Deadlock

Chapter 8

The Deadlock Problem

- A set, *S*, of blocked processes
  - Each holding resources
  - Each waiting to acquire a resource
  - Desired resources all held by other processes in *S*
- We saw this in Dining Philosophers, happens all the time
- Example
  - System has 2 tape drives
  - P_1 and P_2 each have a tape drive and each needs another one.
- Another Example
  - A pair of semaphores, A and B, both initialized to 1

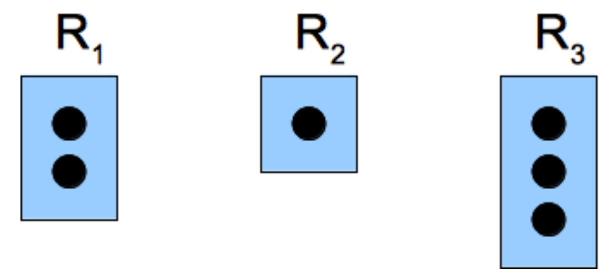
$\mathbf{P_1}$	$\mathbf{P_2}$
`acquire( A );`	`acquire( B );`
`acquire( B );`	`acquire( A );`

Thinking about Deadlock

- *Resource Allocation Graph*: a notation for showing the instantaneous state of a system's resources
  - Makes it easy to talk about particular examples
  - Maintained by the OS (in some form)
- We have resource types: R_1, $R_{2,}$ … R_m
  - e.g., CPU cycles, individual files, blocks of memory, I/O devices, mutex variables
- Each resource type may have multiple instances
  - e.g., maybe you have two tape drives, multiple blocks of memory
  - Each instance of the same resource is equivalent

Resource Allocation Graph

- Resources are notated as a box, with dots inside for the instances:

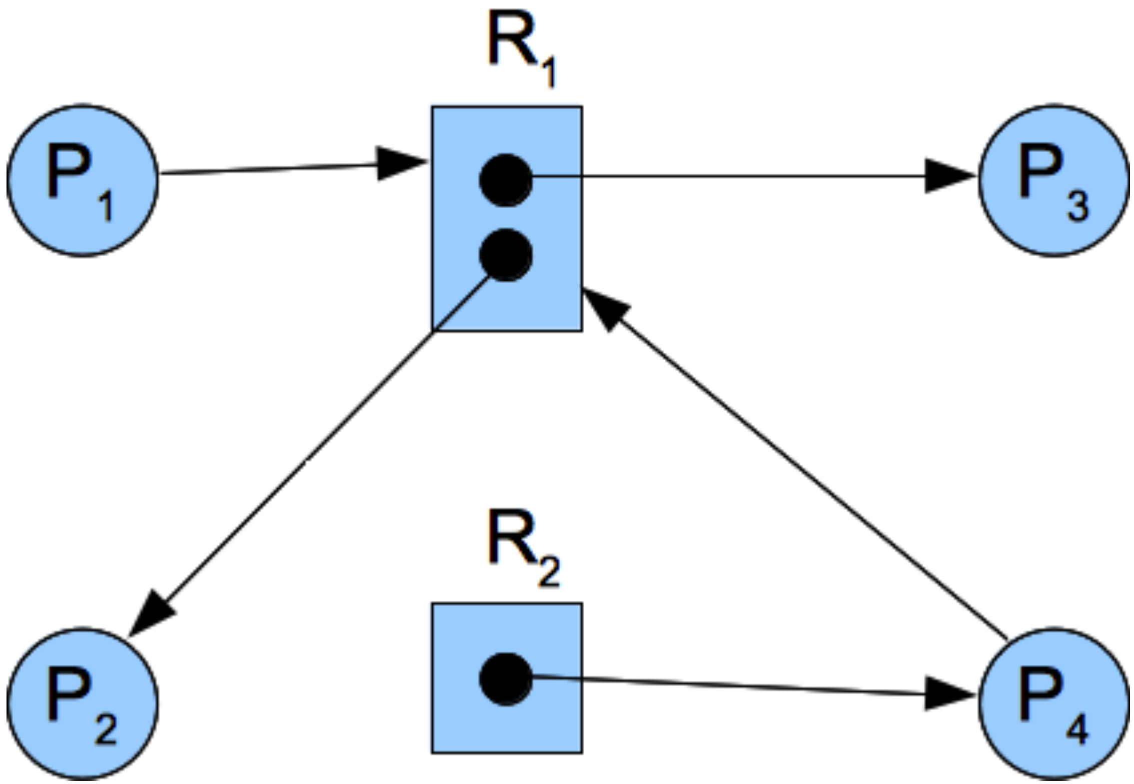


- We have processes: P_1, P_2, … P_n
  - Notated as little circles.

Resource Operations

- Processes work with resources
  - First, they *request* a resource type
  - Then, they are *assigned* a resource instance by the OS
  - Then, they *use* the resource instance
  - Finally, they *release* the resource instance
- Resource Allocation graph captures this state.
  - *Request Edge*: a directed edge from a process to a resource
  - *Assignment Edge*: a directed edge from a resource instance to a process

Resource Allocation Graph



Methods for Dealing With Deadlock

- Prevent
  - Build a system where processes don't have the ability to cause deadlock
- Detect (and recover)
  - Let deadlocks happen, but notice when they do
  - Maybe provide support for getting out of a deadlock
- Avoid
  - Build a system where processes do have the tools to cause deadlock
  - But, watch and regulate process behavior so that they are never able to cause deadlock
  - This is going to be tricky

Methods for Dealing With Deadlock

- Ignore
  - Hope for the best
  - Pretend that deadlocks don’t exist
  - Used by most operating systems, including Unix.
  - Responsibility of the application programmer (for single-process cases)
  - Responsibility of users or the system manager (for multi-process cases)
  - So, we need to be prepared to diagnose and fix our own deadlocks

Why Deadlock Happens

- Four *Necessary Conditions*
  - *Mutual Exclusion*: some resources can only be used by one process at a time
  - *Hold and Wait*: processes can have some resources and ask for more
  - *No Preemption*: OS gets a resource back only when a process is done using it
  - *Circular Wait*: It's possible to build a cycle of processes, P_1, P_2, … P_n such that:
    - P_1 is waiting for a resource held by P_2
    - …
    - P_{n-1} is waiting for a resource held by P_n
    - P_n is waiting for a resource held by P_1

Deadlock Prevention

- Build a system where deadlocks can't occur
  - Maybe enforced by the OS, for all processes
  - Or, maybe enforced by the application programmer, among threads in a process
- Recall, there are four *necessary conditions* for deadlock
- To have deadlock, **all of these must occur**
- So, let's just prohibit one of them, that's *deadlock prevention*

Deadlock Prevention

- Mutual Exclusion
  - So, no mutual exclusion on any resource … ever
  - The OS does this kind of thing all the time, it virtualizes hardware resources
  - Can we do this with all resources?
  - I suppose so
    - But, it would give us a system that's harder to program for
    - e.g., we would have to give up critical sections
    - .. so, probably a bad idea in general.

Deadlock Prevention

- Hold and Wait
  - So, we never let processes ask for more resources after they already have some
  - I think maybe we could do this
    - Maybe processes must ask for all resources at startup
    - Or, maybe they could give up everything and then re-request at any time
  - Problems?
    - Reduced resource utilization
    - Possible starvation

Deadlock Prevention

- No preemption
  - So, we want **no** no preemption … that's preemption
  - Can we really permit preemption on all resources?
    - Maybe you risk preemption whenever you ask for more resources
    - e.g., maybe you temporarily give up what they have when you ask for new resources that aren't available
  - But, some resources won't work well this way
  - Of course, we could always just kill processes and take their resources (that's preemption)
  - Any problems with this?

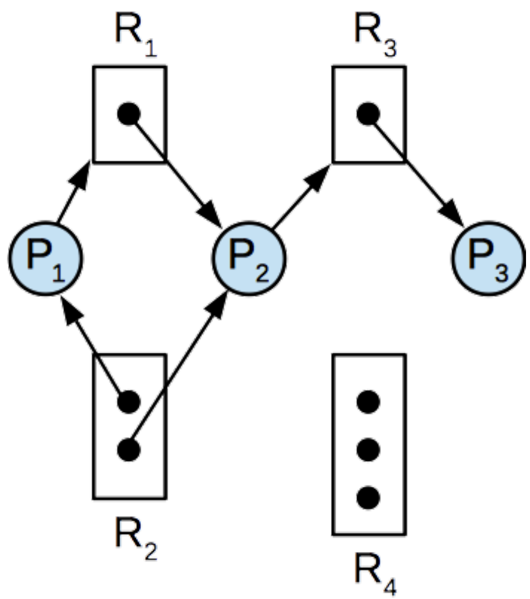
Deadlock Prevention

- Circular Wait
  - So, no circular wait
  - An idea: impose a total ordering on all resource types
  - Processes have to ask for R_1 before R_3, even if they need R_3 first.
  - We've already used this technique once
  - It's a very reusable approach to correcting deadlock in our own programs.

Deadlock Detection

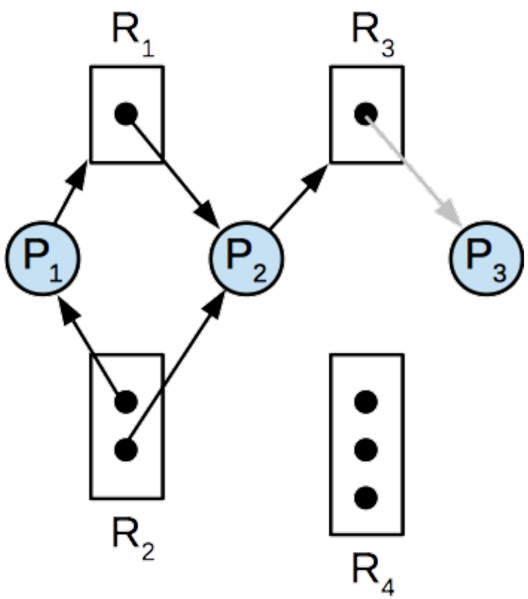
- Maybe we can let deadlocks happen
- Notice when they occur
- And then respond somehow
- That’s a good idea, right?
  – It wouldn’t restrict the execution environment for our processes
  – It wouldn’t hurt resource utilization or promote starvation … most of the time.

Fun With Deadlock Detection

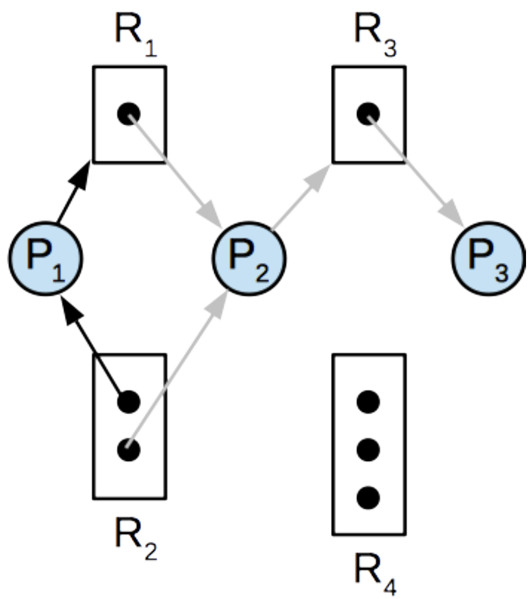


- Are any of these processes runnable?
- Is there a deadlock?

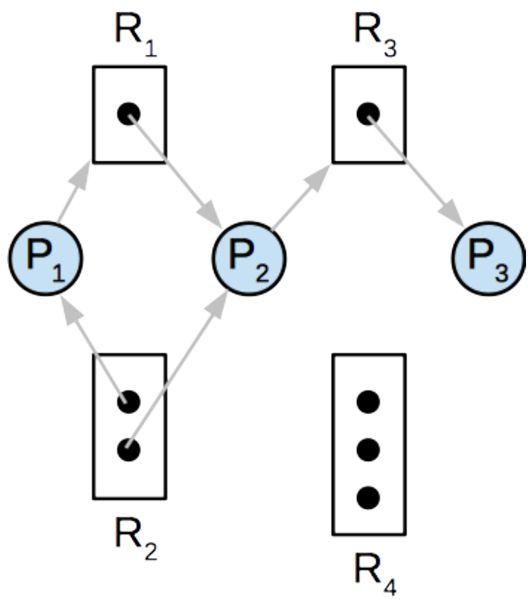
Fun With Deadlock Detection



Fun With Deadlock Detection

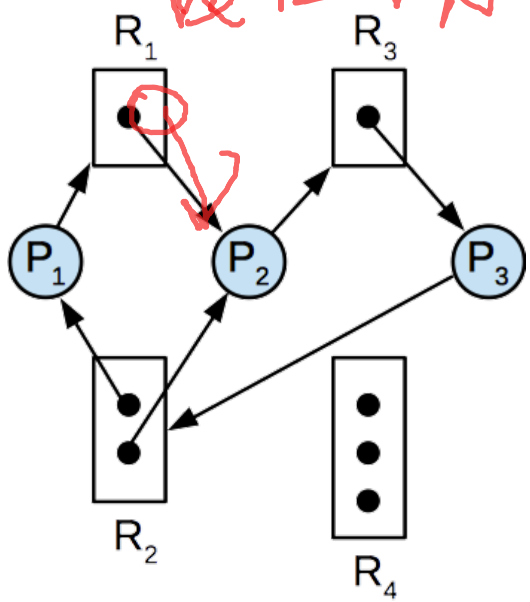


Fun With Deadlock Detection



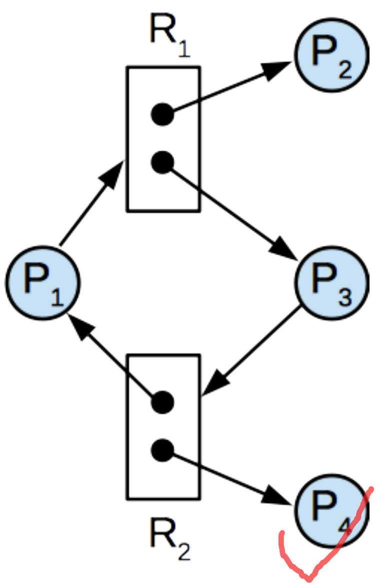
Fun With Deadlock Detection

被P2用了



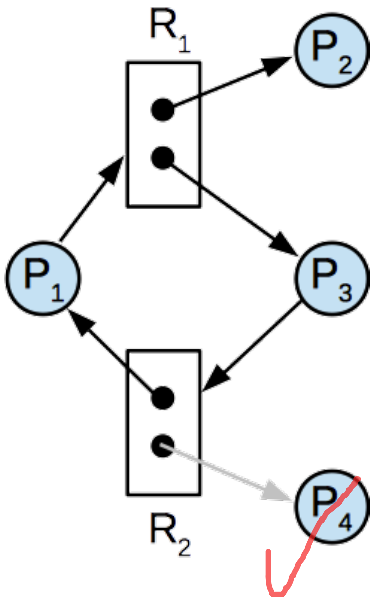
- Are any of these processes runnable?
- Is there a deadlock?

Fun With Deadlock Detection

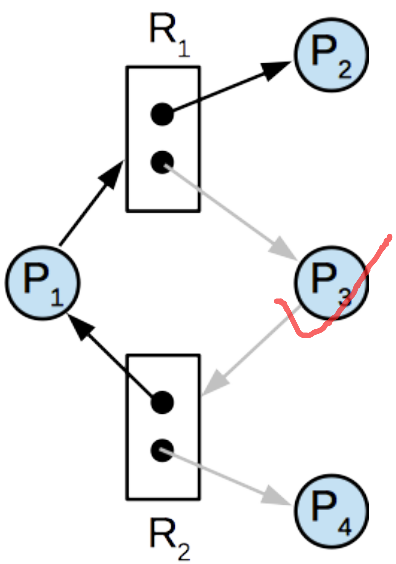


- Are any of these processes runnable?
- Is there a deadlock?

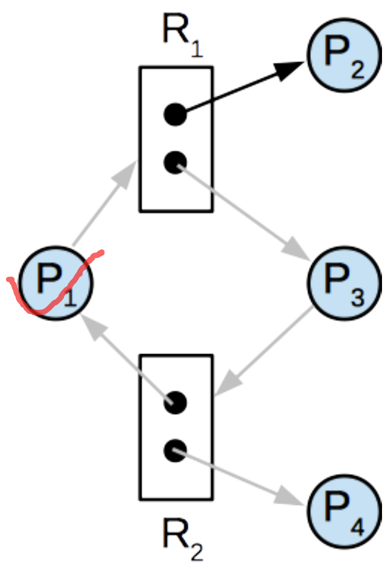
Fun With Deadlock Detection



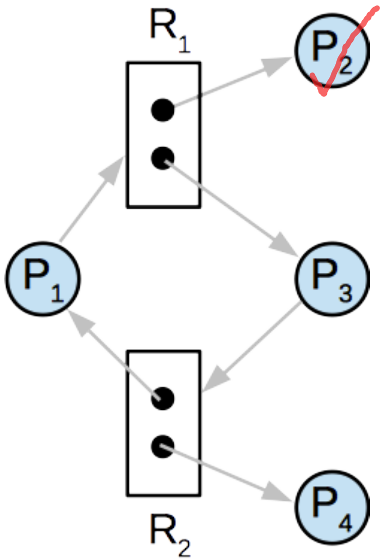
Fun With Deadlock Detection



Fun With Deadlock Detection



Fun With Deadlock Detection



No DL

Deadlock Detection

- If the graph contains no cycle → No Deadlock
- If the graph contains a cycle →
  - If all resources just have one instance, then there's a deadlock
  - If some resources have multiple instances, maybe there's a deadlock, maybe not
- For the general case, we have to run our own *deadlock detection algorithm*
  - Fortunately, we just invented one

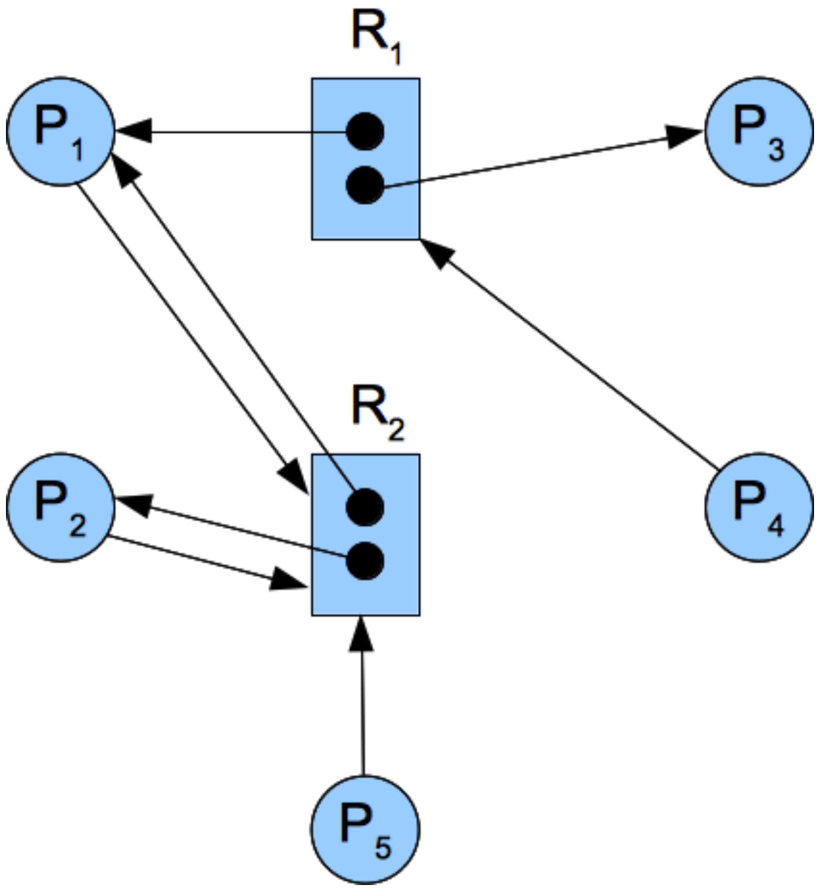
Deadlock Detection

- Save a clean copy of the resource allocation graph

  while (there's some process *P* where **all of** *P*'s
  outstanding requests can be satisfied) {
  // Pretend *P* finishes without asking for more
  Remove all request and assignment edges for *P*
  }

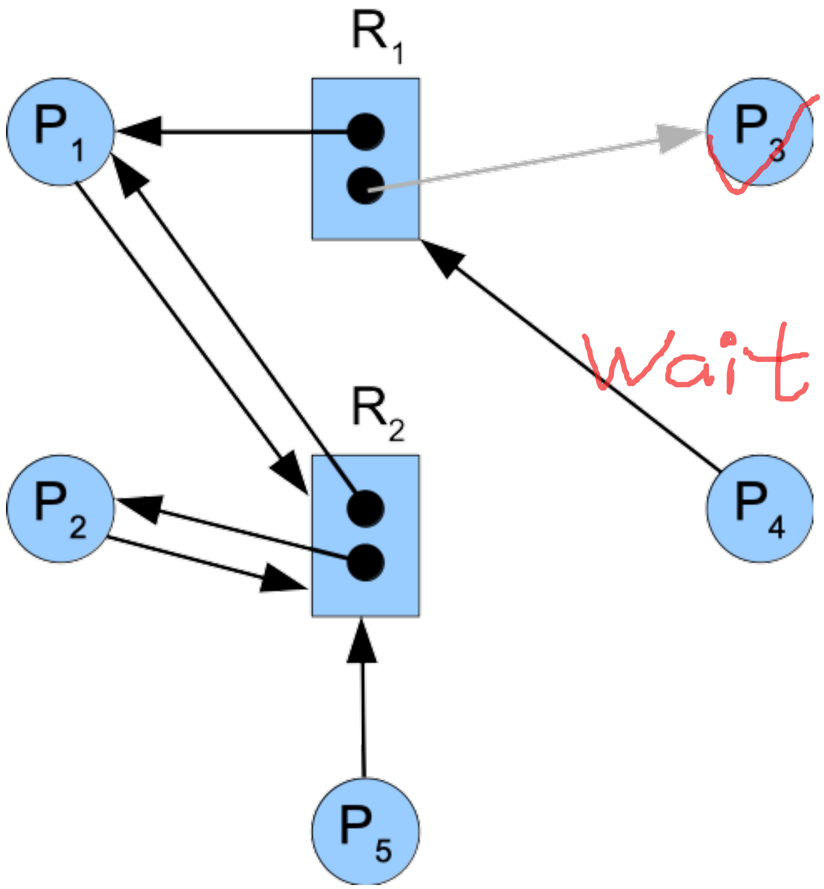
- If any processes remain
  - You have a deadlock
  - Find a cycle among them to find one of the processes responsible
- Restore original resource allocation graph

Deadlock Detection



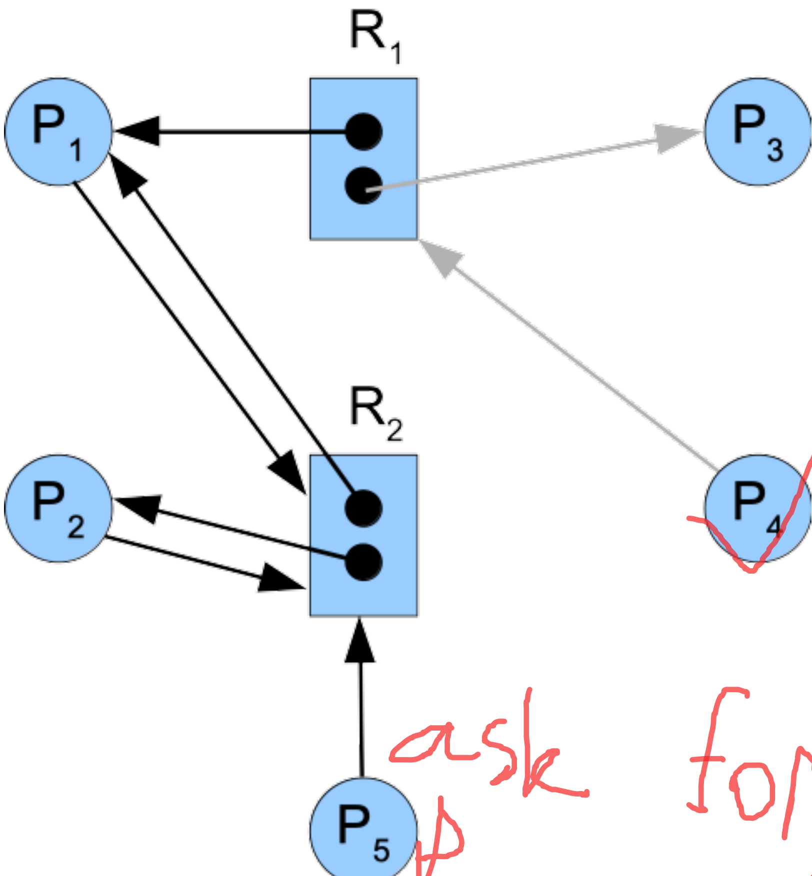
- Is there a deadlock
- Who's to blame

Deadlock Detection



- P_3 can finish first.

Deadlock Detection



- P_3 can finish first.
- Then P_4 can finish.
- But then the remaining process are stuck.
- … but they're not all part of the deadlock.
- Killing P_5 won't help.

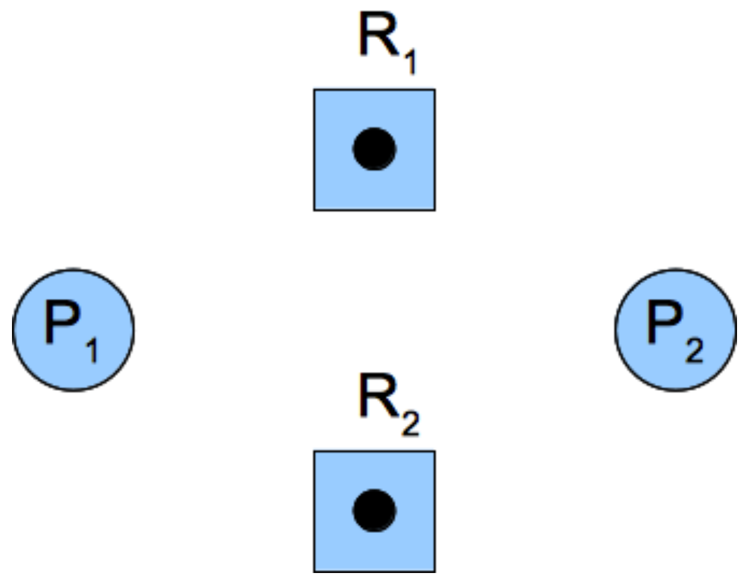
ask for P_2, can't get it

Deadlock Recovery

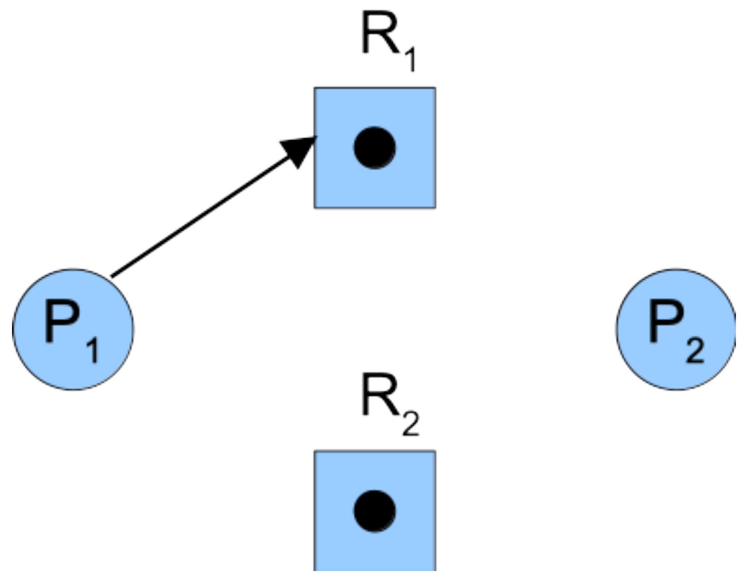
- What to do once we've detected a deadlock?
  - Waiting won't help
  - Some process is going to have to pay!
  - Killing a process might fix things, but which one should we kill?
    - First, one that is on a cycle that the deadlock detection algorithm can't finish
    - Among these, who?
      - A process with lots of resources?
      - A young process?
      - One that's on multiple cycles?

Deadlock Avoidance

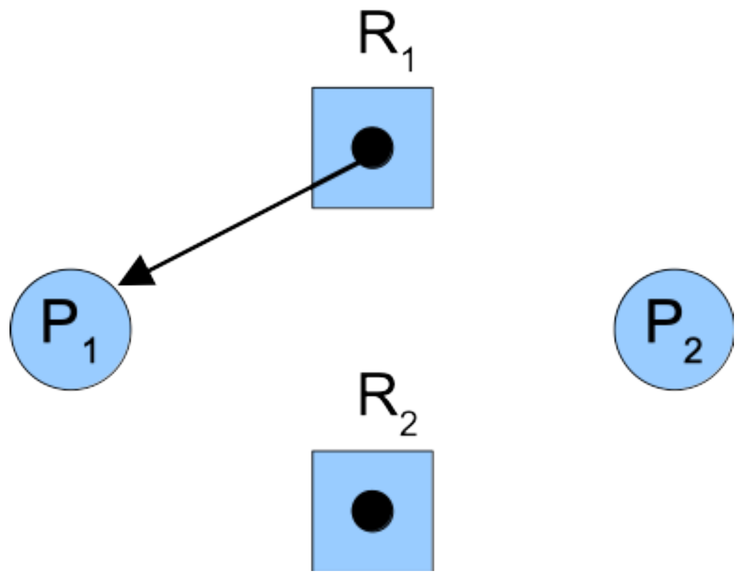
- The idea: stay on a safe path
  - Never grant a request that could lead to a deadlock in the future
- Can we do this?



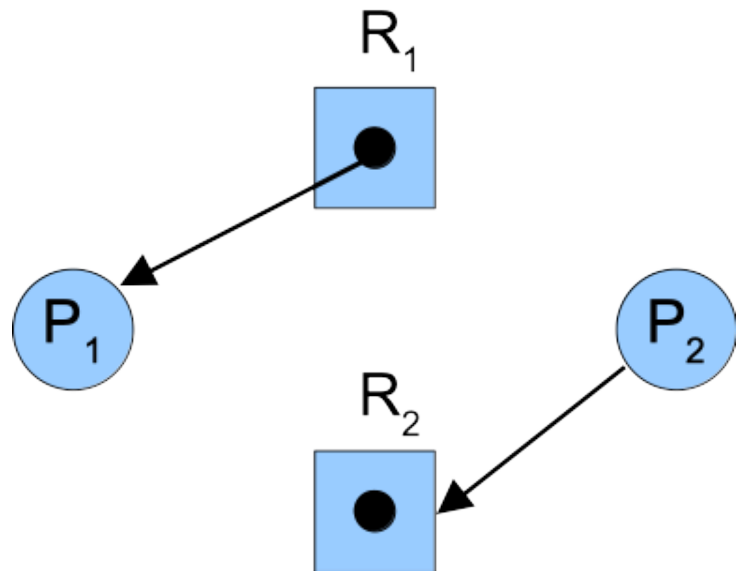
Deadlock Avoidance



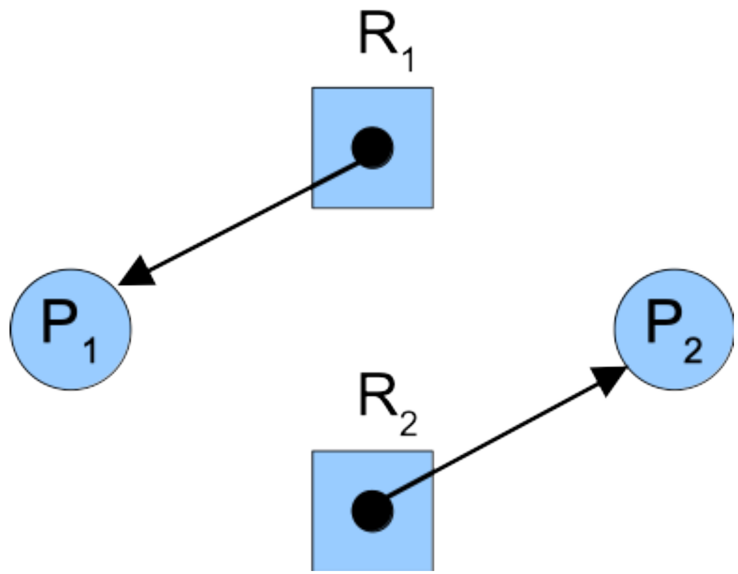
Deadlock Avoidance



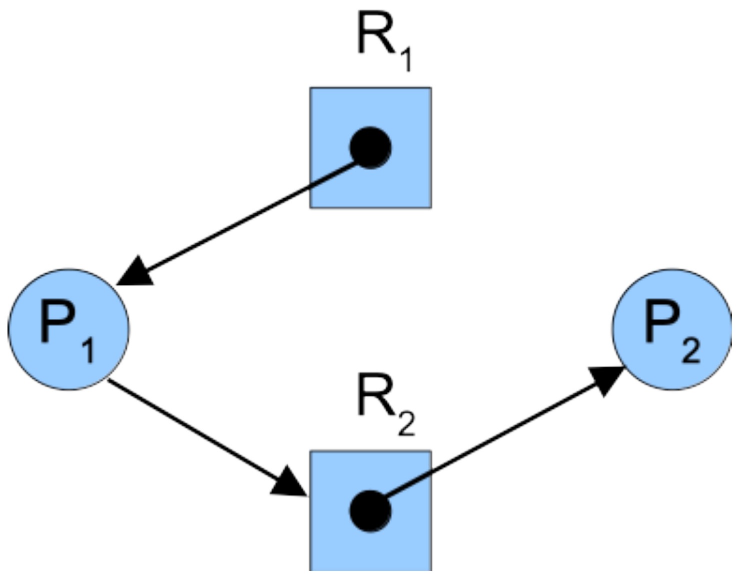
Deadlock Avoidance



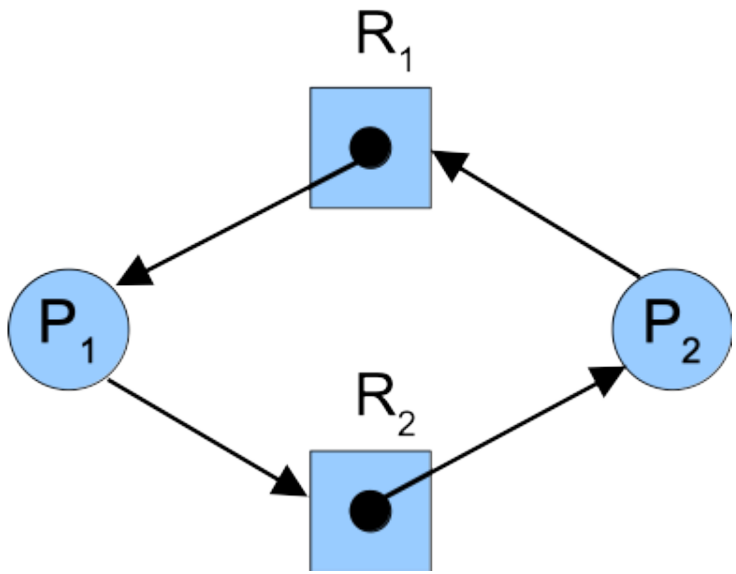
Deadlock Avoidance



Deadlock Avoidance



Deadlock Avoidance



Deadlock Avoidance

- Unfortunately, this requires knowledge of the future
- OK, let’s have the OS know something about the future
  - When a process is admitted, it has to report how many of each resource type it may need
  - Assume the OS has *a priori* knowledge of what resources a process may request
- Deadlock avoidance
  - For each request, decide whether the process should wait
  - Must consider resources currently available, allocated and the possibility of future requests

New Notation for Deadlock Avoidance

- Processes must claim all resources at startup
- A *claim edge* from P_i to R_j indicating that P_i may ask for R_j in the future
  - Notated with a dashed line
- A claim edge becomes a request edge when the process makes the request
- Request edge converted to an assignment edge, when the process gets the resource
- Turns back into a claim edge when the process releases the resource (it might want it again later)

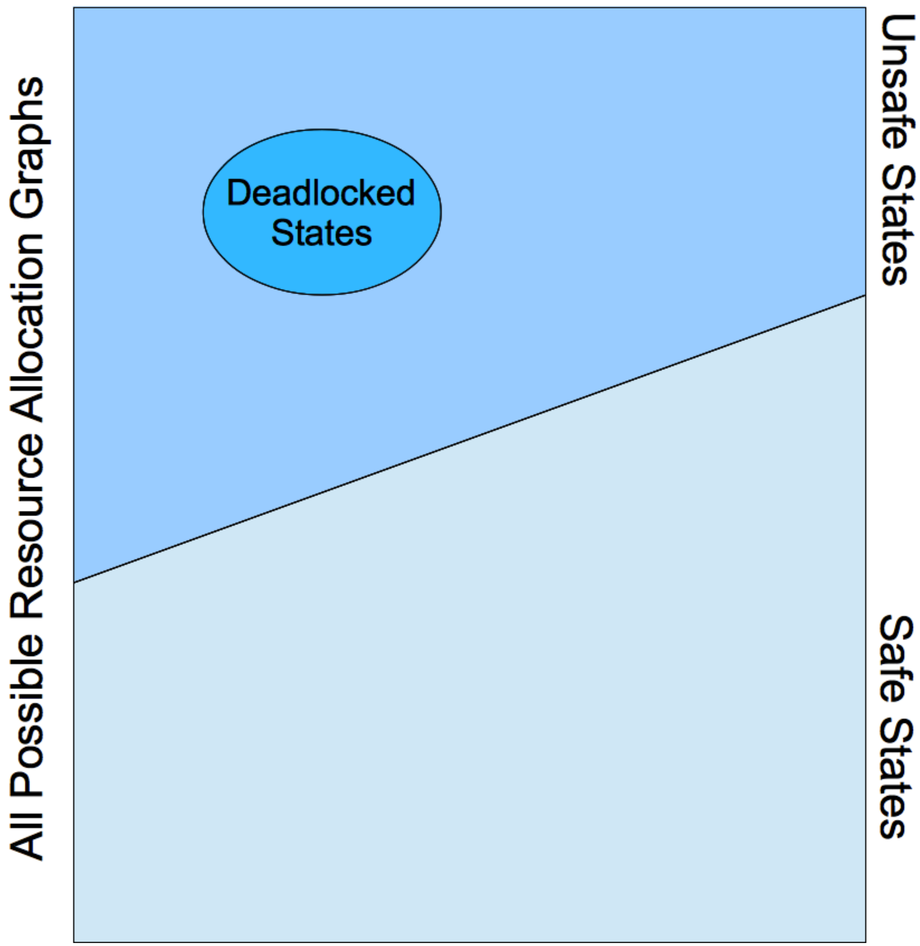
Deadlock Avoidance

- So, on a resource request, ~~three~~ things can happen: four
  - Sure, have the resource right now
  - Wait on the resource, because it's in use
  - Wait on the resource, it's available but I'm worried about deadlock if I give it to you
  - Die you evil process, that's not one of the resources you said you might request

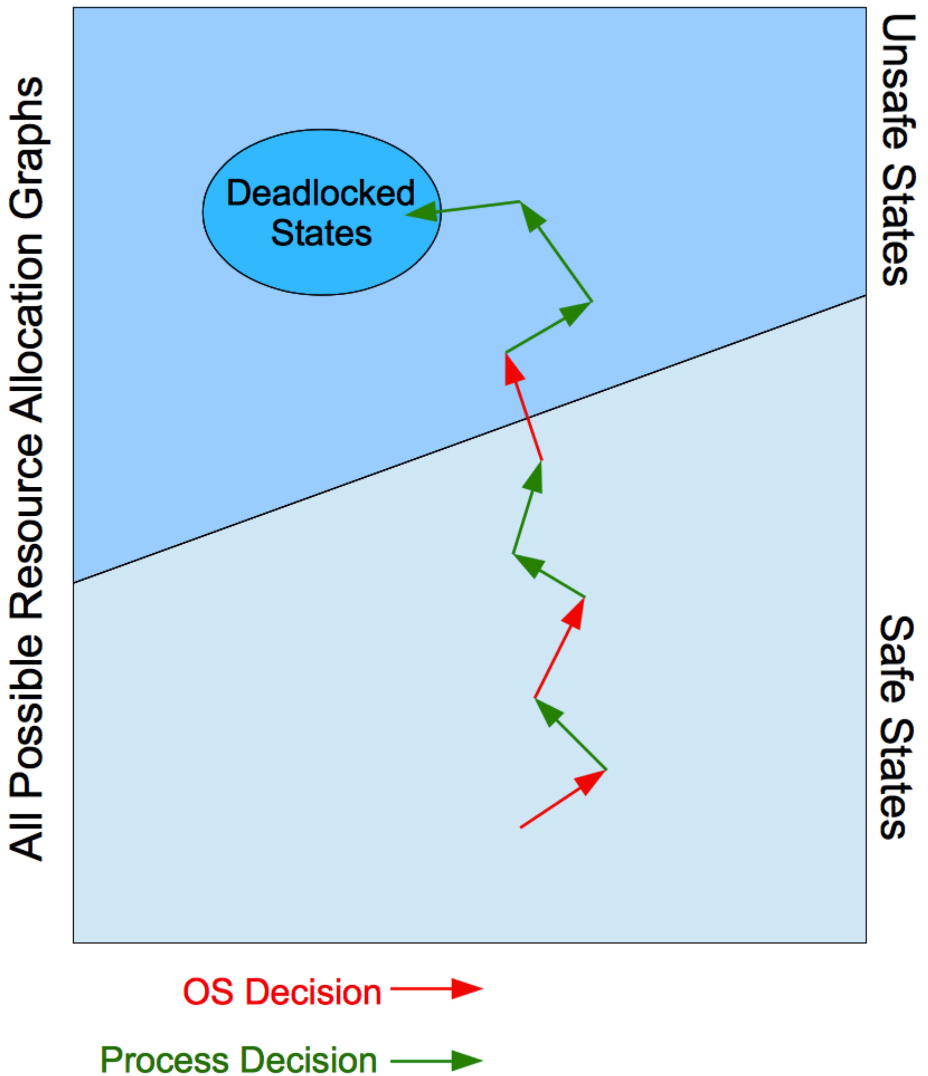
Deadlock Avoidance

- Define a *safe state* for the resource allocation graph
  - In a safe state, processes can’t cause deadlock, no matter what they do
  - In an *unsafe state*, maybe you’re not deadlocked, but:
    - If you’re lucky, maybe processes won’t actually create a deadlock (in the future)
    - But, they could create a deadlock, if they start asking for more resources
    - So, in an unsafe state, whether or not we get a deadlock is up to the processes, it’s out of the OS’s control.

Safe, Unsafe and Deadlocked States



Safe, Unsafe and Deadlocked States



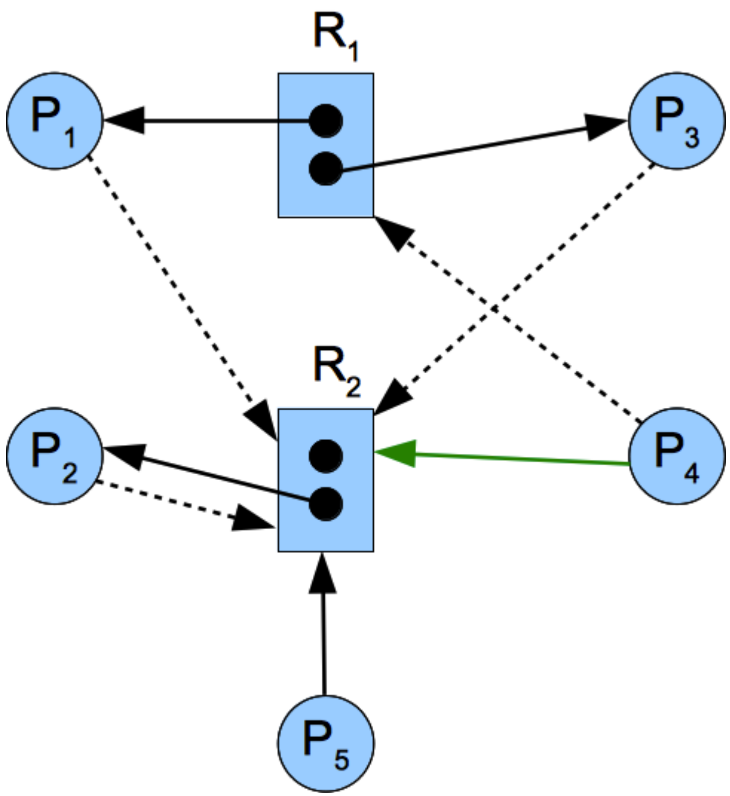
Deadlock and Safe States

- So, for deadlock avoidance, all we need is a safe state detector.
  - Before the OS grants a request, it checks to see if the resulting graph is in a safe state.
  - If so, it grants the request
  - If not, it make the process wait
- This is the *banker's algorithm*

Safe State Detection

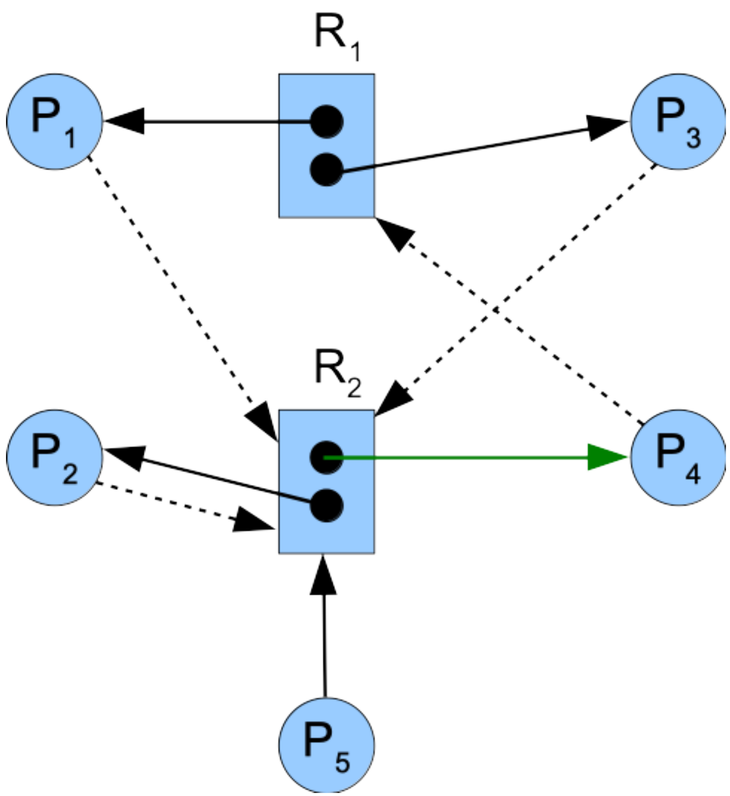
- Make a copy of the Resource Allocation Graph
- Pretend we grant the request (to see if it's safe)
- Now, see if we're guaranteed we can finish up all processes
  - Turn all claim edges into request edges (worst possible scenario, all processes request everything they can)
  - Run the deadlock detection algorithm
- If it's possible to finish up all processes, we're safe → Grant the request
- If not, make requesting process wait.
- Oh, and restore the state of the graph when you're done

Deadlock Avoidance Practice



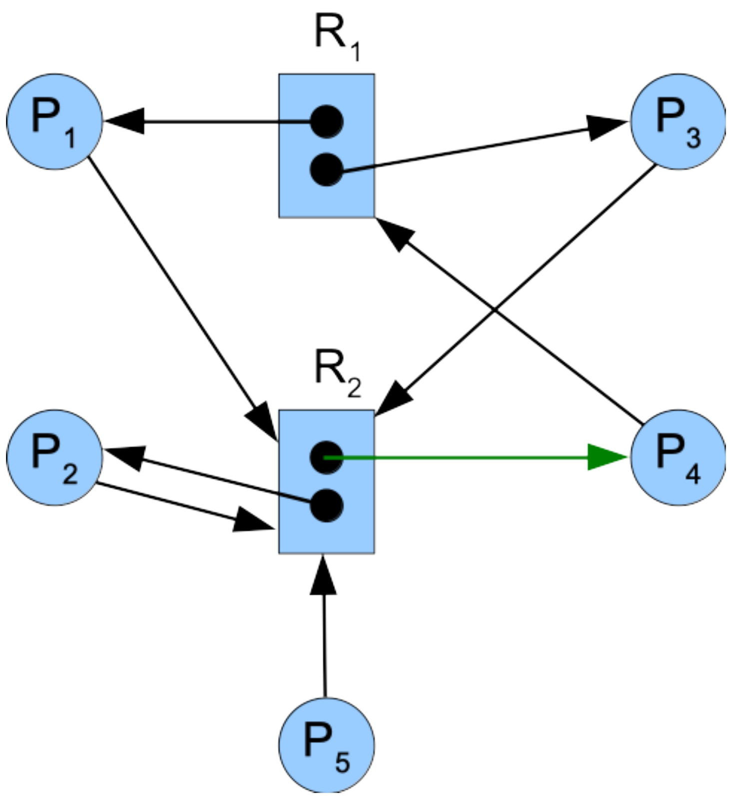
- Should the OS grant the green request?

Deadlock Avoidance Practice



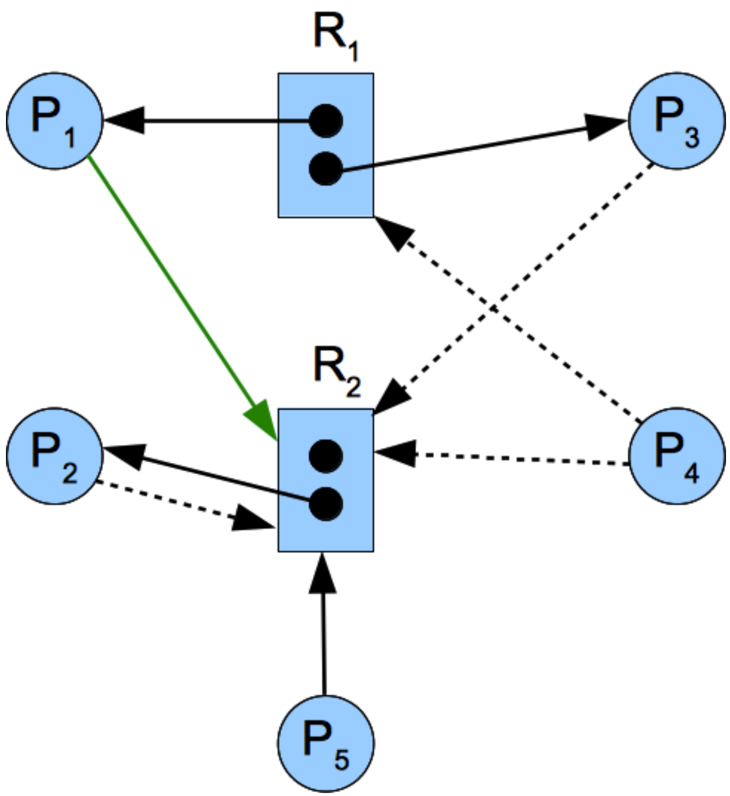
- Consider what could happen if you granted the request.

Deadlock Avoidance Practice



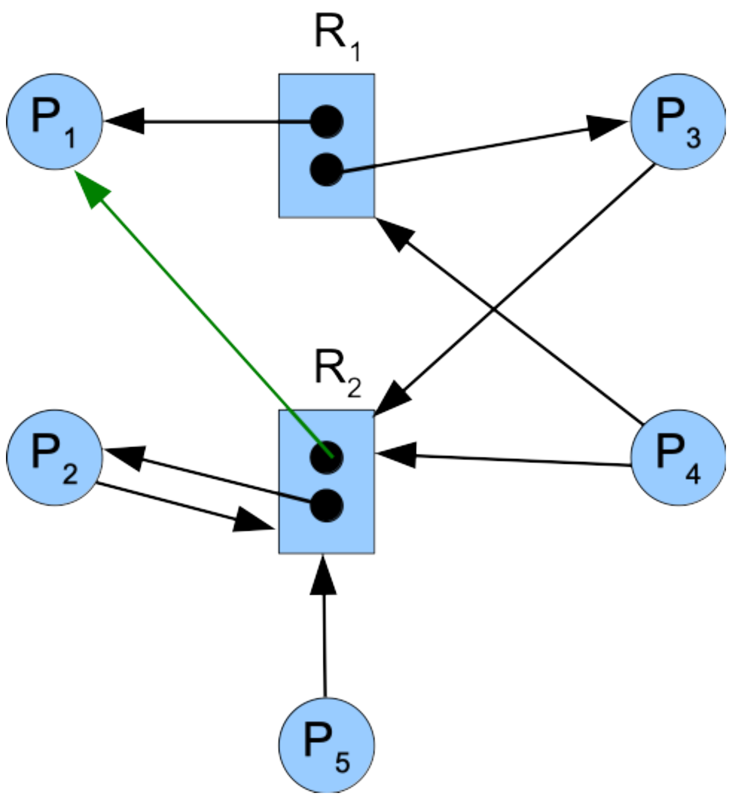
- And then every process requested everything it could.
- Now, we have a deadlock.

Deadlock Avoidance Practice



- What if P_1 requested R_2 instead?

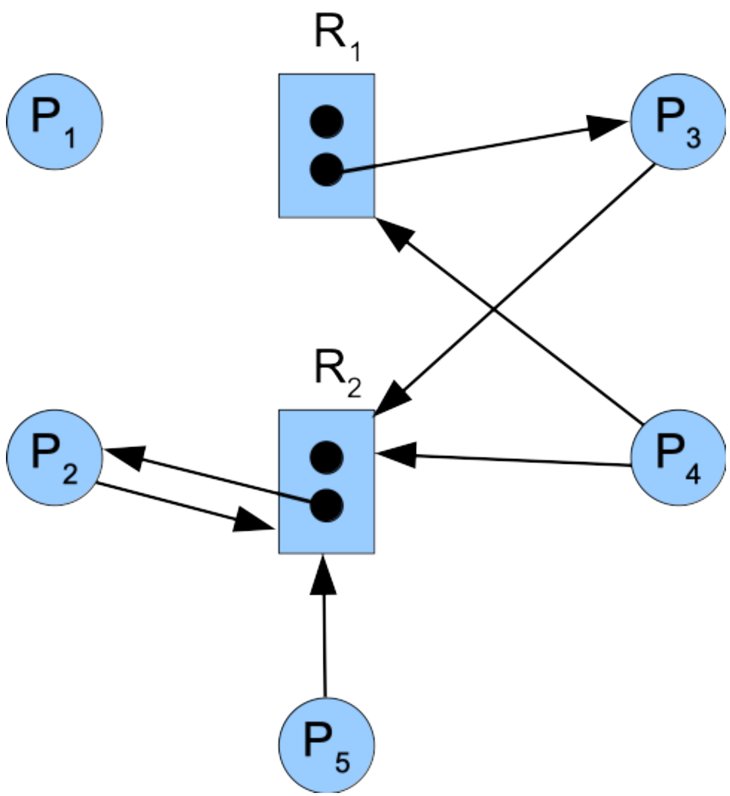
Deadlock Avoidance Practice



- What if we granted the request
- … and every process requested everything it had claimed?

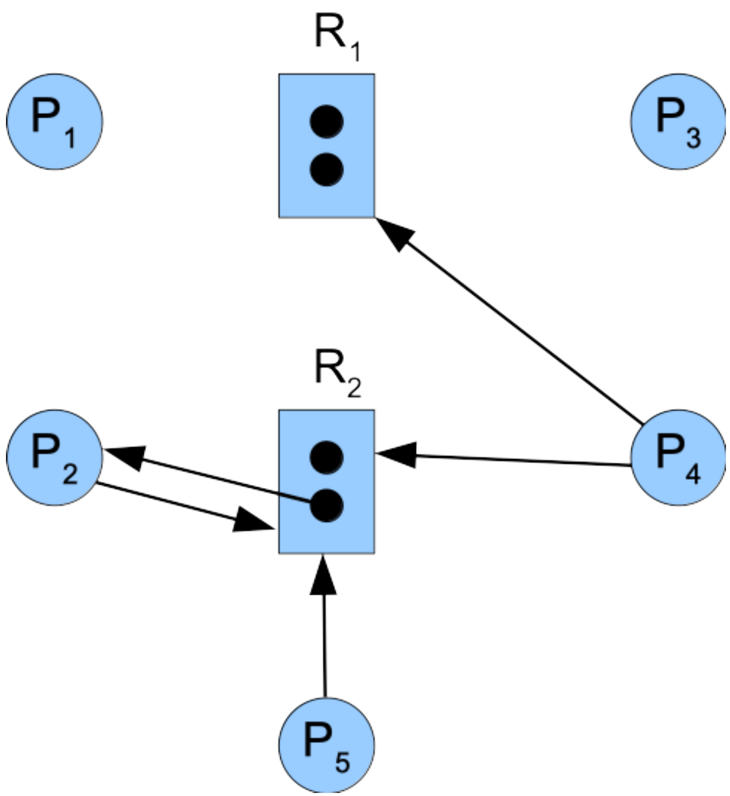
- Here, P_1 can still finish.

Deadlock Avoidance Practice



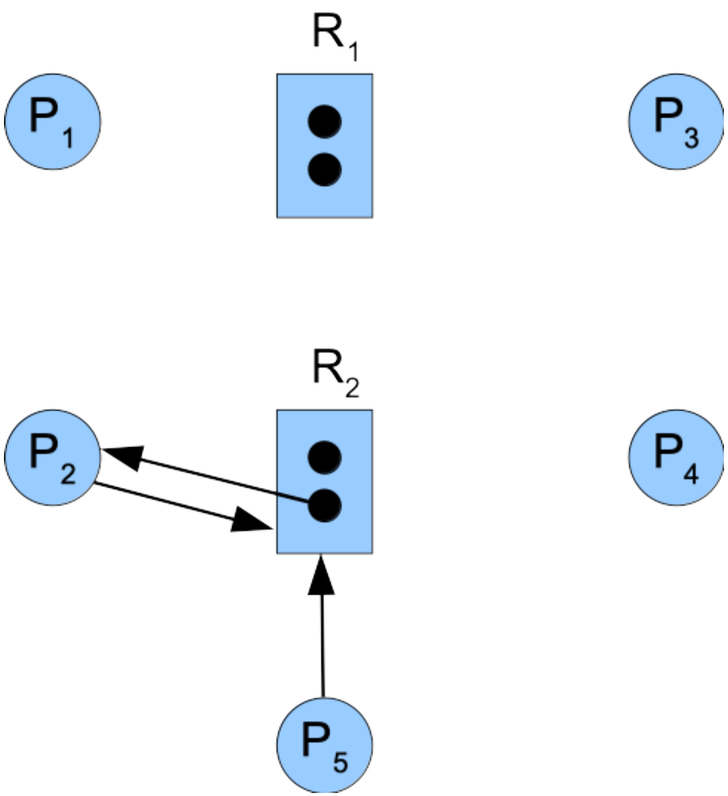
- Then, P_3 could finish.

Deadlock Avoidance Practice



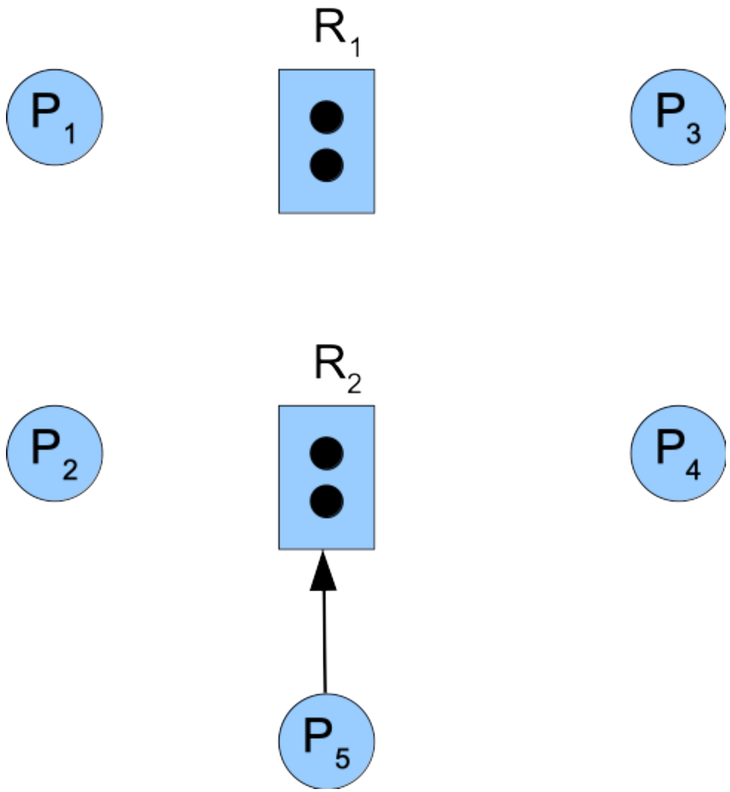
- Then, P_4 could finish.

Deadlock Avoidance Practice



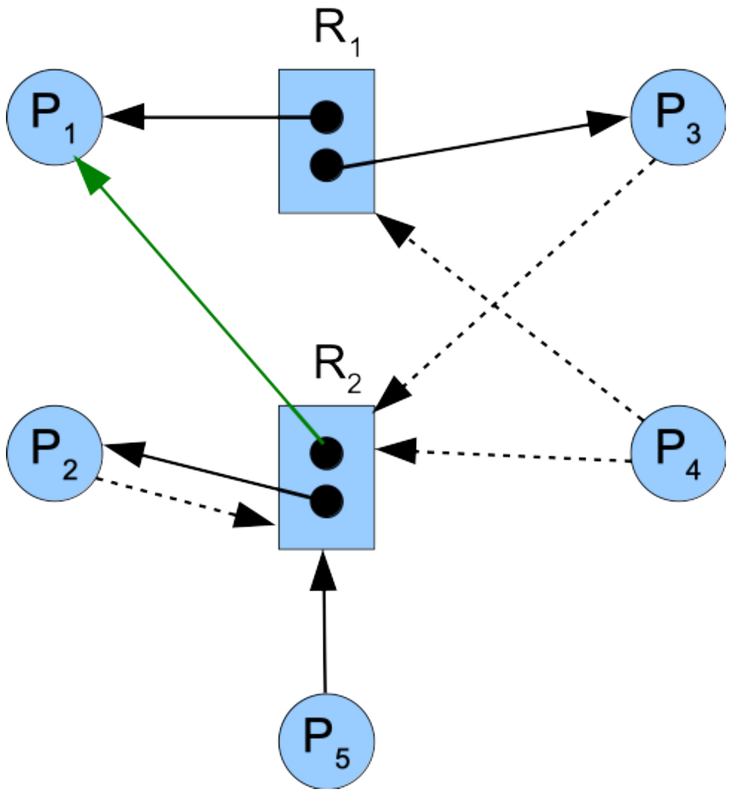
- Then P_2

Deadlock Avoidance Practice



- Then P_5

Deadlock Avoidance Practice



- So, it's safe to grant an instance of R_2 to P_1

Deadlock Avoidance Summary

- If a system is in a safe state → No deadlocks, no matter what processes do
- If a system is in an unsafe state → Possibility of a deadlock in the future, but it's up to the processes
- Avoidance → Ensure that a system never enters an unsafe state.
  - Reduced resource utilization
  - OS: "The resource you want is available, but I'm afraid of what will happen if I let you have it."

Summary

- Deadlock, definition and necessary conditions
- Resource allocation graph notation
- Three techniques for dealing with it
  - Deadlock prevention
  - Deadlock detection (and recovery)
  - Deadlock avoidance