument

Induction connection?

- Base case focuses on two threads, A and T
  - A is holding X_A and wants X_T
  - T is holding X_T and will wait for A
  - But T is violating policy. So we can't deadlock with two threads
- Induction case: assume no deadlocks with n-1 threads. Show no deadlocks with n threads.
  - We won't write this out in logic, but we could.

Paris traffic circles: Deadlock in action

- Paris has a strange rule at some traffic circles: *priorité a droite*
- Traffic circles around, say, the Arc de Triomphe
- Roads enter from the right
- You must yield to let them enter

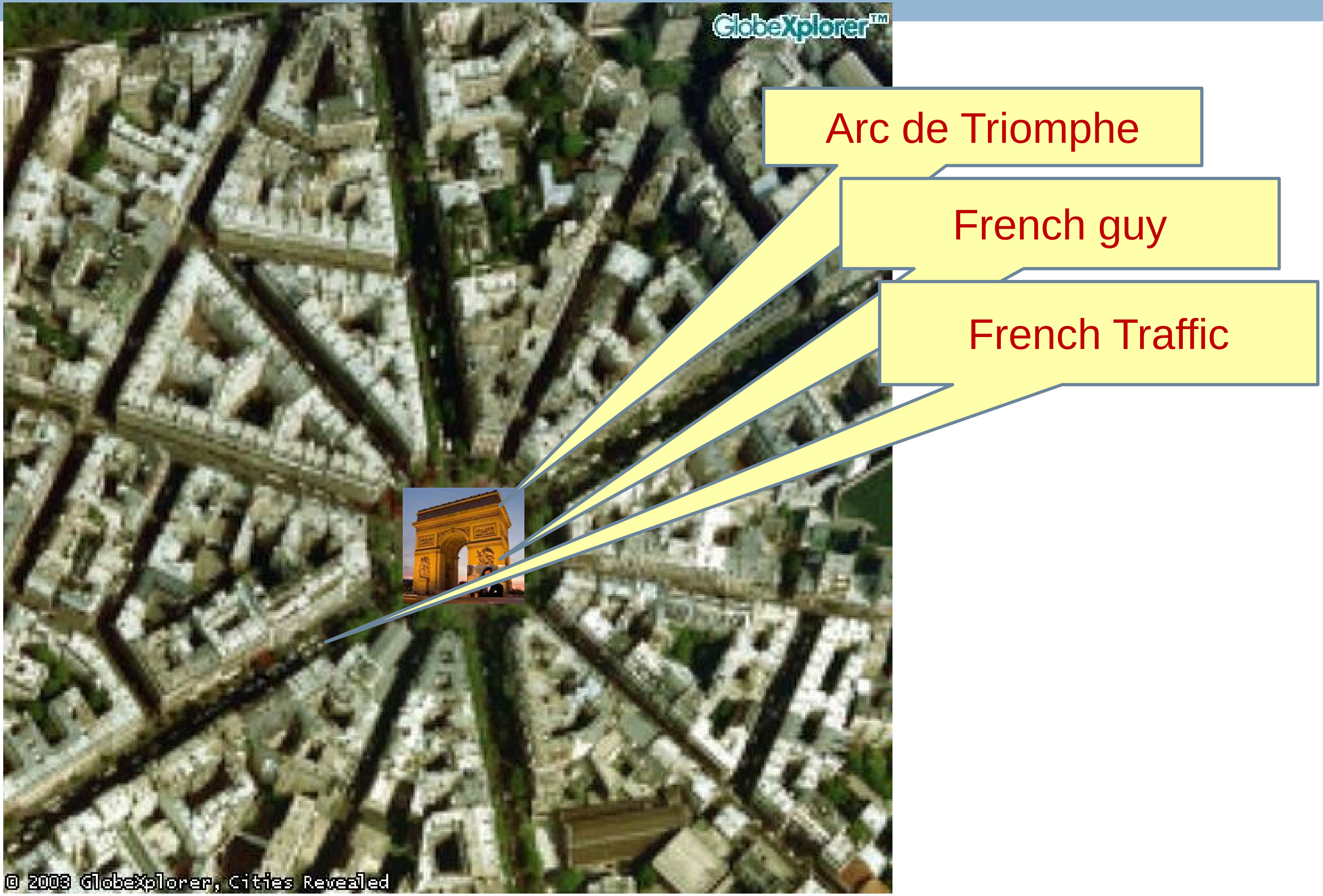
Paris traffic circle: *priorité a droite*

- An issue at Place d'Etoile and Place Victor Hugo (rest of France uses *priorité a gauche*)

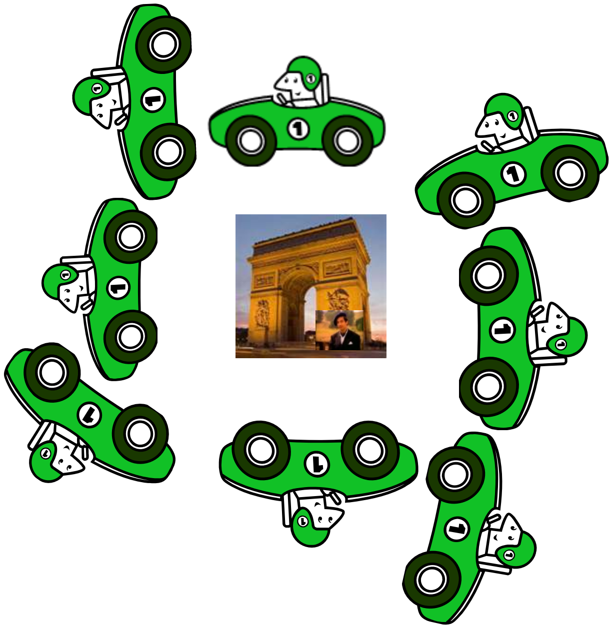
- Think of cars as threads and "space" as objects
  - If thread A occupies a space that thread B wishes to enter, then B waits for A
  - Under this rule, deadlocks can form!

- To see this, look for a wait-for cycle

Why is *priorité a droite* a bad rule?

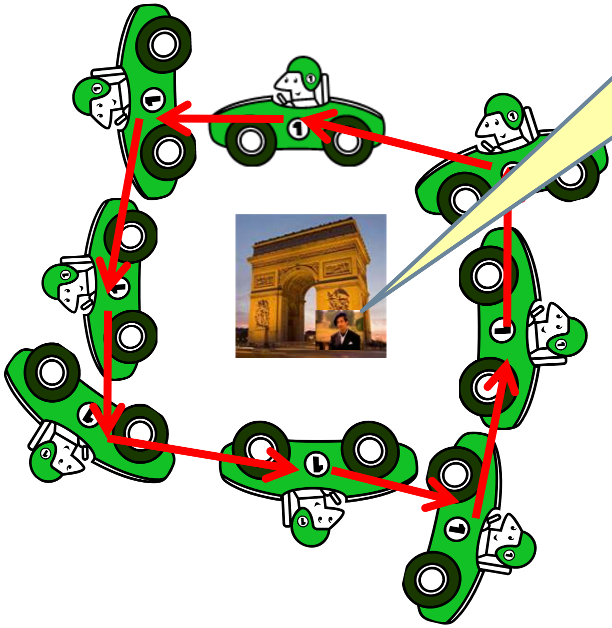


Why is *priorité a droite* a bad rule?



Why is *priorité a droite* a bad rule?

Ooh la la! Quel catastrophe!

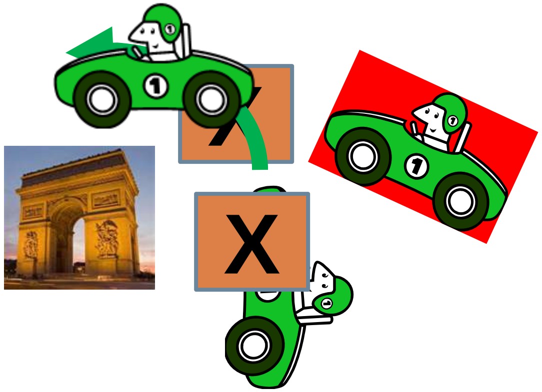


But why is this specific to *priorité a droite*?

- With *priorité a gauche* cars already in the circle have priority over cars trying to enter
- Cars can drive around the circle until each car gets to its desired exit road and the traffic drains away
  - In fact can drive around and around if they like
  - Deadlock can’t arise!

Inductive proof?

- Again, lends itself to an inductive proof
- Here's the key step in graphical form:
  - Assume we are not yet deadlocked: there is at least one space "X" free on the traffic circle
  - Red and Green cars both want to advance into X
  - Green is on the left, so it wins
  - This leaves space behind it



As a proof

- Two base cases
  - Traffic circle is “fully populated”.
    - Then traffic can rotate around circle until cars reach their exit streets and leave
  - Traffic circle has at least one gap
    - Priority-a-gauche ensures that the in-circle traffic will claim it, not the car contending to enter from right

As a proof

- Inductive case
  - Assumes that “chains” of n-1 cars are deadlock free
  - Add one car
    - If you add it in the circle, it waits for the car in front to move (which it will, by induction), then follows it
    - If you add it outside the circle, it can only enter if there is no contention with any car in the circle

- We conclude: the circle itself won’t deadlock!

But are cars happy?

- A car trying to enter might have bad luck and wait… forever!
  - This is called « starvation »

Starvation

- We say that a thread **starves** if it can't execute
  - A common reason: some thread locks a resource but forgets to unlock it
  - Not a deadlock because only one thread is stuck

What did this example show?

- We can sometimes prevent deadlock by controlling the "order" that contending threads grab resources
  - Priorite a gauche is such a rule.
  - But this also creates risk of starvation
- Ensuring that a system is both deadlock and starvation free requires clever design

Recap

- To prove a concurrent program correct we need to
  - Prove that the shared memory is accessed safely
  - Prove that threads can make useful progress
    - No deadlocks or livelocks or starvation
  - Guarantee determinism (optional, but useful)

- In practice this is very hard to do because of the vast number of possible interleavings

Debugging concurrent programs

- When we add threads to a program, or create a threaded program, debugging becomes more challenging
  - Without threads we think only about the “straight line” execution of our code
  - With threads need to think about all the orderings that can arise as they get scheduled

Bugs in concurrent programs

- In addition to regular kinds of bugs they often have bugs specific to concurrency!
  - Non-determinism and race conditions
  - Deadlock, livelock, starvation
  - Harder to reason about

Bugs in concurrent programs

- Bruce Lindsay once suggested that there are two kinds of bugs
  - Bohrbugs are like the Bohr model of the nucleus: we can track them down and exterminate them
    - Most deterministic, non-concurrent programs only have Bohrbugs and this is a good thing
  - Heisenbugs are hard to pin down: the closer you look the more they shift around, like a Heisenberg model of the atomic nucleus (a “cloud”)

Bugs in concurrent programs

- Concurrent programs often have latent Heisenbugs
  - Something that happened a while ago was the case
  - And the thread scheduling order may determine when you actually see the crash!

Bugs in concurrent programs

- Concurrent programs notorious for Heisenbugs
- You tend to focus on their eventual effect
  - But that was the *symptom,* not the cause!
  - You work endlessly but aren't actually even looking at the thing that caused the problem!
- And the debugger might cause the problem to shift around

Adding threads to unsafe code

- Modern fad: Adding threading to a program so that it can benefit from multicore hardware
  - Start with a program that was built without threads. Then introduce threads and synchronization
  - If you weren't the original designer, this is a risky way to work!

Risky style?
I am liking concurrency very much!

Our recommendations?

- Threads are an unavoidable evil
  - We need them for performance and responsiveness
  - But they make it (much) harder to prove things about our programs
  - Must use them cautiously and in very controlled ways

- Linearizability can greatly simplify analysis
- Use inductive style of proofs to reason about chains of threads that wait for one-another