Deadlock

Chapter 32 in "Three Easy Steps"
Chapter 19 in the Harmony Book

Dining Philosophers

```
Pi: do forever
      acquire( left(i) );
      acquire( right(i) );
      eat;
      release( left(i) );
      release( right(i) );
    end
```

left(i): i

right(i): (i+1) mod 5

Dining Philosophers in Harmony

```
from synch import Lock, acquire, release

const N = 5

forks = [Lock(),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            acquire(?forks[left])
            acquire(?forks[right])
            # dine
            release(?forks[left])
            release(?forks[right])
            # think

for i in {0..N–1}:
    spawn diner(i)
```

Dining Philosophers in Harmony

Issue: Non-terminating state

Turn	Thread	Instructions Executed	PC	Shared Variables: forks 0	1	2	3	4	Output
1	T0: __init__()		1122	False	False	False	False	False	
2	T4: diner(3)		797	False	False	False	True	False	
3	T1: diner(0)		797	True	False	False	True	False	
4	T2: diner(1)		797	True	True	False	True	False	
5	T3: diner(2)		797	True	True	True	True	False	
6	T5: diner(4)		797	True	True	True	True	True	

/Users/rvr/github/harmony/harmony/harmony_model_checker/modules/synch.hny:31
atomically when not !binsema:

756	Load
757	LoadVar old
758	DelVar old
759	2-ary ==
760	StoreVar result
761	LoadVar result
762	JumpCond False 768
763	LoadVar p
764	DelVar p

Threads

ID	Status	Stack Trace		Stack Top
T0	terminated	__init__()		
T1	blocked	diner(0)	left: 0, result: None, right: 1	
		acquire(forks[1])	binsema: ?forks[1], result: None	
T2	blocked	diner(1)	left: 1, result: None, right: 2	
		acquire(forks[2])	binsema: ?forks[2], result: None	
T3	blocked	diner(2)	left: 2, result: None, right: 3	
		acquire(forks[3])	binsema: ?forks[3], result: None	
T4	blocked	diner(3)	left: 3, result: None, right: 4	
		acquire(forks[4])	binsema: ?forks[4], result: None	
T5	blocked	diner(4)	left: 4, result: None, right: 0	
		acquire(forks[0])	binsema: ?forks[0], result: None	

Problematic Emergent Properties

- Starvation: Process waits forever
- Deadlock: a set of processes exist, where each is blocked and can become unblocked only by the action of another process in the same set
  - Deadlock implies Starvation (but not viceversa)
  - Starvation often tied to fairness — which requires that a process be not forever blocked on a condition that becomes (i) continuously true or (ii) infinitely-often true

Testing for starvation or deadlock is difficult in practice

More Examples of Deadlock

- Example 1 (initially in1 = in2 = False):

in1 = True; await not in2; in1 = False
//
in2 = True; await not in1; in2 = False

- Example 2 (initially lk1 = lk2 = released):

acquire(lk1); acquire(lk2); release(lk2); release(lk1)
//
acquire(lk2); acquire(lk1); release(lk1); release(lk2)

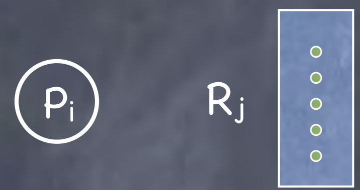
System Model

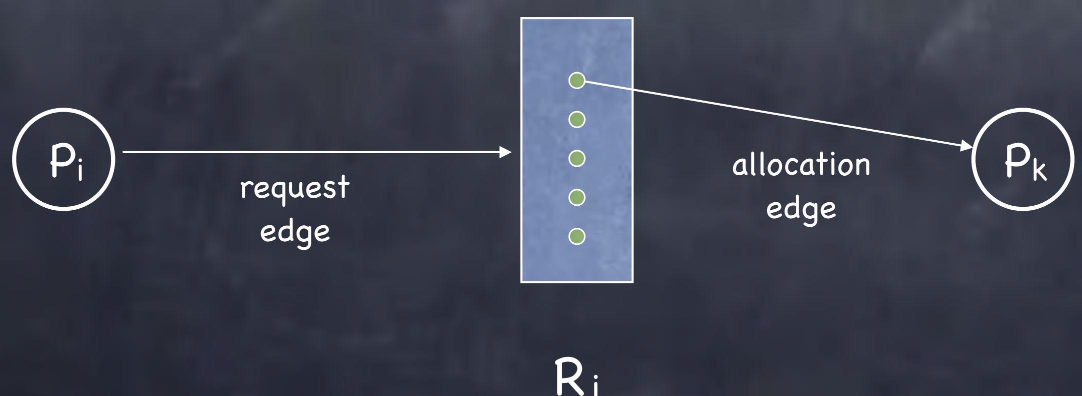
- Set of resources requiring "exclusive" access
  - Might be "k exclusive access" if k instances of resource are available
  - Examples: buffers, packets, I/O devices, processors
- Protocol to access a resource causes blocking
  - If resource is free, access is granted and process proceeds
    - Uses resource
    - Releases resource
  - If resource is in use, process blocks

A Graph Theoretic Model of Deadlock

Resource Allocation Graph

- Computer system modeled as a RAG, a directed graph G(V, E)
  - $V = \{P_1,\ldots,P_n\} \cup \{R_1,\ldots,R_n\}$
  - E = {edges from a resource to a process} ∪ {edges from a process to a resource}





Necessary conditions for deadlock

Deadlock only if they all hold

1. Bounded resources

   Acquire can block invoker

2. No preemption

   the resource is mine, MINE! (until I release it)

3. Wait while holding

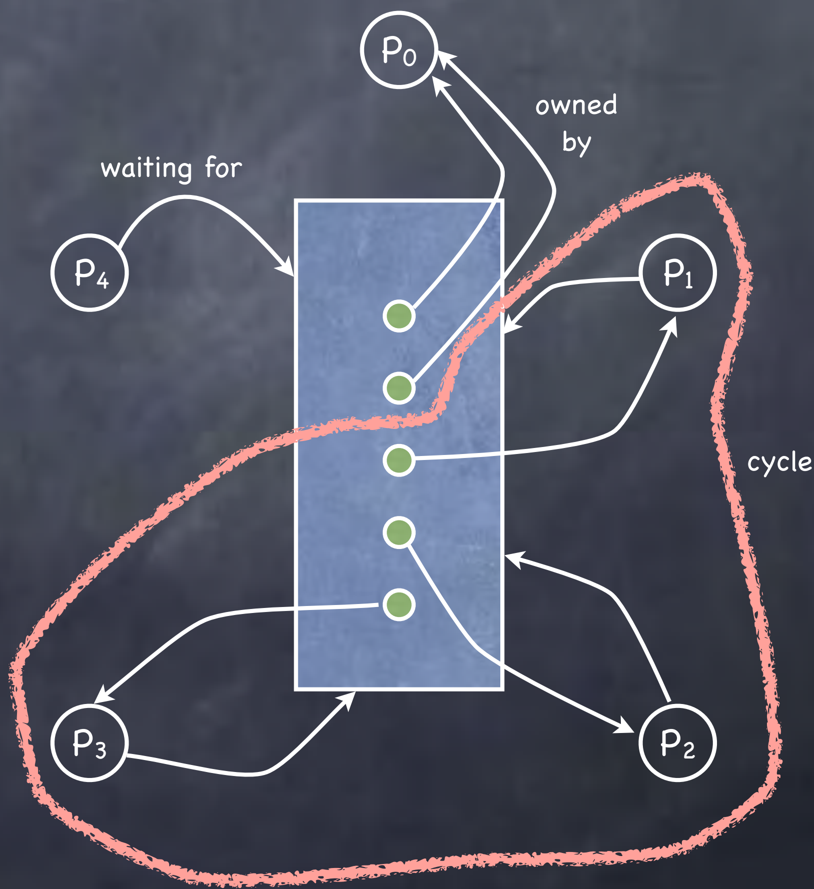
   holds one resource while waiting for another

4. Circular waiting

   P_i waits for P_{i+1} and holds a resource requested by P_{i-1}

   sufficient if one instance of each resource

Not sufficient in general



Deadlock is Undesirable!

- Deadlock prevention: Ensure that a necessary condition cannot hold
- Deadlock avoidance: System does not allocate resources that may lead to a deadlock
- Deadlock detection: Allow system to deadlock; detect it; recover

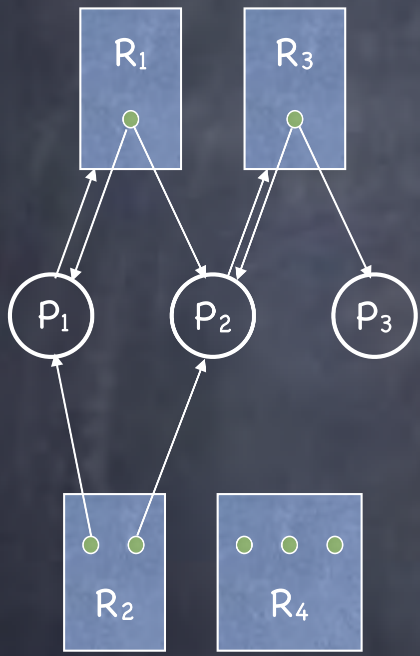
Testing for cycles

- Reduction Algorithm
  - Find a node with no outgoing edges
    - Erase any edges coming into it
    - Repeat until no such node
- Intuition: Node with no outgoing edges is not waiting on any resource
  - It will eventually finish and release its resources
  - Processes waiting for <u>those</u> resources will be able to acquire them and will no longer be waiting!

Erase all edges $\Longleftrightarrow$ Graph has no cycles

Edges remain $\Longleftrightarrow$ Deadlock

RAG Reduction



Deadlock?

NO! (no cycles)

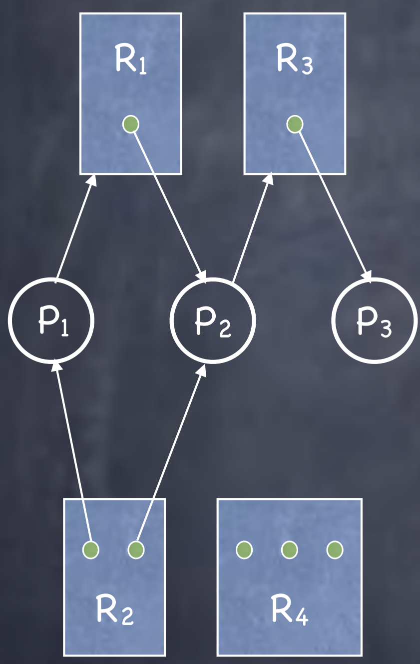
Step 1: Satisfy P_3's requests

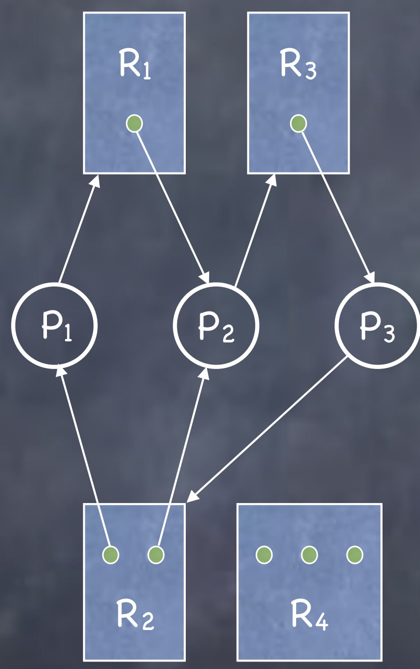
Step 2: Satisfy P_2's requests

Step 3: Satisfy P_1's requests

Schedule [P_3 P_2 P_1] completely eliminates edges!

RAG Reduction





Deadlock?

NO! (no cycles)

Step 1: Satisfy P_3's requests

Step 2: Satisfy P_2's requests

Step 3: Satisfy P_1's requests

Schedule [P_3 P_2 P_1] completely eliminates edges!

Deadlock?

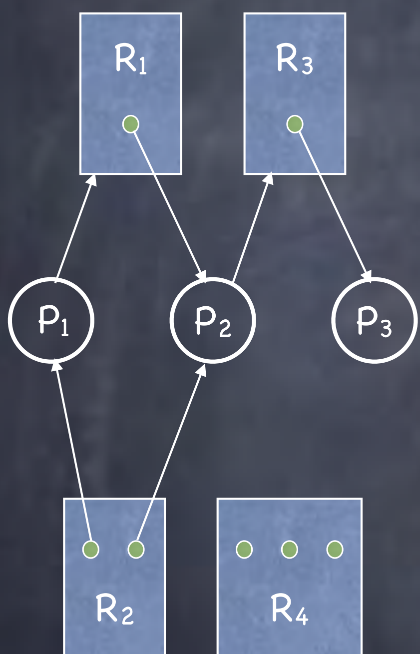
Yes!

RAG has a cycle

Every node has some outgoing edge

Cannot satisfy any of P_1, P_2, P_3 requests!

RAG Reduction



Deadlock?

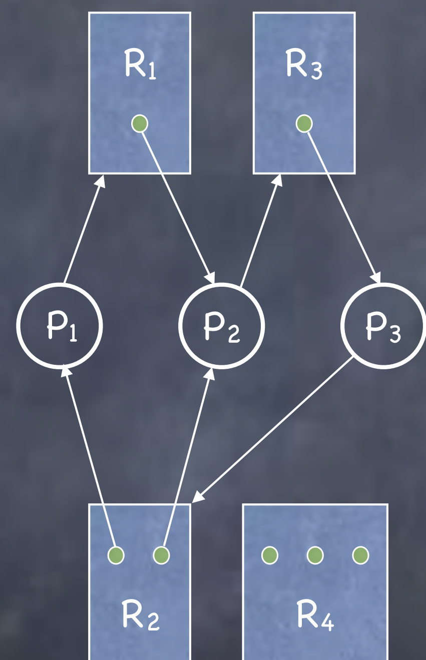
NO! (no cycles)

Step 1: Satisfy P_3's requests

Step 2: Satisfy P_2's requests

Step 3: Satisfy P_1's requests

Schedule [P_3 P_2 P_1] completely eliminates edges!



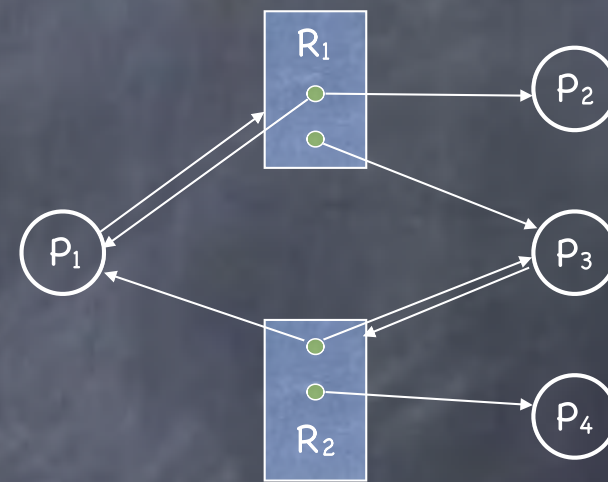
Deadlock?

Yes!

RAG has a cycle

Every node has some outgoing edge

Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

NO!

RAG has a cycle

Schedule [P_2 P_1 P_3 P_4] completely eliminates edges!

More Musings on Deadlock

- Does the order of RAG reduction matter?
  - No. If P_i and P_j can both be reduced, reducing P_i does not affect the reducibility of P_j
- Does a deadlock disappear on its own?
  - No. Unless a process is killed or forced to release a resource, we are stuck!
- If a system is not deadlocked at time T, is it guaranteed to be deadlock-free at T+1?
  - No. Just by requesting a resource (never mind being granted one) a process can create a circular wait!

Deadlock Prevention: Negate ①

- Eliminate "Acquire can block invoker/bounded resources"
  - Make resources sharable without locks
    - Wait-free synchronization
    - The Harmony book (Chapter 23) has examples of non-blocking data structures
  - Have sufficient resources available, so acquire never delays (duh!)
    - E.g., use an unbounded queue, or make sure that queue is "large enough"

Deadlock Prevention: Negate ②

- Allow preemption
  - Requires mechanisms to save/restore resource state
    - multiplexing (registers, memory, etc). VS.
    - undo/redo (database transaction processing)
  - Allow OS to preempt resources of waiting processes
  - Allow OS to preempt resources of requesting processes