Dining Philosophers.... Then Deadlocks (part I)

Ken Birman

Dining Philosophers

- A problem that was invented to illustrate some issues related to synchronization with objects
- Our focus here is on the notion of sharing resources that *only one user at a time* can own
  - Such as a keyboard on a machine with many processes active at the same time
  - Or a special disk file that only one can write at a time (bounded buffer is an instance)

Dining Philosopher's Problem

- Dijkstra

- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time

Idea is to capture the concept of multiple processes competing for limited resources

---

Rules of the Game

- The philosophers are very logical
  - They want to settle on a shared policy that all can apply concurrently
  - They are hungry: the policy should let everyone eat (eventually)
  - They are utterly dedicated to the proposition of equality: the policy should be totally fair

---

What can go wrong?

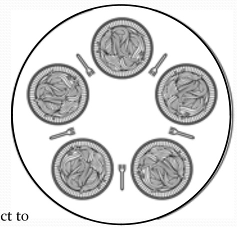
- Lots of things! We can give them names:
  - Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
    - Fairness: even if nobody starves, should we worry about policies that let some eat more often than others?
  - Deadlock: A policy that leaves all the philosophers "stuck", so that nobody can do anything at all
  - Livelock: A policy that makes them all do something endlessly without ever eating!

A flawed conceptual solution

```
Const int N = 5;


Philosopher i (0, 1, .. N-1)

do {
  think();
  take_fork(i);
  take_fork((i+1)%N);
  eat(); /* yummy */
  put_fork(i);
  put_fork((i+1)%N);
} while (true);
```

Coding our flawed solution?

```
Shared: semaphore fork[5];
Init: fork[i] = 1 for all i=0 .. 4

Philosopher i

do {
   fork[i].acquire();
   fork[i+1].acquire();

   /* eat */

   fork[i].release();
   fork[i+1].release();

   /* think */
} while(true);
```



Oops! Subject to deadlock if they all pick up their "right" fork simultaneously!

Dining Philosophers Solutions

- Set table for five, but only allow four philosophers to sit simultaneously
- Asymmetric solution
  - Odd philosopher picks left fork followed by right
  - Even philosopher does vice versa
- Pass a token
- Allow philosopher to pick fork only if both available

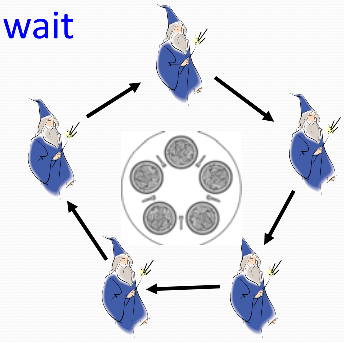
Why study this problem?

- The problem is a cute way of getting people to think about deadlocks
- Our goal: understand properties of solutions that work and of solutions that can fail!

Cyclic wait

- How can we "model" a deadlocked philosophers state?
  - Every philosopher is holding one fork
  - ... and each is waiting for a neighbor to release one fork
- We can represent this as a graph in which
  - Nodes represent philosophers
  - Edges represent waiting-for

Cyclic wait



Cyclic wait

- We can define a system to be in a deadlock state if
  - There exists ANY group of processes, such that
  - Each process in the group is waiting for some other process
  - And the wait-for graph has a cycle
- Doesn't require that every process be stuck... even two is enough to say that the system as a whole contains a deadlock ("is deadlocked")

Four Conditions for Deadlock

- **Mutual Exclusion**
  - At least one resource must be held is in non-sharable mode
- **Hold and wait**
  - There exists a process holding a resource, and waiting for another
- **No preemption**
  - Resources cannot be preempted
- **Circular wait**
  - There exists a set of processes {P_1, P_2, ... P_N}, such that
    - P_1 is waiting for P_2, P_2 for P_3, ... and P_N for P_1

All four conditions must hold for deadlock to occur

What about livelock?

- This is harder to express
  - Need to talk about making "meaningful progress"
- In CS414 we'll limit ourselves to deadlock
  - Detection: For example, build a graph and check for cycles (not hard to do)
  - Avoidance – we'll look at several ways to avoid getting into trouble in the first place!
- As it happens, livelock is relatively rare (but you should worry about it anyhow!)

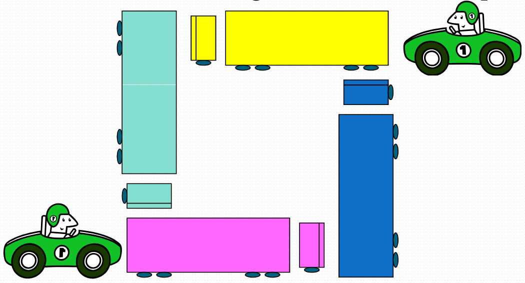
Real World Deadlocks?

- Truck A has to wait for truck B to move

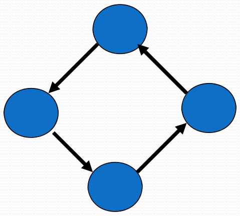
- Not deadlocked

Real World Deadlocks?

- Gridlock (assuming trucks can't back up)

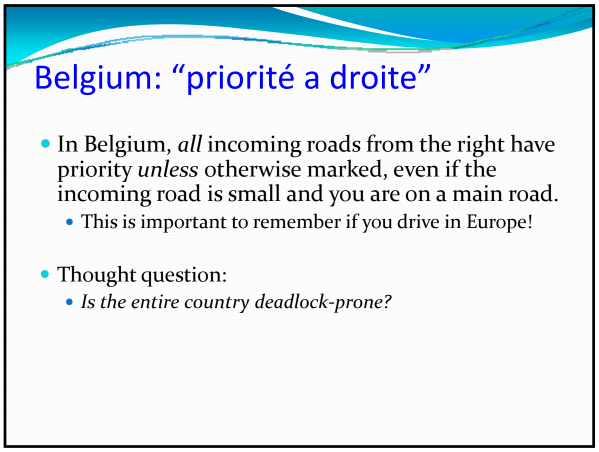


Real World Deadlocks?



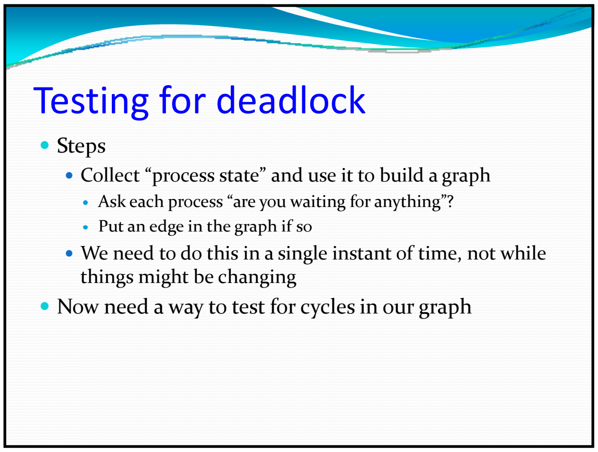
The strange story of "priorité a droite"

- France has many traffic circles…
  - … normally, the priority rule is that a vehicle trying to enter must yield to one trying to exit
  - Can deadlock occur in this case?
- But there are two that operate differently
  - Place Etoile and Place Victor Hugo, in Paris
  - What happens in practice?



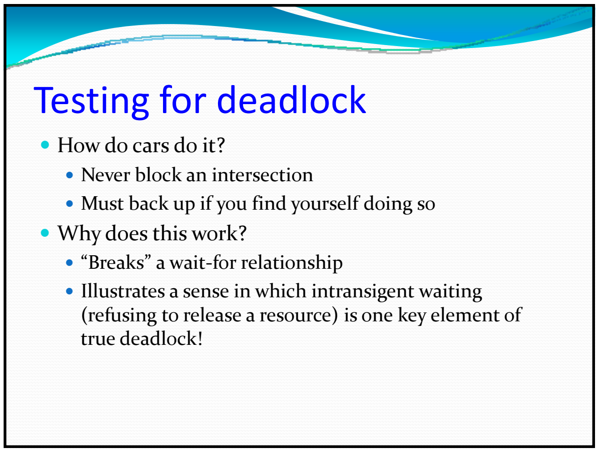
Belgium: “priorité a droite”

- In Belgium, *all* incoming roads from the right have priority *unless* otherwise marked, even if the incoming road is small and you are on a main road.
  - This is important to remember if you drive in Europe!
- Thought question:
  - *Is the entire country deadlock-prone?*



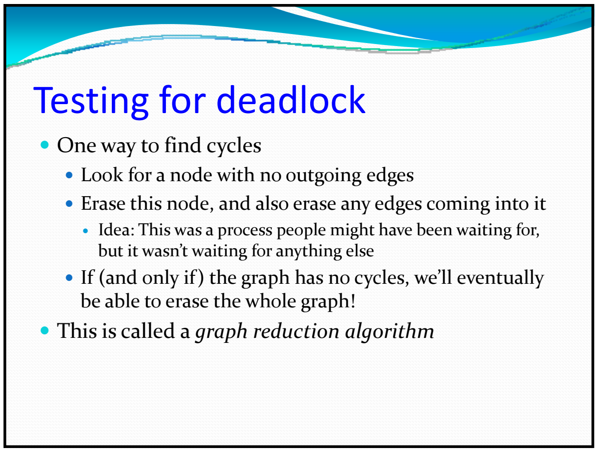
Testing for deadlock

- Steps
  - Collect “process state” and use it to build a graph
    - Ask each process “are you waiting for anything”?
    - Put an edge in the graph if so
  - We need to do this in a single instant of time, not while things might be changing
- Now need a way to test for cycles in our graph



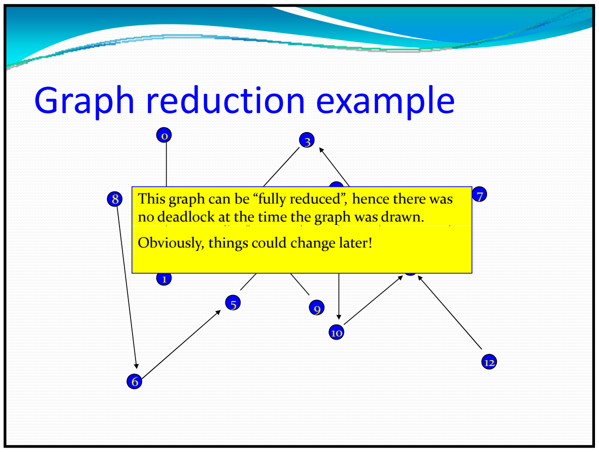
Testing for deadlock

- How do cars do it?
  - Never block an intersection
  - Must back up if you find yourself doing so
- Why does this work?
  - “Breaks” a wait-for relationship
  - Illustrates a sense in which intransigent waiting (refusing to release a resource) is one key element of true deadlock!



Testing for deadlock

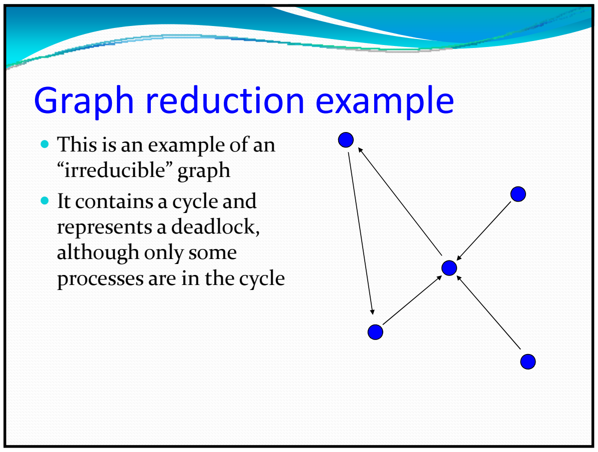
- One way to find cycles
  - Look for a node with no outgoing edges
  - Erase this node, and also erase any edges coming into it
    - Idea: This was a process people might have been waiting for, but it wasn’t waiting for anything else
  - If (and only if) the graph has no cycles, we’ll eventually be able to erase the whole graph!
- This is called a *graph reduction algorithm*



Graph reduction example

This graph can be “fully reduced”, hence there was no deadlock at the time the graph was drawn.

Obviously, things could change later!



Graph reduction example

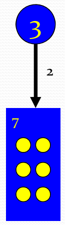
- This is an example of an “irreducible” graph
- It contains a cycle and represents a deadlock, although only some processes are in the cycle

Graph Reduction

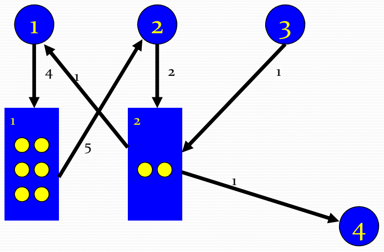
- Given a “state” that our system is in, tells us how to determine whether the system is deadlocked
- But as stated only works for processes that wait for each other, like trucks in our deadlock example
- What about processes waiting to acquire locks?
  - Locks are “objects”
  - Our graphs don’t have a notation for this…

Resource-wait graphs

- With two kinds of nodes we can extend our solution to deal with resources too
- A process: P_3 will be represented as:
  - A big circle with the process id inside it
- A resource: R_7 will be represented as:
  - A resource often has multiple identical units, such as “blocks of memory”
  - Represent these as circles in the box
- Arrow from a process to a resource: “I want *k* units of this resource.” Arrow to a process: this process holds *k* units of the resource
  - P_3 wants 2 units of R_7



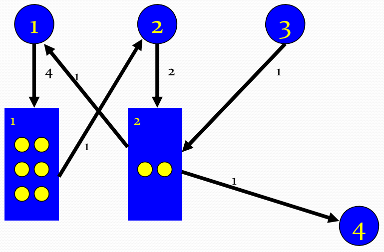
Resource-wait graphs



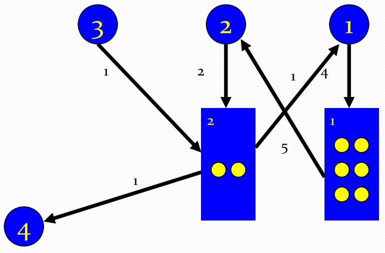
Reduction rules?

- Find a process that can have all its current requests satisfied (e.g. the “available amount” of any resource it wants is at least enough to satisfy the request)
- Erase that process (in effect: grant the request, let it run, and eventually it will release the resource)
- Continue until we either erase the graph or have an irreducible component. In the latter case we’ve identified a deadlock

This graph is reducible: The system is not deadlocked



This graph is not reducible: The system is deadlocked



A tricky choice…

- When should resources be treated as “different classes”?
  - Seems obvious
    - “memory pages” are different from “forks”
  - But suppose we split some resource into two sets?
    - The *main group of memory* and the *extra memory*
  - Keep this in mind next week when we talk about ways of avoiding deadlock.
    - It proves useful in doing “ordered resource allocation”

Take-Away: Conditions for Deadlock

- **Mutual Exclusion**
  - At least one resource must be held is in non-sharable mode
- **Hold and wait**
  - There exists a process holding a resource, and waiting for another
- **No preemption**
  - Resources cannot be preempted
- **Circular wait**
  - There exists a set of processes {P_1, P_2, … P_N}, such that
    - P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur