Deadlock

Science

Computer Science

CS 450 : Operating Systems

Michael Lee <lee@iit.edu>

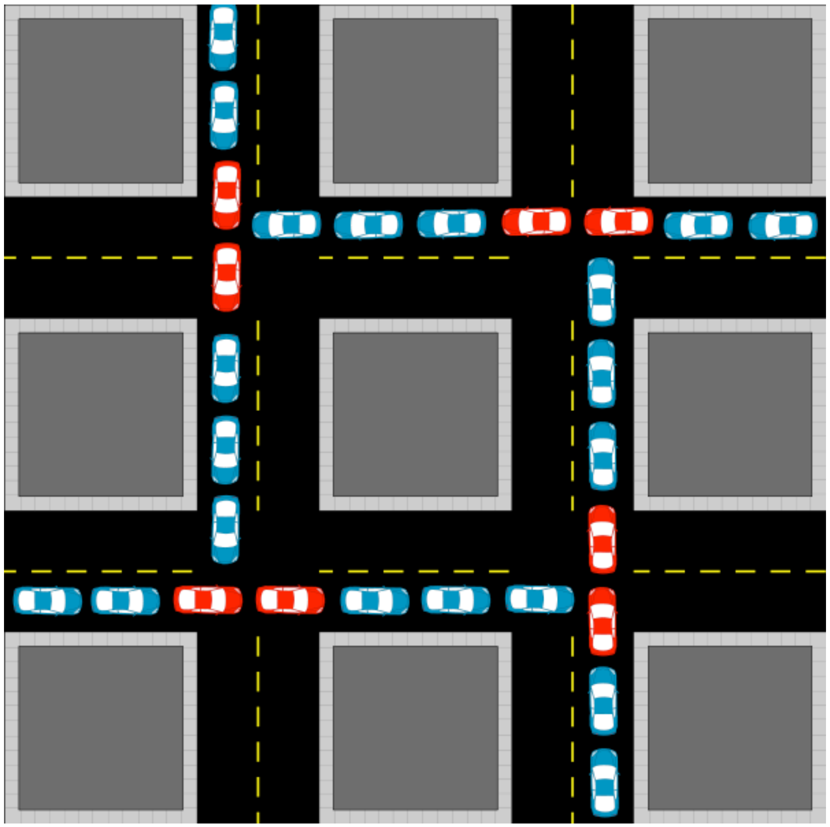
IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

deadlock |ˈdedˌläk|

noun

1 [in sing.] a situation, typically one involving opposing parties, in which no progress can be made *: an attempt to* ***break the deadlock***.

- New Oxford American Dictionary



Traffic Gridlock

```
mtx_A.lock()
mtx_B.lock()

  # critical section

mtx_B.unlock()
mtx_A.unlock()
```

```
mtx_B.lock()
mtx_A.lock()

  # critical section

mtx_B.unlock()
mtx_A.unlock()
```

Software Gridlock

§ Necessary conditions for Deadlock

i.e., what conditions need to be true (of some system) so that deadlock *is possible*?

(not the same as *causing* deadlock!)

I. Mutual Exclusion

- resources can be held by processes in a mutually exclusive manner

II. Hold & Wait

- while holding one resource (in mutex), a process can request another resource

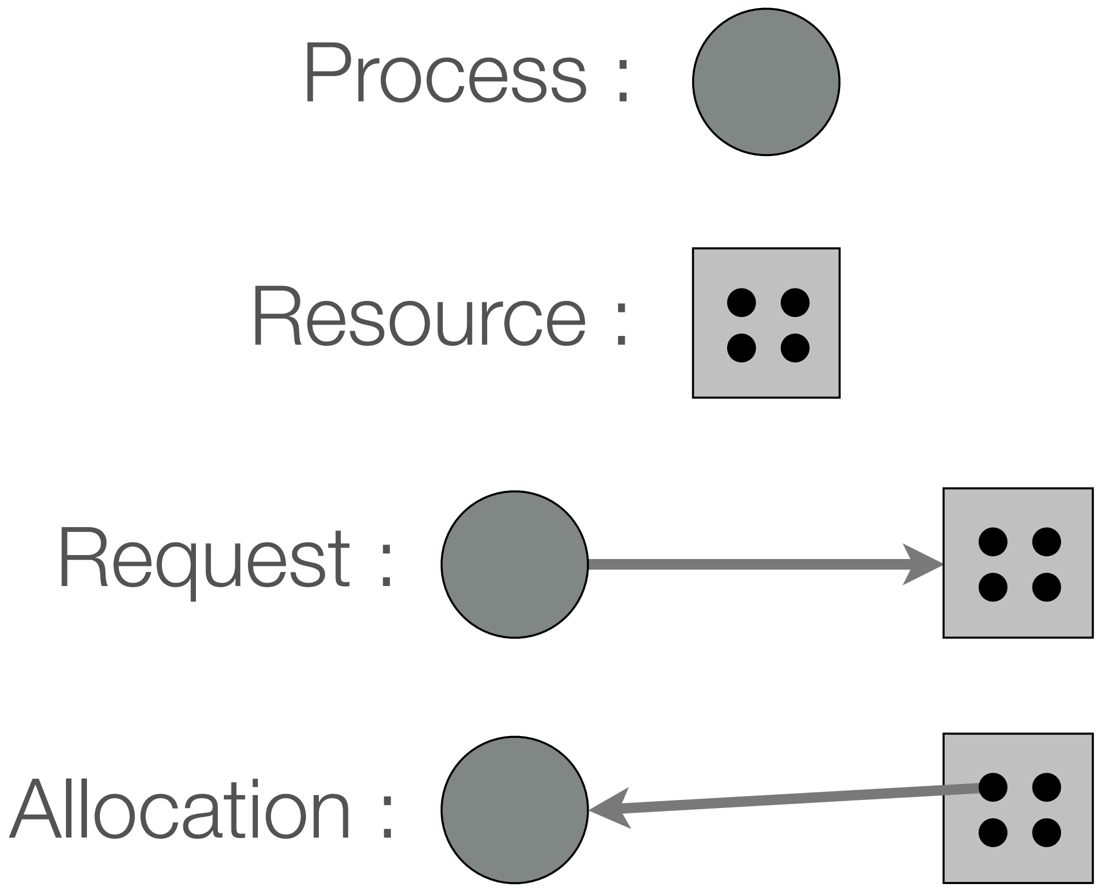
III. No Preemption

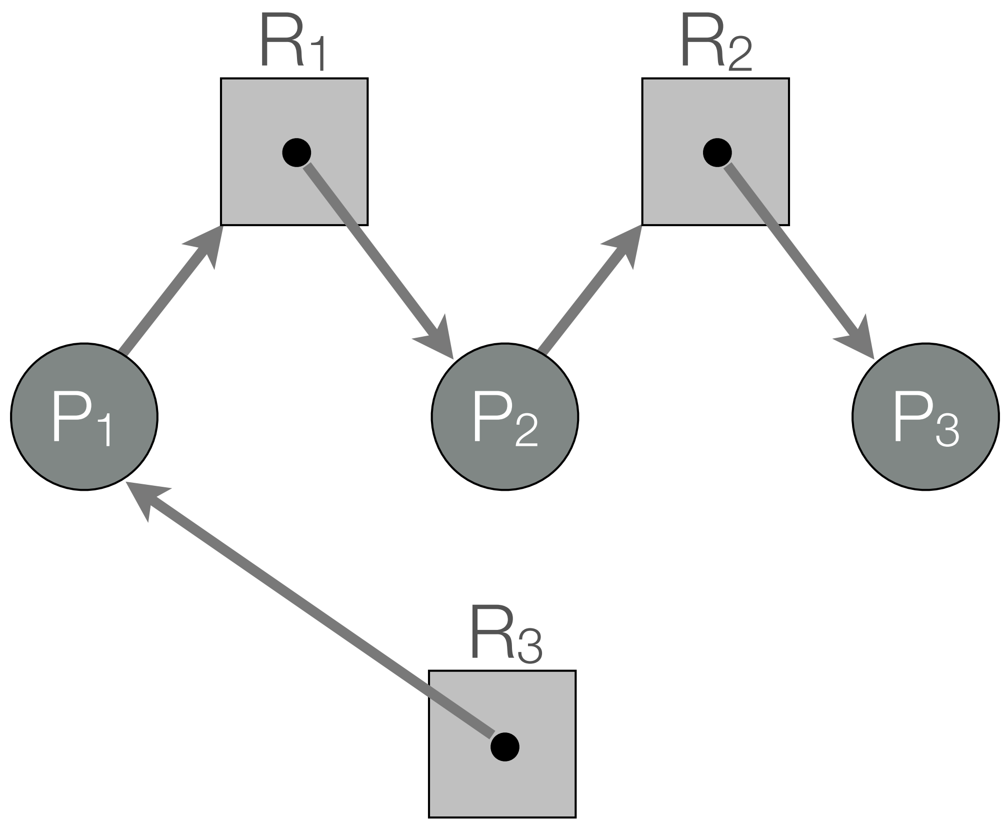
- one process can not force another to give up a resource; i.e., releasing is *voluntary*

IV. Circular Wait

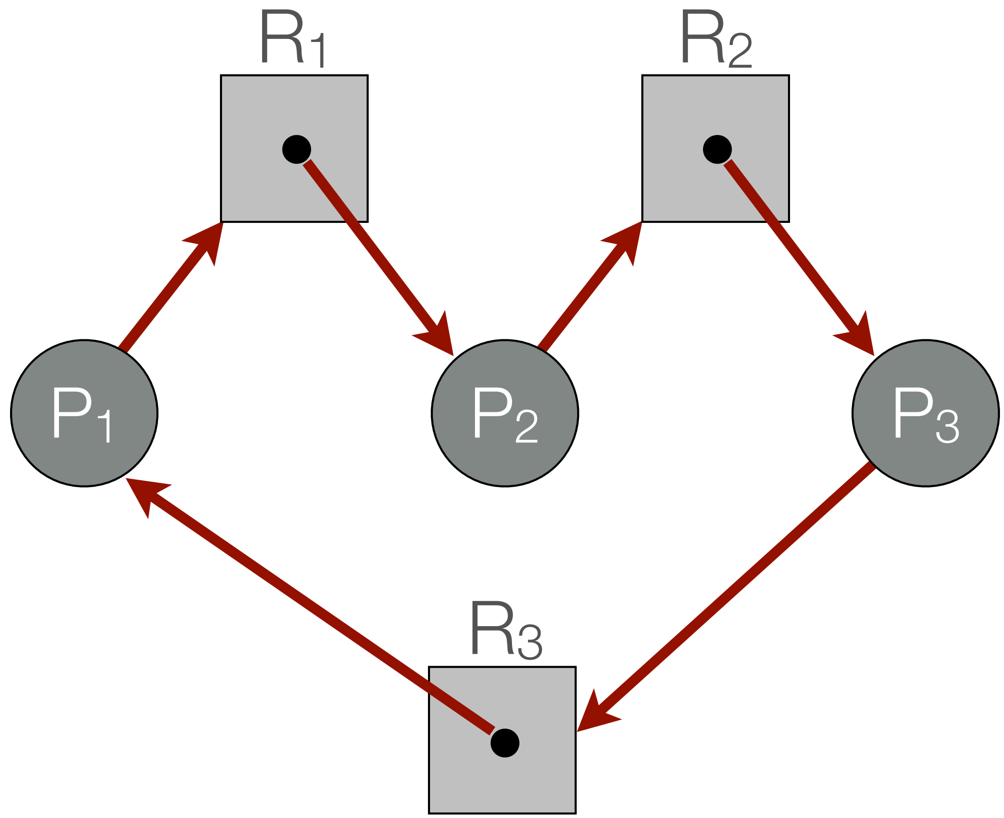
- resource requests and allocations create a *cycle* in the *resource allocation graph*

§ Resource Allocation Graphs


Process :
Resource :
Request :
Allocation :



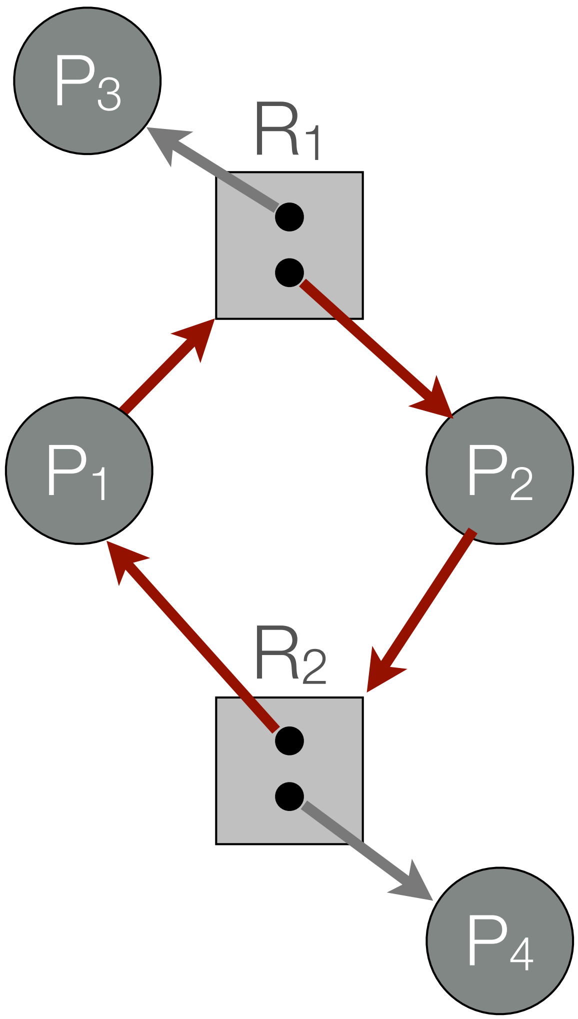
Circular wait is absent = no deadlock



All 4 necessary conditions in place; Deadlock!

in a system with *only single-instance resources*,

necessary conditions ⇔ deadlock



Cycle without Deadlock!

not practical (or always possible) to detect deadlock using a graph

— but convenient to help us reason about things

§ Approaches to Dealing with Deadlock

1. Ostrich algorithm
   (ignore it and hope it never happens)
2. Prevent it from occurring (avoidance)
3. Detection & recovery

§ Deadlock avoidance

¶ Approach 1: eliminate necessary condition(s)

Mutual exclusion?

- eliminating mutex requires that all resources be *shareable*
- when not possible (e.g., disk, printer), can sometimes use a *spooler process*

but what about semaphores, file locks, etc.?

- not all resources are spoolable
- *cannot eliminate mutex* in general

Hold & Wait?

- elimination requires resource requests to be all-or-nothing affair
  - if currently holding, needs to release all before requesting more

in practice, very inefficient
& starvation is possible!

— cannot eliminate hold & wait

No preemption?

- alternative: allow process to preempt each other and “steal” resources
  - mutex locks can not be counted on to stay locked!
- in practice, *cannot eliminate* this either!

Circular Wait is where it’s at.

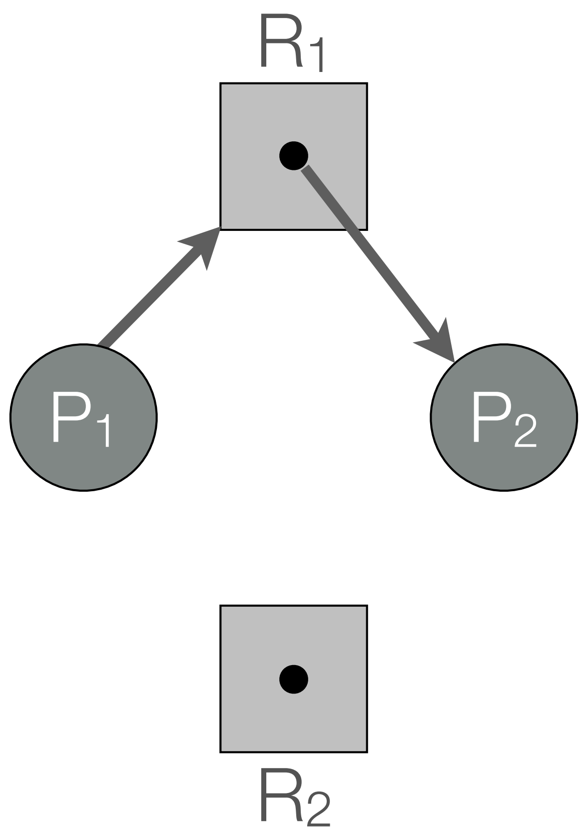
simple mechanism to prevent wait cycles:

- order all resources
- require that processes request resources in order

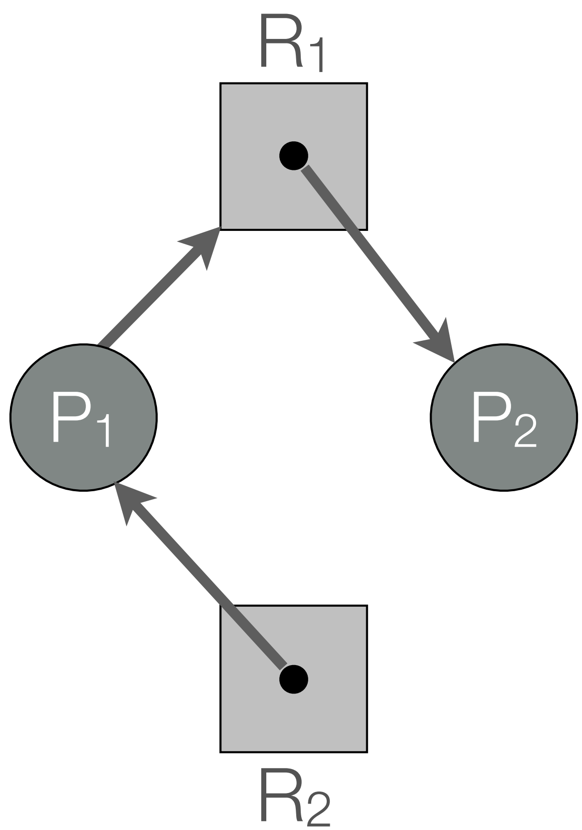
but *impractical* — can not count on processes to need resources in a certain order

... and forcing a certain order can result in *poor resource utilization*

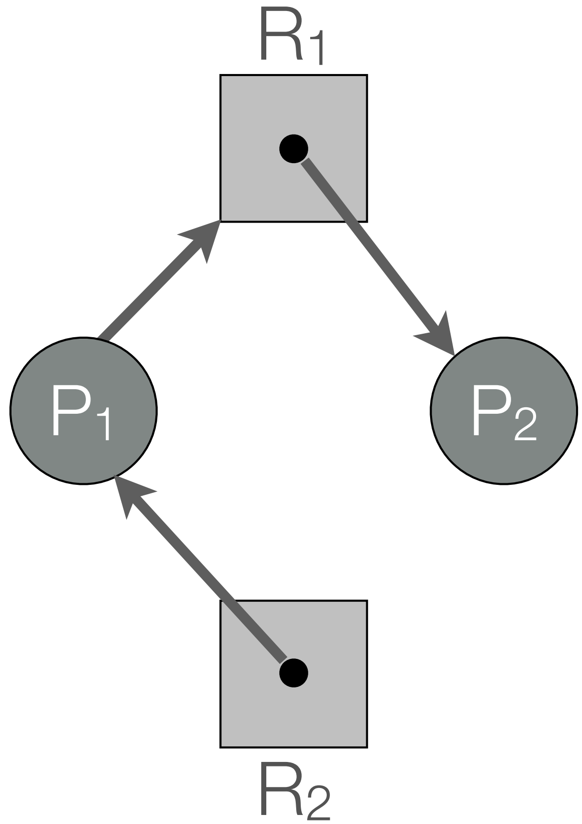
¶ Approach 2: *intelligently prevent* circular wait



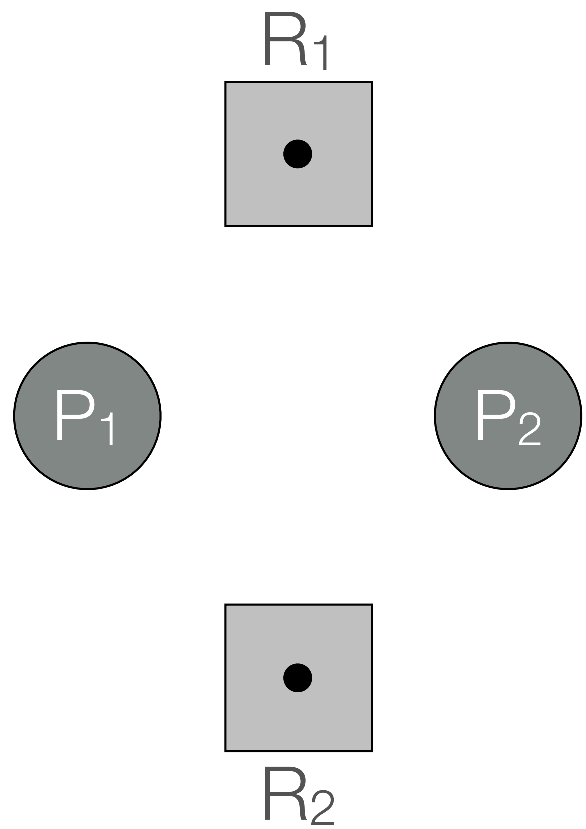
possible to create a cycle (with one edge)?



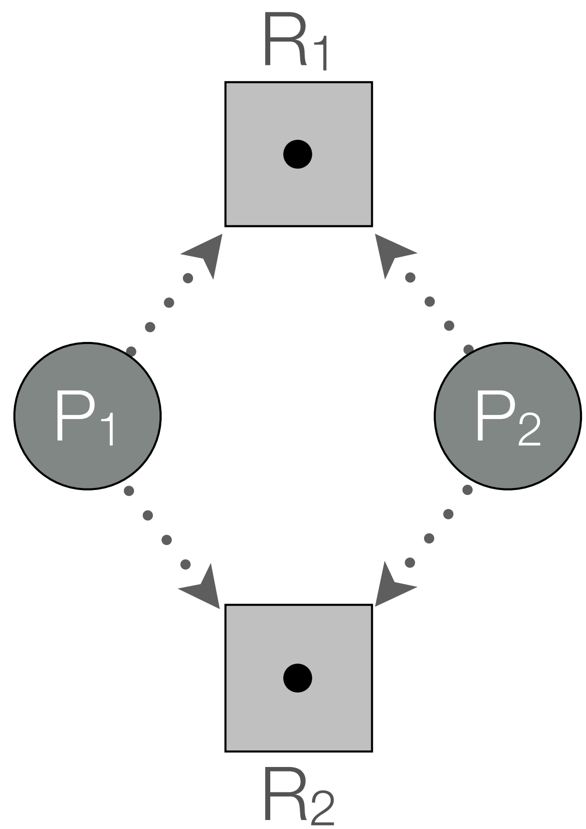
possible to create a cycle (with one edge)?



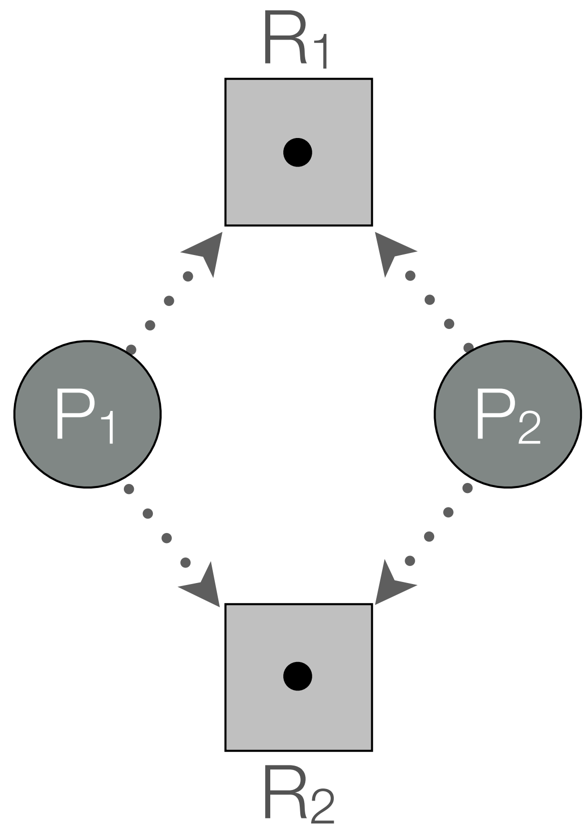
it's quite possible that P_2 won't need R_2, or maybe P_2 will release R_1 before requesting R_2, but we don't know if/when...



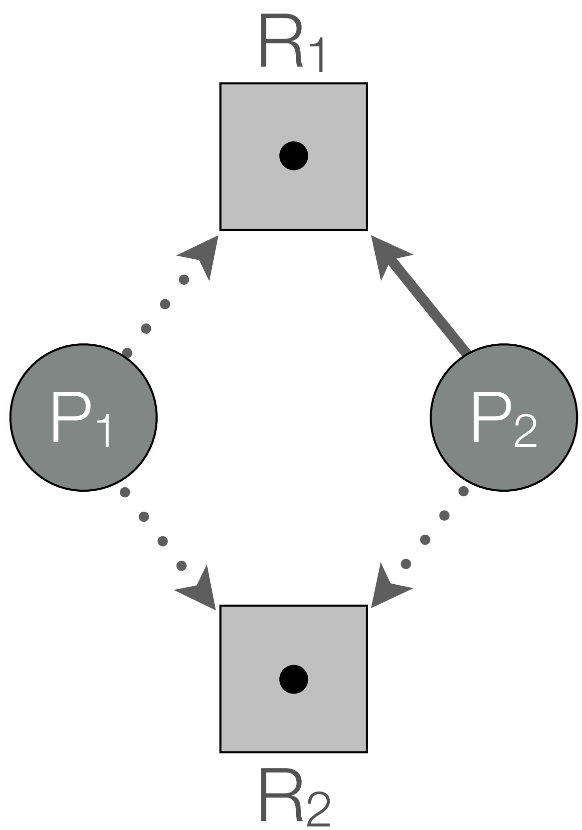
preventing circular wait means avoiding a state where a cycle is an imminent *possibility*



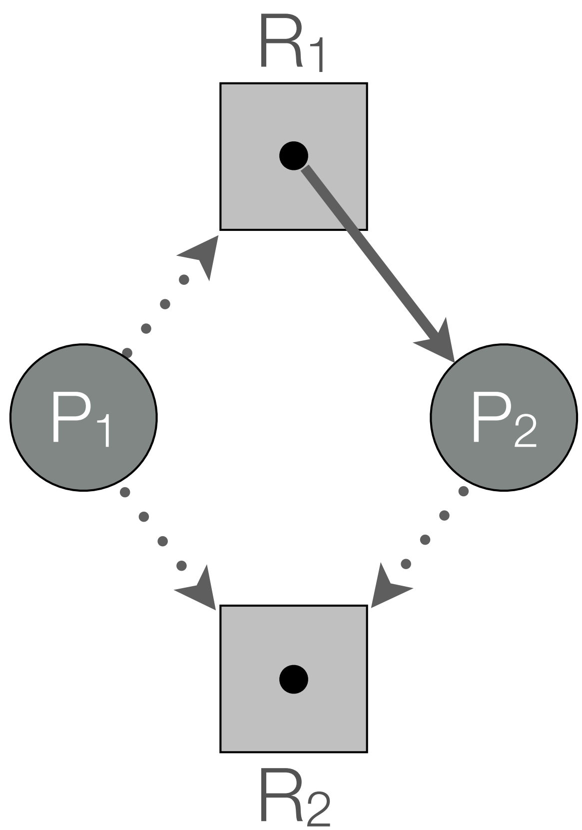
to predict deadlock, we can ask processes to "claim" all resources they need in advance



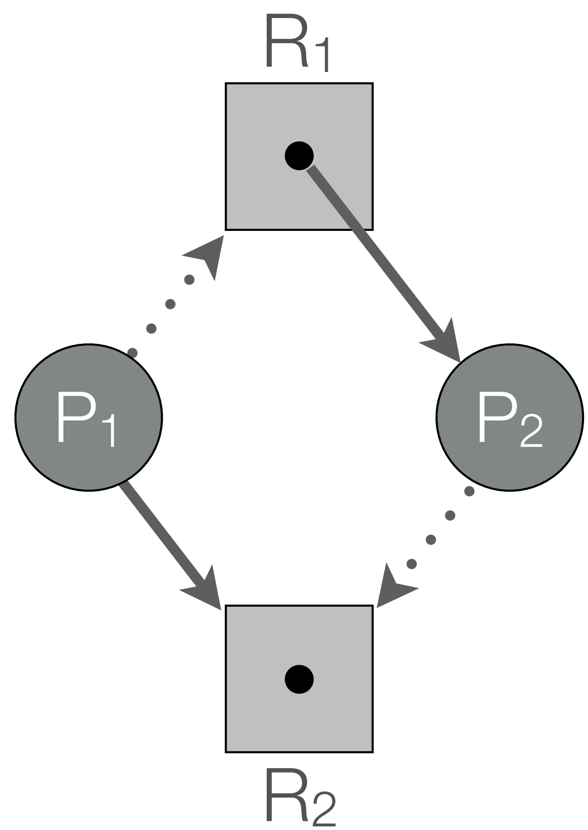
graph with "claim edges"



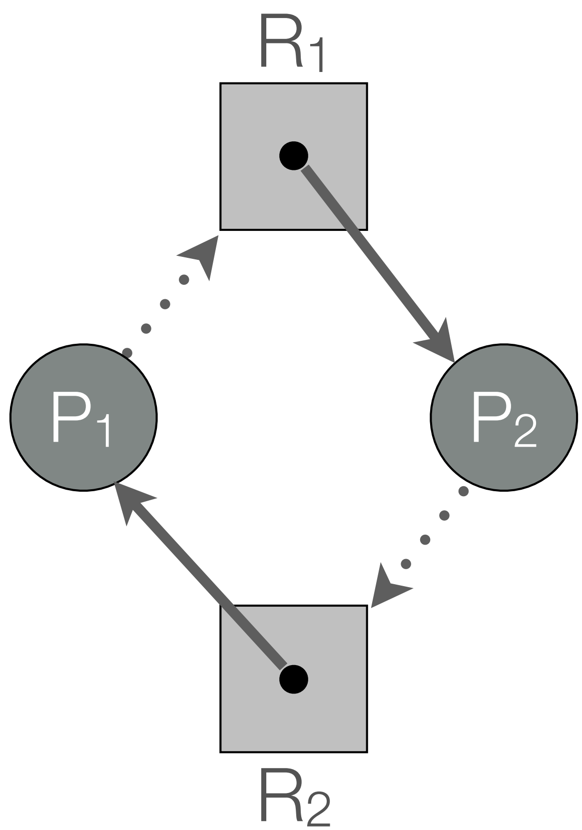
P_2 requests R_1



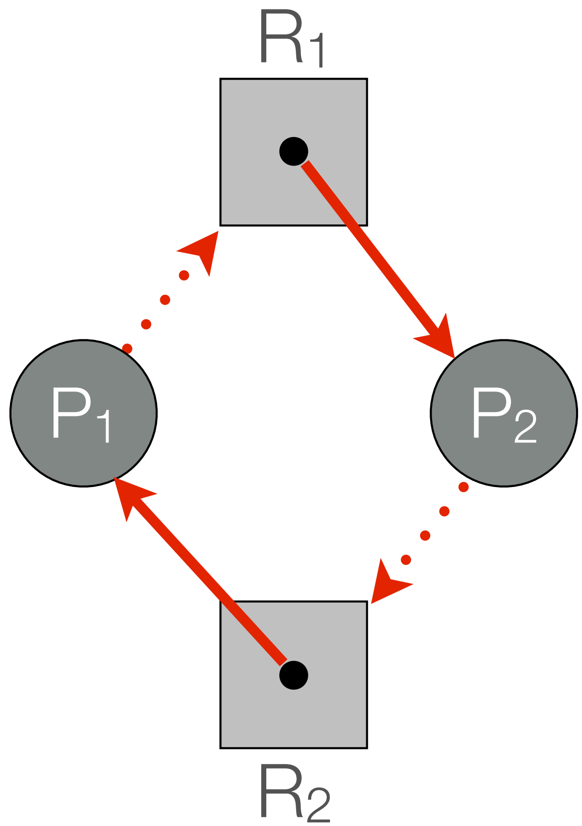
convert to allocation edge; no cycle



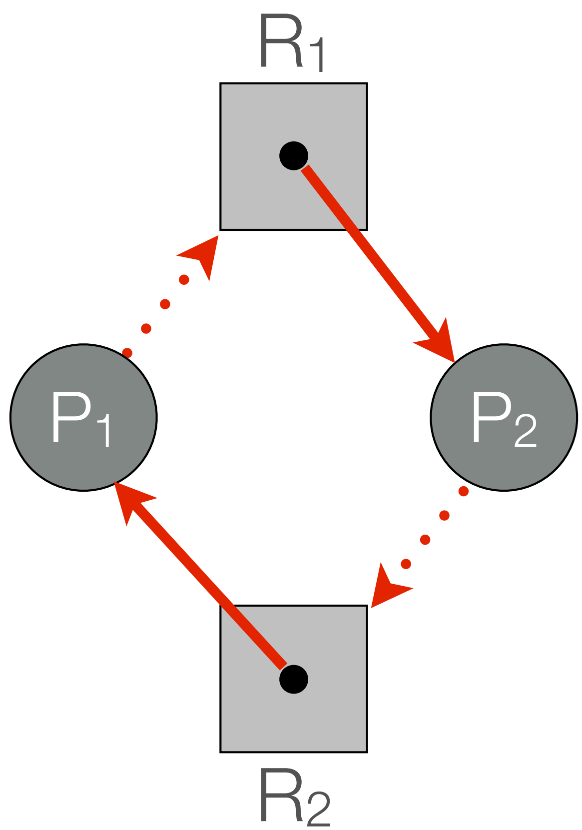
P_1 requests R_2



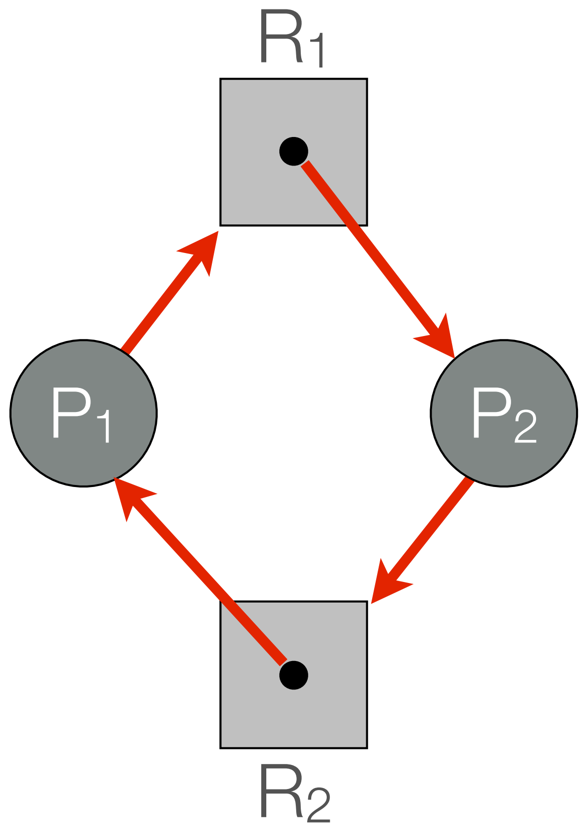
if we convert to an allocation edge ...



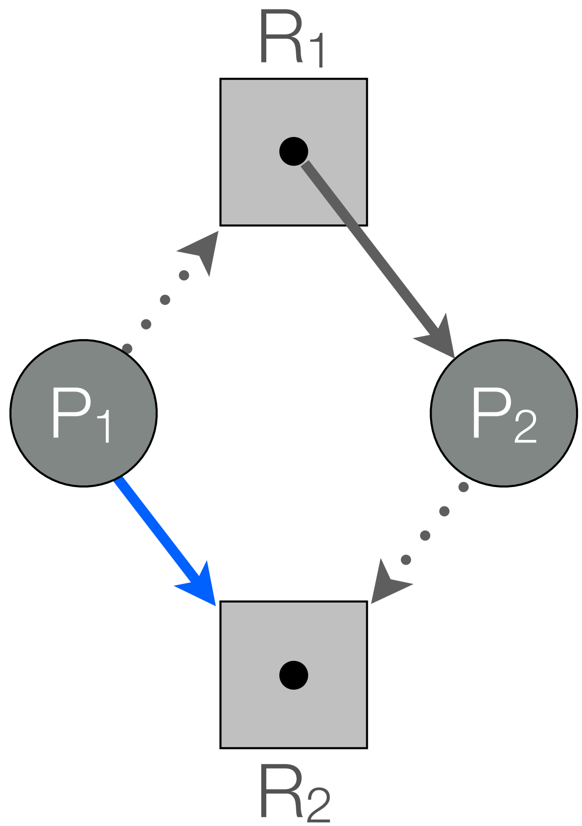
cycle involving claim edges!



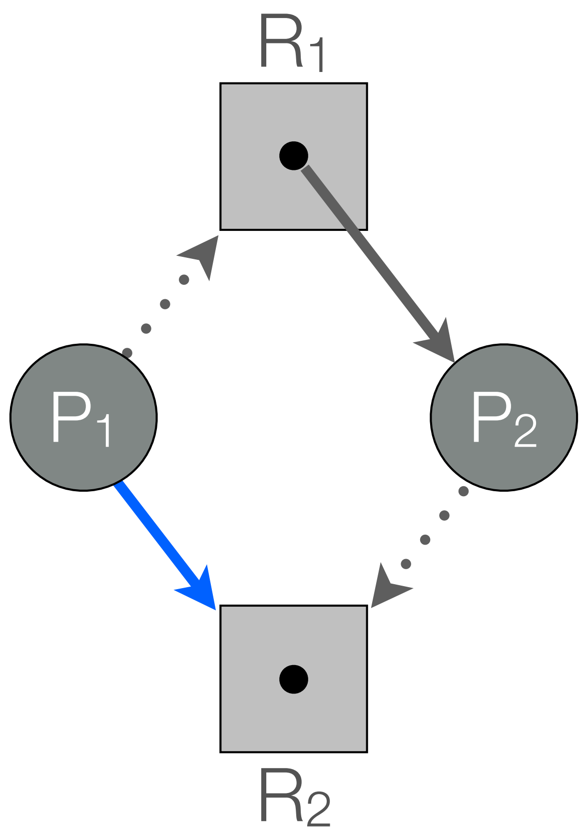
means that if processes fulfill their claims,
we cannot avoid deadlock!



i.e., $P_1 \rightarrow R_1$, $P_2 \rightarrow R_2$

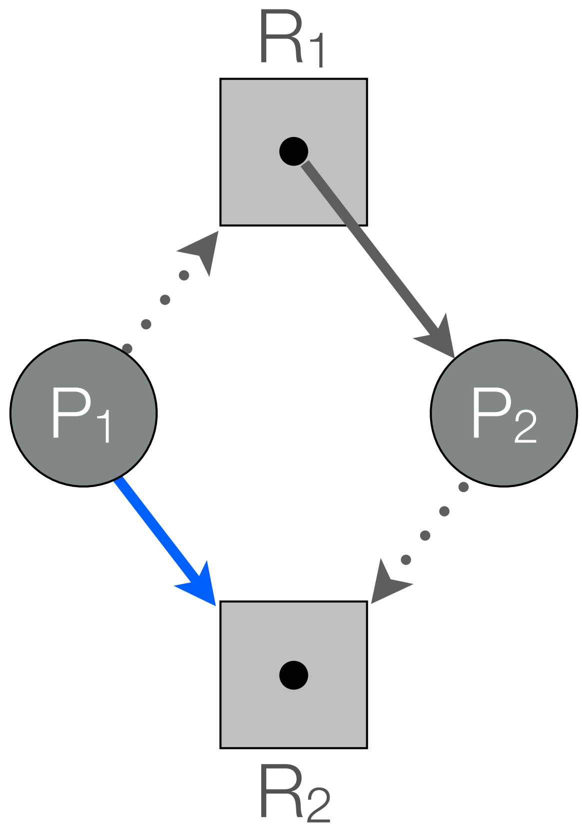


$P_1 \rightarrow R_2$ should be blocked by the kernel,
even if it can be satisfied with available resources

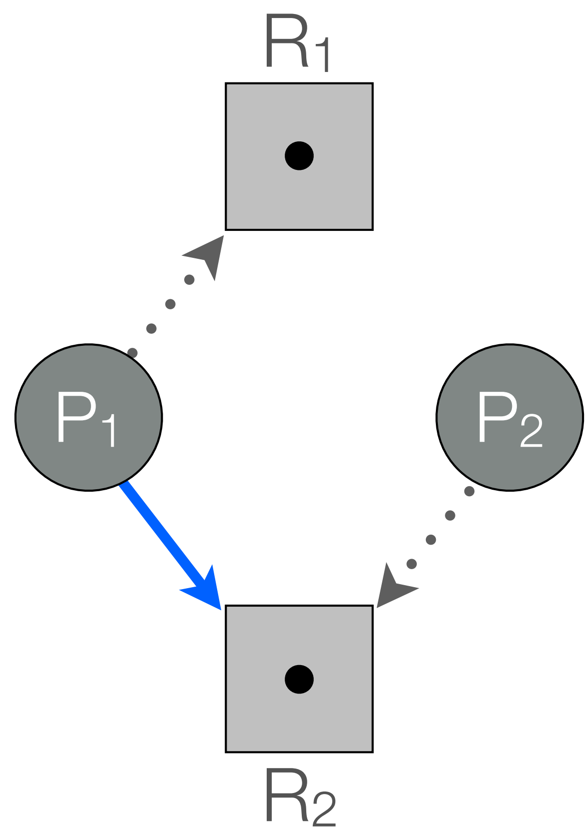


this is a “safe” state … i.e., no way a process can cause deadlock directly (i.e., without OS alloc) idea: if granting an incoming request would create a cycle in a graph with claim edges, deny that request (i.e., block the process)

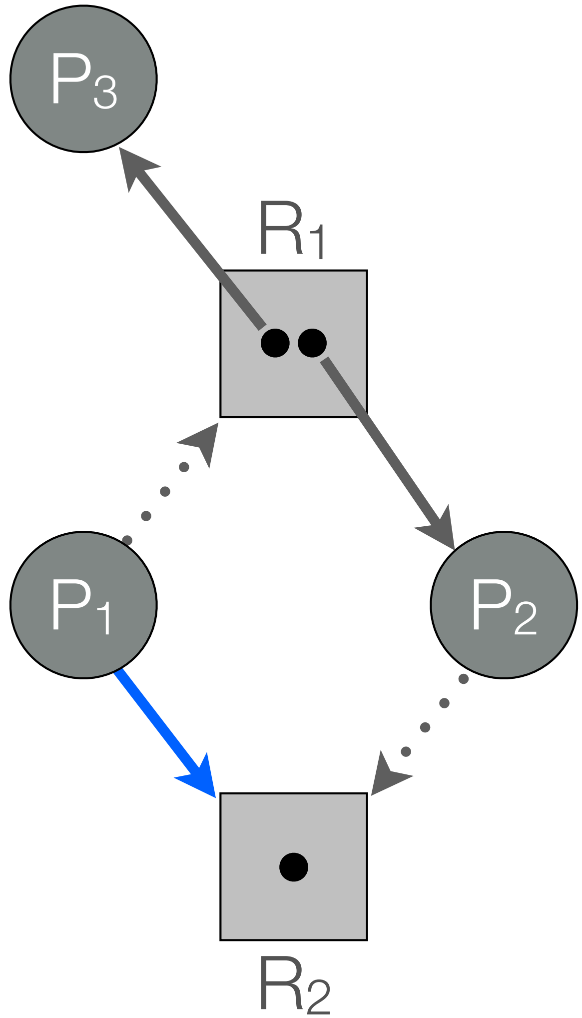
— approve later when no cycle would occur



P_2 releases R_1

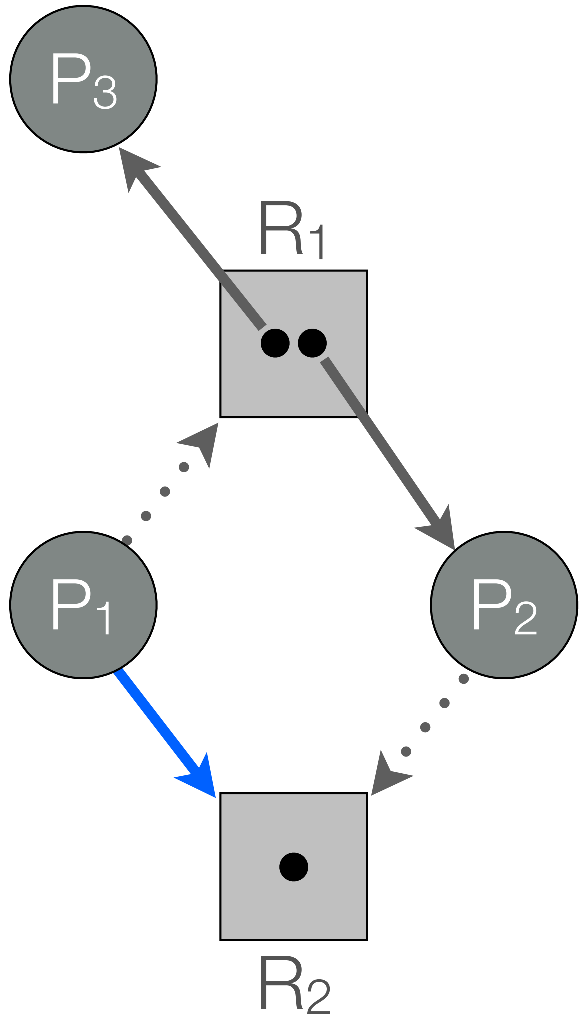


now ok to approve $P_1 \rightarrow R_2$ (unblock P_1)



should we still deny $P_1 \rightarrow R_2$?

problem: this approach may incorrectly predict imminent deadlock when *resources with multiple instances* are involved



requires a *more general* definition of "safe state"

❡ Banker's Algorithm

(by Edsger Dijkstra)

basic idea:

- define how to recognize system “safety”
- whenever a resource request arrives:
  - *simulate* allocation & check state
  - allocate iff simulated state is safe

some assumptions we need to make:

1. a non-blocked process holding a resource will *eventually* release it
2. it is known *a priori* how many instances of each resource a given process needs

Safe State

- There exists a sequence <P_1, P_2, ..., P_n>, where each P_k can complete with:
  - currently available (free) resources
  - resources held by $P_1 \ldots P_{k-1}$

Data Structures

Processes $P_1 \ldots P_n$, Resources $R_1 \ldots R_m$:

available[j] = num of R_j available

max[i][j] = max num of R_j required by P_i

allocated[i][j] = num of R_j allocated to P_i

need[i][j] = max[i][j] - allocated[i][j]

Safety Algorithm

1. finish[i] $\leftarrow$ false $\forall$ i $\in$ 1...n
   work $\leftarrow$ available
2. Find i : finish[i] = false & need[i][j] $\leq$ work[j] $\forall$ j
   If none, go to 4.
3. work $\leftarrow$ work + allocated[i]; finish[i] $\leftarrow$ true
   Go to 2.
4. Safe state iff finish[i] = true $\forall$ i

incoming request represented by *request array*

`request[j]` = num of resource R_j requested

(a process can require multiple instances of more than one resource at a time)

Processing Request from P_k:

1. If request[j] $\leq$ need[k][j] $\forall$ j, continue, else error
2. If request[j] $\leq$ available[j] $\forall$ j, continue, else block
3. Run safety algorithm with:
    - available $\leftarrow$ available - request
    - allocated[k] $\leftarrow$ allocated[k] + request
    - need[k] $\leftarrow$ need[k] - request

if safety algorithm fails, do not allocate, *even if resources are available*!

— either deny request or block caller

3 resources: A (10), B (5), C (7)

	Max			Allocated			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	3	3	2	7	4	3
P_1	3	2	2	2	0	0				1	2	2
P_2	9	0	2	3	0	2				6	0	0
P_3	2	2	2	2	1	1				0	1	1
P_4	4	3	3	0	0	2				4	3	1

- Safe state: $\langle P_1, P_3, P_0, P_2, P_4 \rangle$
- P_3 requests <0, 0, 1>
- P_0 requests <0, 3, 0>

❡ Banker's algorithm discussion

1. Efficiency?
   - how fast is it?
   - how often is it run?

1. $finish[i] \leftarrow false \;\forall\; i \in 1 \ldots n$

   $work \leftarrow available$

   for up to N processes, check M resources

2. Find i : $finish[i] = false$ & $need[i][j] \leq work[j] \;\forall\; j$

   If none, go to 4.

3. $work \leftarrow work + allocated[i]$; $finish[i] \leftarrow true$

   Go to 2. *loop for N processes*

4. Safe state iff $finish[i] = true \;\forall\; i$

$$O(N \cdot N \cdot M) = O(N^2 \cdot M)$$

how often to run?

- need to run on *every resource request*
- can't relax this, otherwise system might become unsafe!

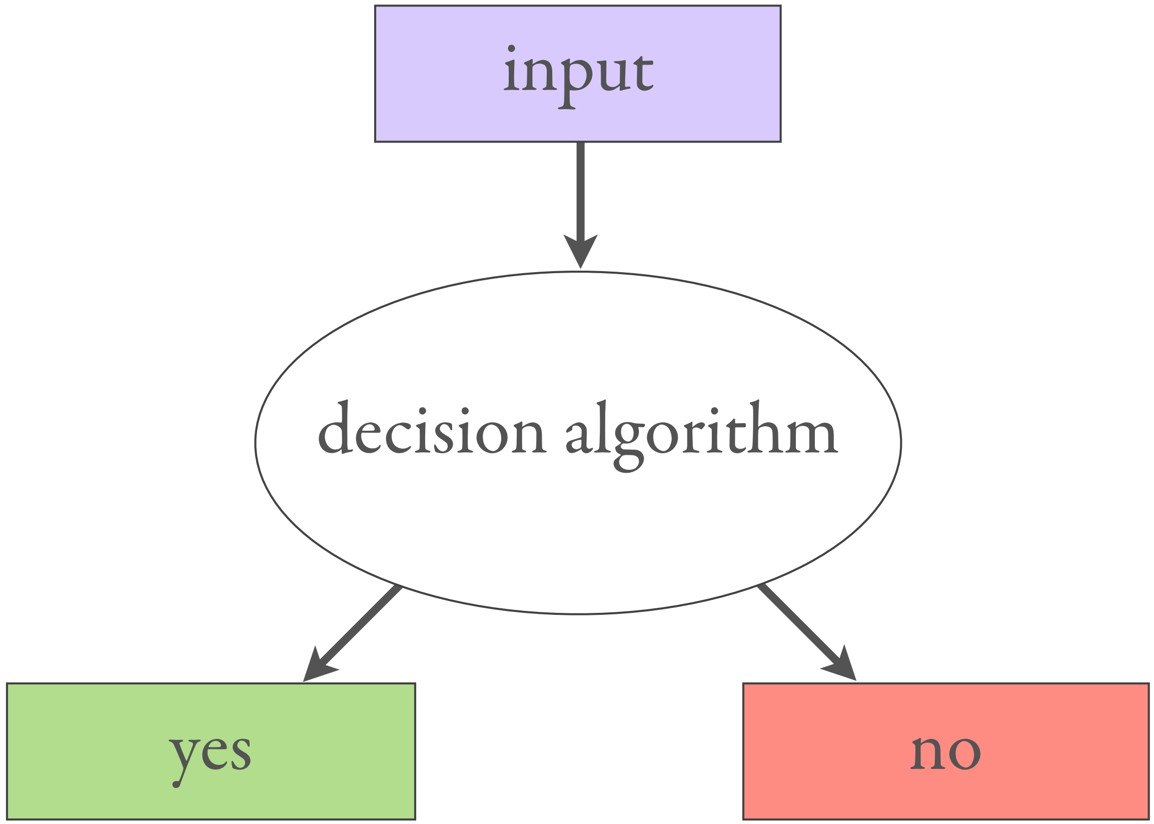
2. Assumption #1: processes will *eventually* release resources

- assuming well-behaved processes
- not 100% realistic, but what else to do?

3. Assumption #2: a priori knowledge of max resource requirements

- highly unrealistic
- process resource needs are dynamic!
- without this assumption, deadlock prevention becomes *much harder*…

¶ Aside: decision problems, complexity theory & the halting problem



a decision problem

e.g., is X evenly divisible by Y?

is N a prime number?

does string S contain pattern P?

a lot of important problems can be reworded as decision problems:

e.g., traveling salesman problem (find the shortest tour through a graph)

$\Rightarrow$ is there a tour shorter than L?

complexity theory *classifies* decision problems by their *difficulty*, and draws *relationships* between those problems & classes class P: solutions to these problems can be found in polynomial time (e.g., $O(N^2)$)

class NP: solutions to these problems can be *verified* in polynomial time

— but *finding* solutions may be harder!

(i.e., superpolynomial)

big open problem in CS:

$$\mathrm{P} = \mathrm{NP}?$$

why is this important?

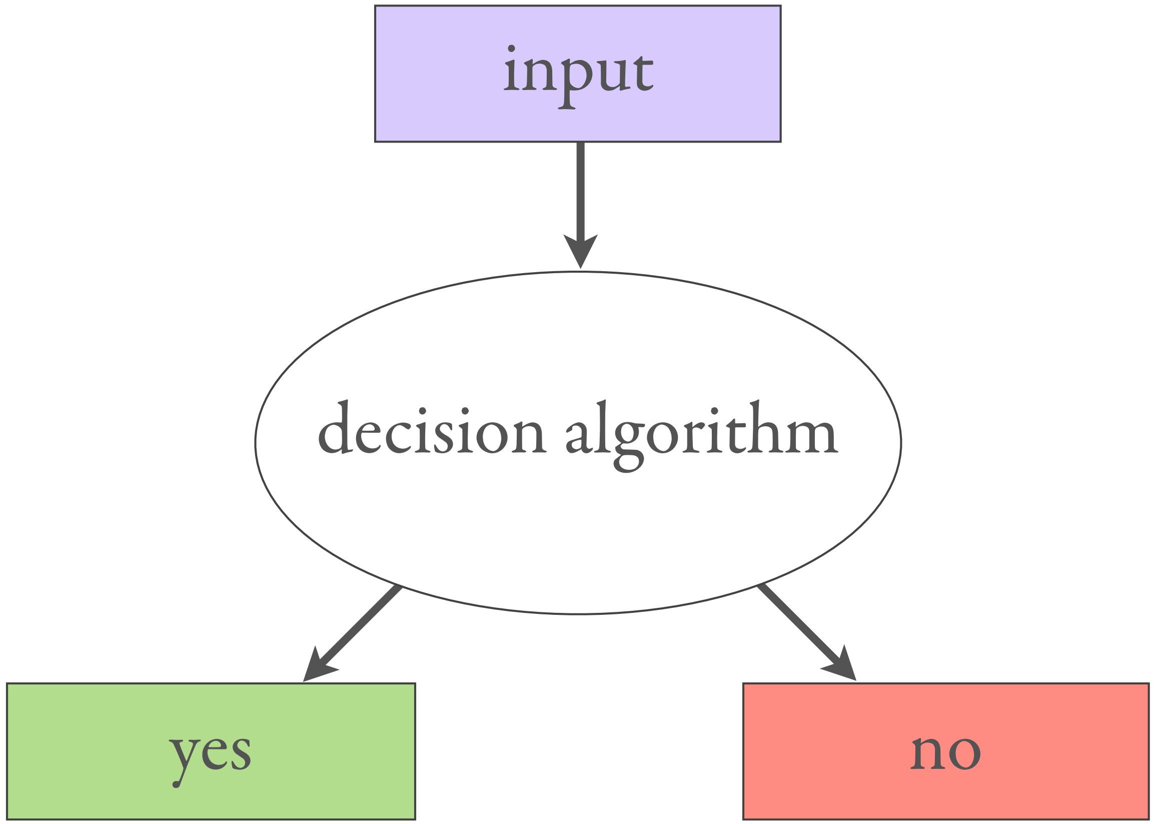
all problems in NP can be reduced to another problem in the NP-complete class,

and all problems in NP-complete can be reduced to each other) if you can prove that *any* NP-complete problem is in P, then *all* NP problems are in P!

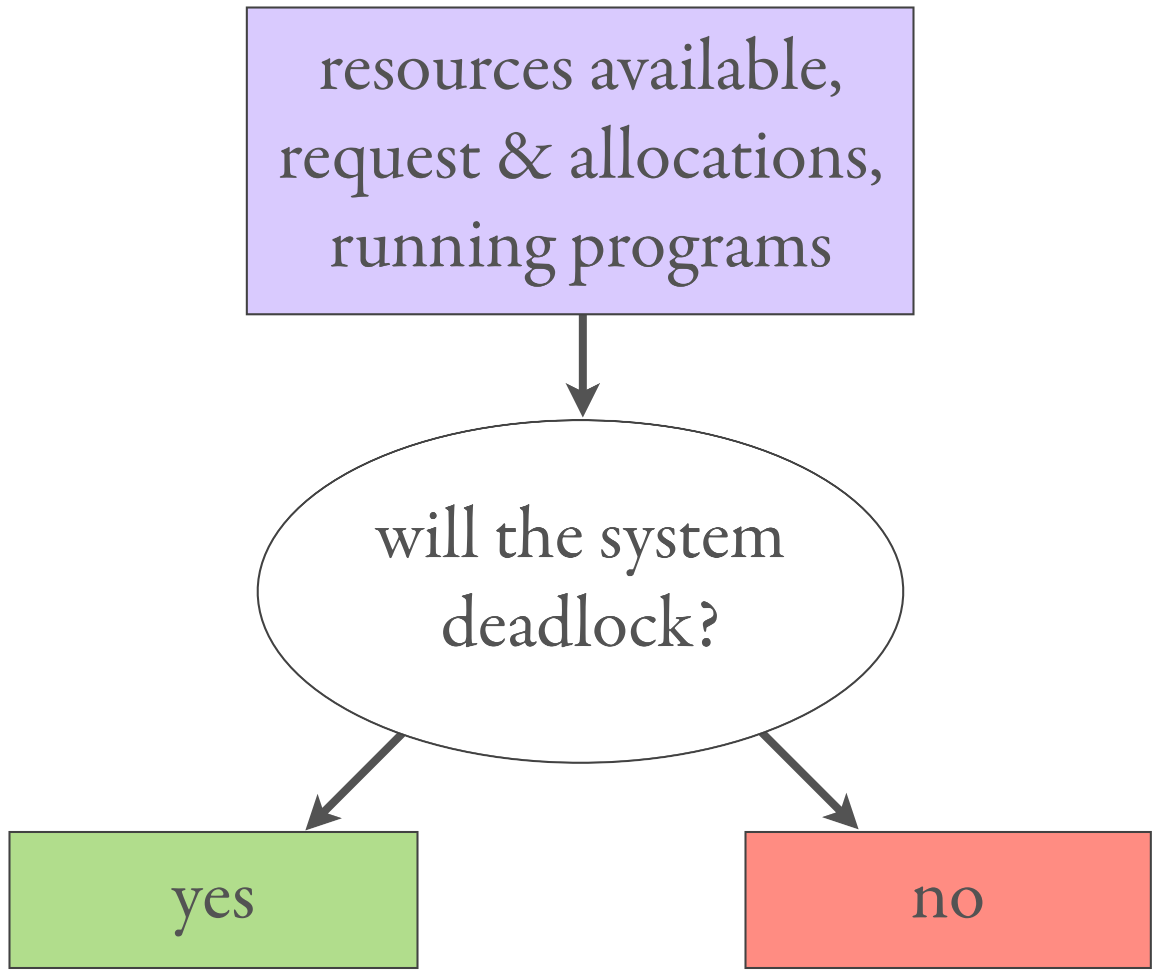
(more motivation: you also win $1M)

if you can prove P ≠ NP, we can *stop looking* for fast solutions to many hard problems

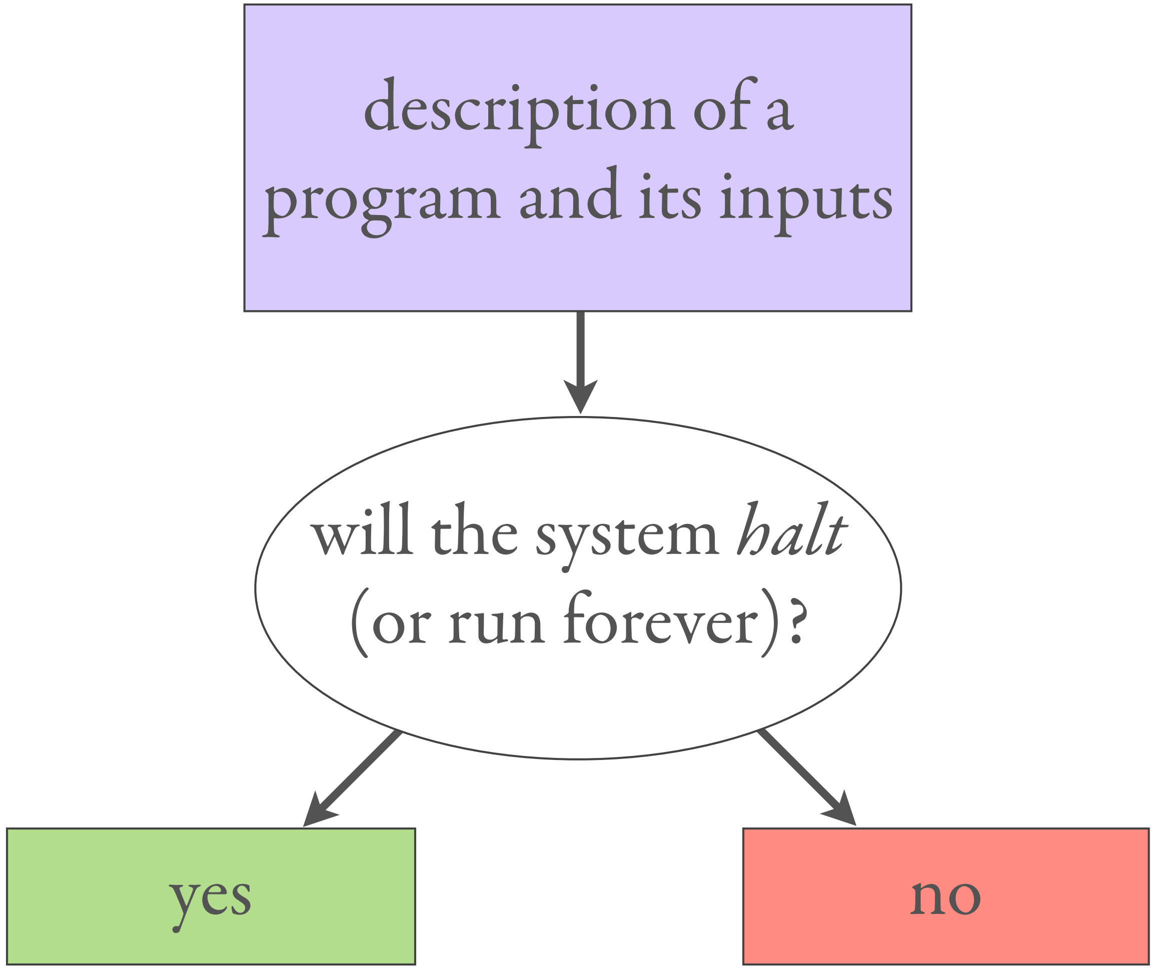
(motivation: you *still* win $1M)



a decision problem



deadlock prevention



the halting problem

e.g., write the function:

```
halt(f) → bool
```

- return true if `f` will halt
- return false otherwise

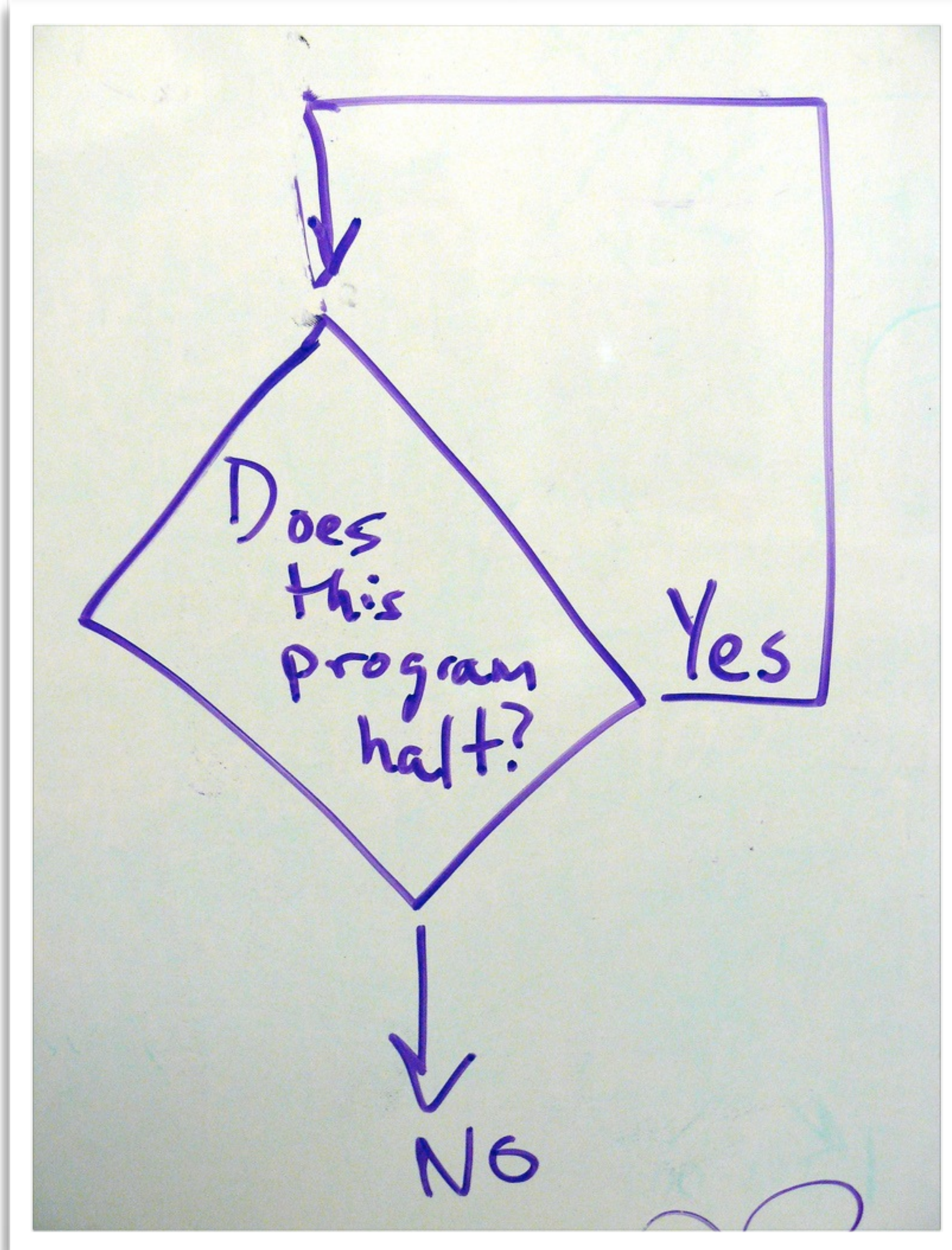
```python
def halt(f):
    # your code here

def loop_forever():
    while True: pass

def just_return():
    return True
```

```python
halt(loop_forever)  # => False

halt(just_return)   # => True
```

```python
def halt(f):
    # your code here

def gotcha():
    if halt(gotcha):
        loop_forever()
    else:
        just_return()
```

```python
halt(gotcha)
```

#$^%&#@!!!


Does this program halt?
Yes
NO

proof by contradiction:
the halting problem is *undecidable*

generally speaking, deadlock prediction *can be reduced to* the halting problem i.e., determining if a system is deadlocked is, in general, *provably impossible*!!

§ Deadlock Detection & Recovery

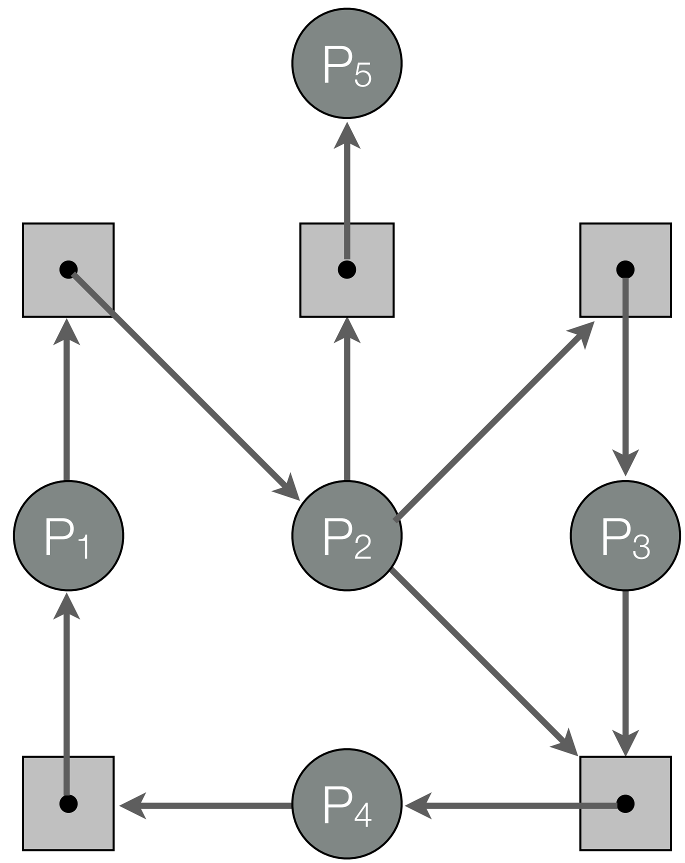
¶ Basic approach: cycle detection

e.g., Tarjan's strongly connected components algorithm; $O(|V|+|E|)$

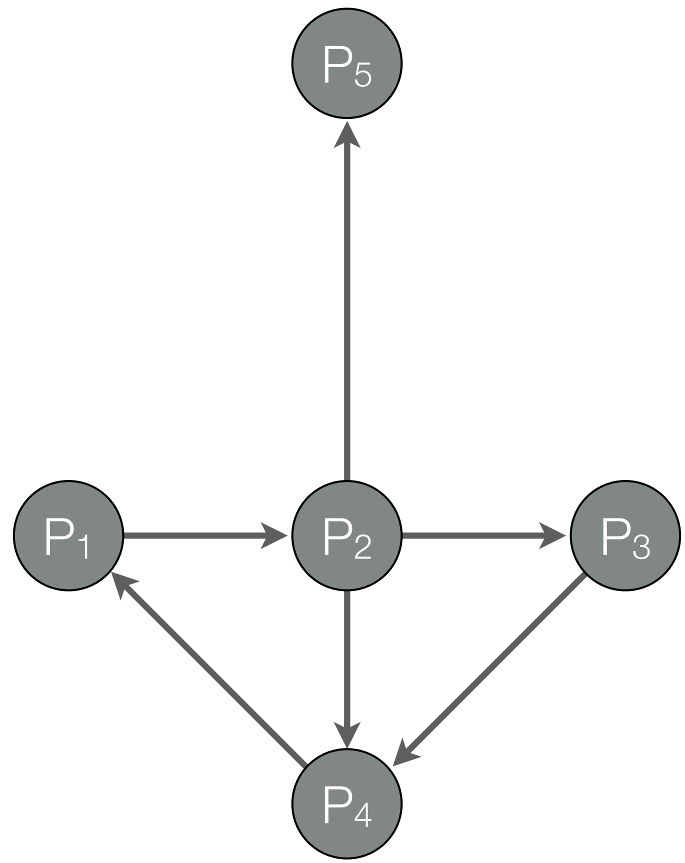
need only run on mutex resources and “involved” processes

… still, would be nice to reduce the size of the resource allocation graph

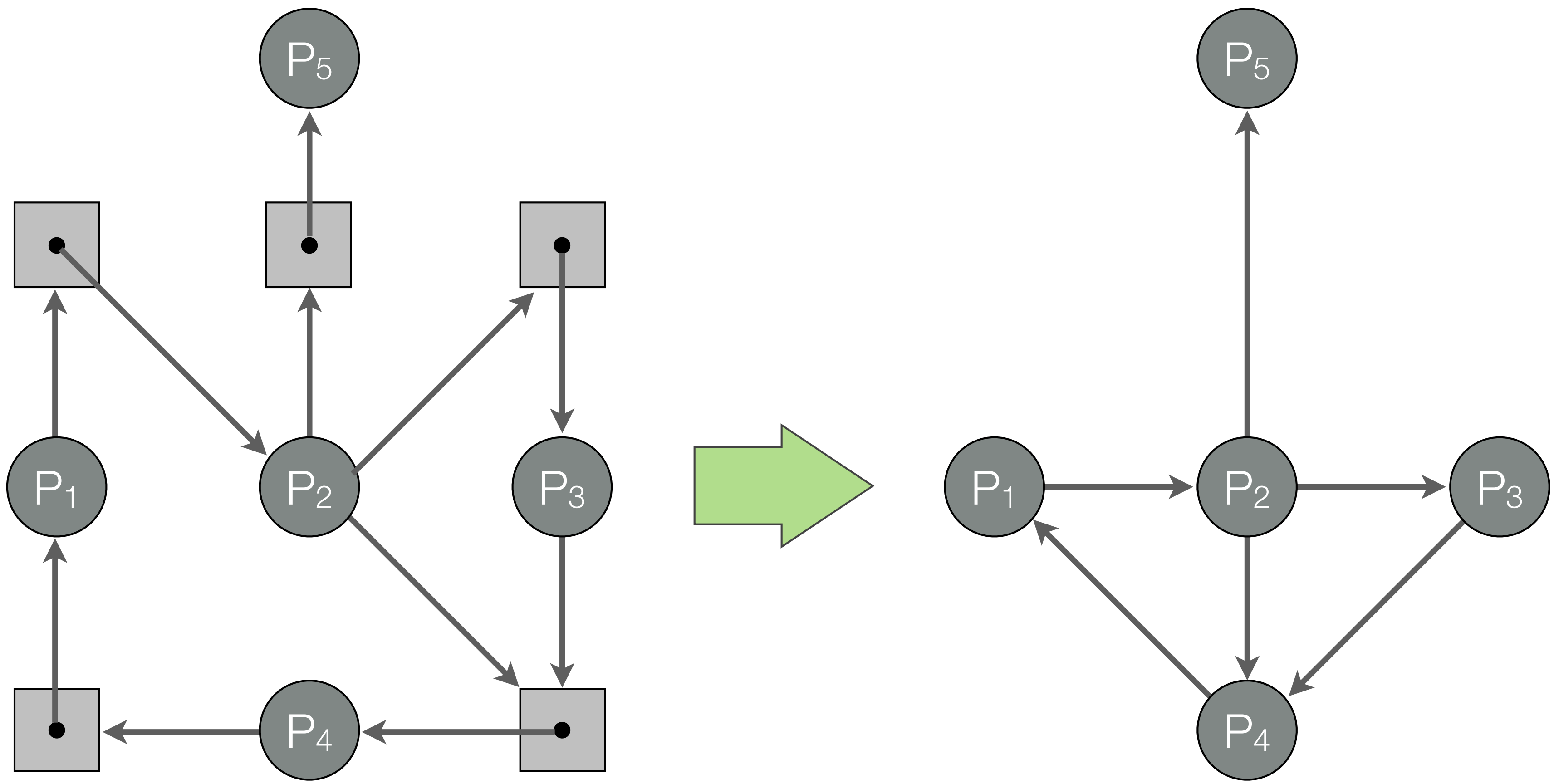
actual resources involved are unimportant —
only care about *relationships between processes*



Resource Allocation Graph



“Wait-for” Graph



Substantial optimization!

… but not very useful when we have multi-instance resources (false positives are likely)

❡ Deadlock detection algorithm

important: do away with requirement of
a priori resource need declarations

new assumption: processes can complete with *current allocation + all pending requests*

i.e., no future requests

unrealistic! (but we have no crystal ball) keep track of all pending requests in:

`request[i][j]` = num of R_j requested by P_i

Detection algorithm

ignore processes that aren't allocated anything

1. finish[i] ← all_nil?(allocated[i]) ∀ i ∈ 1…n
   work ← available
2. Find i: finish[i] = false & request[i][j] ≤ work[j] ∀ j
   If none, go to 4.
3. work ← work + allocated[i]; finish[i] ← true
   Go to 2.
4. If finish[i] ≠ true ∀ i, system is deadlocked.

3 resources: A (7), B (2), C (6)

Allocated

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

Request

	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	0
P_3	1	0	0
P_4	0	0	2

Available

A	B	C
0	0	0

- Not deadlocked: $<P_0, P_2, P_1, P_3, P_4>$
- P_2 requests <0, 0, 1>

¶ Discussion

1. Speed?

1. finish[i] $\leftarrow$ all_nil?(allocated[i]) $\forall$ i $\in$ 1...n
   work $\leftarrow$ available
2. Find i: finish[i] = false & request[i][j] $\leq$ work[j] $\forall$ j
   If none, go to 4.
3. work $\leftarrow$ work + allocated[i]; finish[i] $\leftarrow$ true
   Go to 2.
4. If finish[i] $\neq$ true $\forall$ i, system is deadlocked.

Still $O(N \cdot N \cdot M) = O(N^2 \cdot M)$

2. When to run?

… as seldom as possible!

tradeoff: the longer we wait between checks, the messier resulting deadlocks might be

3. Recovery?

One or more processes must release resources:

- via forced termination
- resource preemption
- system rollback

cool, but how?

Resource preemption only possible with certain types of resources

- no intermediate state
- can be taken away and returned (while blocking process)
  - e.g., mapped VM page

Rollback requires process *checkpointing*:

- periodically autosave/reload process state
- cost depends on process complexity
- easier for special-purpose systems

How many to terminate/preempt/rollback?

- at least one for each disjoint cycle
  - non-trivial to determine how many cycles and which processes!

Selection criteria (who to kill) = minimize cost

- # processes
- completed run-time
- # resources held / needed
- arbitrary priority (no killing system processes!)

Dealing with deadlock is *hard*!

Moral of this and the concurrency material:

- be careful with concurrent resource sharing
- use concurrency mechanisms that avoid explicit locking whenever possible!