Deadlocks

ICS332

Operating Systems

Real-Life Deadlock

Three kinds of OS-deadlock solutions:
(i) have mechanisms so that a deadlock never happens in the first place
(ii) detect that we're in a deadlock, and do something to fix it
(iii) do nothing and when things don't work have the "operator" reboot it all

Deadlocks

- Early 20th Century Kansas legislature proposed bill: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone" *(… likely bogus)*
- Deadlock with two threads and two resources (see Figure 7.4: Java example)

Thread #1	Thread #2
lock(A)	lock(B)
lock(B)	lock(A)
unlock(B)	unlock(A)
unlock(A)	unlock(B)

- Typically it's the responsibility of the programmer to avoid deadlocks
  - Deadlocks should be rare if the burden is placed on programmers who are highly motivated to avoid deadlocks
  - A manual restart (i.e., kill-restart) is always an option
  - Therefore, avoid making the OS more complicated and let users fend for themselves
  - e.g., Windows/Linux provide no help in this matter
- We're going to look at what OSes could provide, because understanding this leads us to understand how to avoid deadlocks in our own programs

System Model

- System consists of
  - some resources of different resource types: $R_1, R_2, \ldots, R_m$
    - There could be one or more resource in each type
    - e.g., Physical: 4 printers, 2 network cards
      - each resource type is protected by an associated lock
      - either visible to the application, or within the Kernel
      - e.g., when you do an open(), there is a lock in the Kernel for that file
  - processes: $P_1, P_2, \ldots, P_n$
- Each process can:
  - request a resource of a given type
    - And block/wait until one resource instance of that type becomes available
  - use a resource
  - release a resource

Deadlock State

- We have a deadlock if every process P_i is waiting for a resource instance that is being held by another process
- A deadlock can arise only if all four conditions hold
  - Mutual Exclusion
    - At least one resource is non-sharable: at most one process at a time can use it
  - Hold-and-Wait
    - At least one process is holding one resource while waiting to acquire others, that are being held by other processes
  - No preemption
    - A resource cannot be preempted (a process needs to give it up voluntarily)
  - Circular Wait
    - There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that
      - P_i is waiting for a resource that is held by P_{i+1}, $0 \leq i < n$
      - P_n is waiting for a resource that is held by P_0

Deadlock State

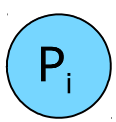
- The four conditions:
  - Mutual Exclusion
  - Hold-and-Wait
  - No preemption
  - Circular Wait

- Note that "circular wait" implies "Hold-and-Wait"
  - It's useful to separate them, as we'll see later

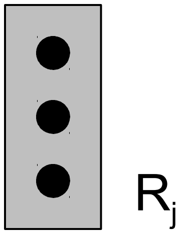
- The four conditions together are only a necessary condition
  - If the four conditions hold, there may be a deadlock
  - If there is a deadlock, then the four conditions hold

Resource Allocation Graphs

- Describing the system can be done precisely and easily with a system resource-allocation graph
- The graph contains:
  - A set of vertices, V, that contains
    - One vertex for each process: $\{P_1, P_2, ..., P_n\}$
    - One vertex for each resource type: $\{R_1, R_2, ..., R_m\}$
      - Which indicates the number of resource instances for that type



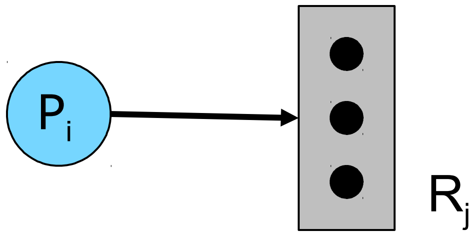
vertex for process P_i

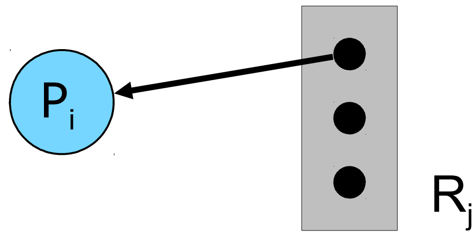


vertex for resource type R_j with 3 resource instances

Resource Allocation Graphs

- The graph contains:
  - A set of directed edges, E, that contains
    - request edge: from P_i to R_j if process P_i has requested a resource of type R_j
      - points to the resource type rectangle
    - assignment edge: from a R_j instance to process P_i is P_i holds a resource instance of type R_j
      - points from a dot inside the resource type rectangle

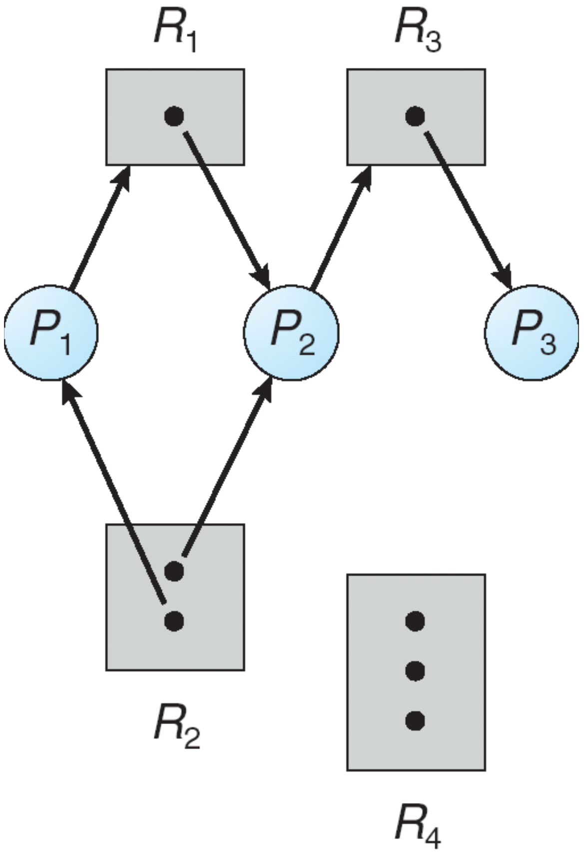




- If a resource request can be fulfilled, then a request edge is transformed into an assignment edge
- When a process releases a resource, the assignment edge is deleted

Example Resource Graph

- Figure 7.1



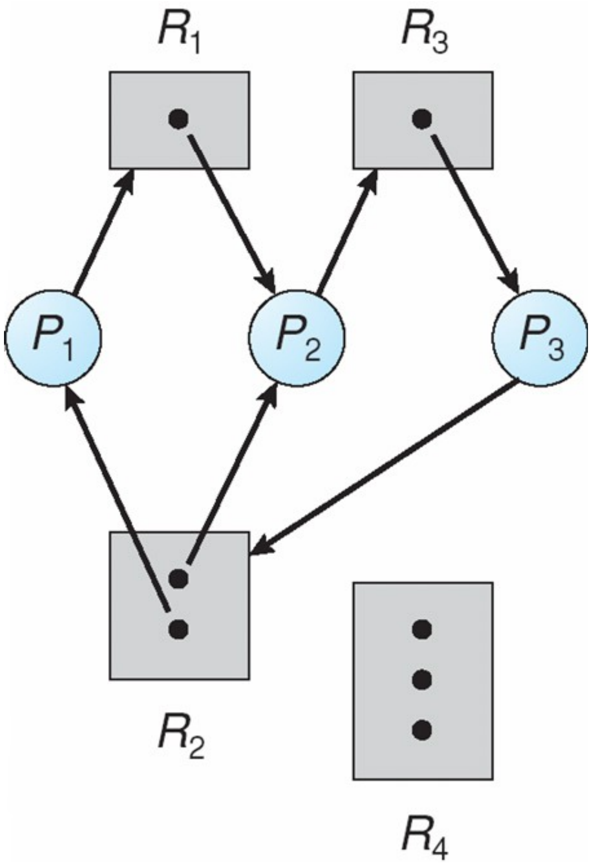
Graphs and Deadlocks

- Theorem:
  - If the graph contains no (directed) cycle, then there is no deadlock

    Note: If the graph contains a cycle, then there may be a deadlock (P=>Q does not mean that Q=>P)
- If there is only one resource instance per resource type, then we have a stronger Theorem:
  - The existence of a cycle is a sufficient and necessary condition for the existence of a deadlock
    - Each process involved in the cycle is deadlocked

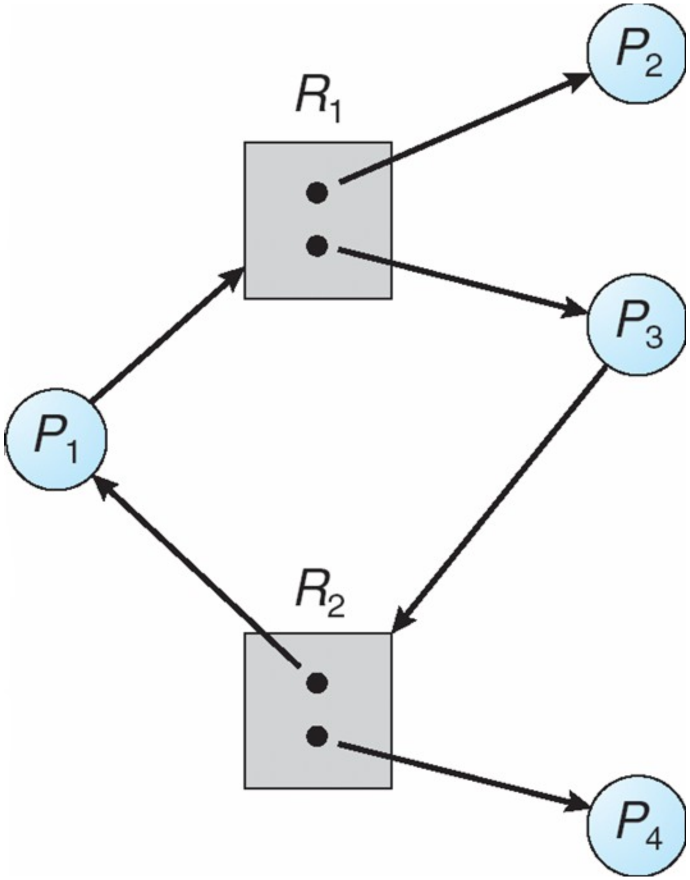
Cycle and Deadlock

- Figure 7.2



Cycle and No Deadlock

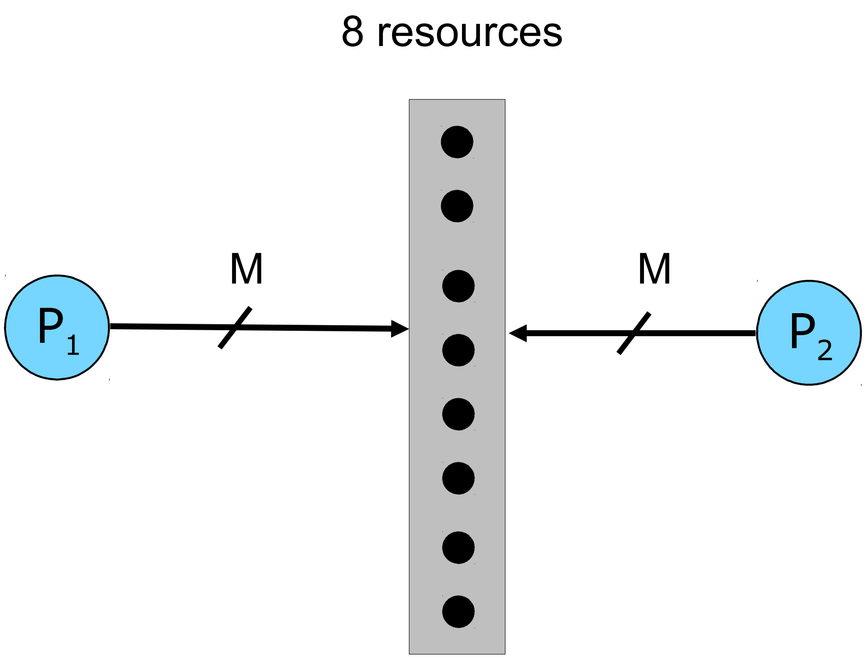
- Figure 7.3



Simple Example

- 8 resources
- 2 threads
- Each thread does:

```
while (true)  {
      for (int i=0; i < M; i++)
            <grab a resource>
      <do some work>
      for (int i=0; i < M; i++)
            <release a resource>
}
```



Question: What is the largest M value to guarantee no deadlocks?

(should we attempt an in-class live simulation?)

Deadlock Handling

- What do we do about deadlocks?
  - We can prevent deadlocks
    - Deadlock prevention
      - Ensure that one of the four conditions never holds
    - Deadlock avoidance
      - Use information about future resource usage of processes
  - We can identify deadlocks and take action
    - Deadlock detection and recovery
      - An algorithm for deadlock detection
      - A recovery strategy
  - We can do nothing and hope
    - That's what Windows, Linux, and the JVM do
    - Eventually the deadlock may snowball until the system no longer functions and requires manual intervention (restart)

Deadlock Prevention

- The four conditions:
  - Mutual Exclusion
  - Hold-and-Wait
  - No preemption
  - Circular Wait
- Getting rid of Mutual Exclusion?
  - In general we cannot design a system in which we don't have some type of mutual exclusion on some types of resources
- Getting rid of No Preemption?
  - This would force resource releases from a waiting process (A) that holds a resource needed by another process (B)
  - A is restarted later and must reacquire all its resources
  - This is easily done for resource that have an easily saved/restored state (e.g., CPU with registers)
  - But cannot be done in general as the processes may be in the middle of doing something that leaves an inconsistent state

Getting Rid of Hold and Wait

- A process cannot request a resource if it holds any other resource
- Option #1: a process could acquire all the resources it needs before it begins execution
  - Problem: low resource utilization
    - A resource is held during the whole process lifetime even if it's used for a tiny fraction of it
- Option #2: a process can request a (bulk of) resource(s) only if it holds no other resources
  - Problem: may not be possible to implement every process as a sequence of "acquire N / release N" steps
- Problem in both options: starvation is possible
  - Some other process may always hold one of the needed resources and acquiring them one after the other is the only way

Getting Rid of Circular Wait

- Preventing cycles:
  - Impose a total ordering on resource types
    - An integer value is assigned to each type
  - A process must request resources by increasing type order
    - or, must release all resources of higher order before requesting a resource of lower order
  - If several instances of the same resources are needed, then a single request for all of them must be issued
- The above will prevent circular wait
  - Simple proof by contradiction in Section 7.4.4
- This works trivially for the two-lock deadlock (A < B)

Thread #1	Thread #2	
lock(A)	lock(B)	
lock(B)	lock(A)	← illegal
unlock(B)	unlock(A)	
unlock(A)	unlock(B)	

Getting Rid of Circular Wait

- It is up to application developers to follow the order
  - Otherwise code will simply say “fail”
- It may not be easy to define the order a-priori
  - If some process may need resource type A before type B, and some other may need resource type B before type A, then you can’t define the order
  - Hard to figure out an order for all system resources
- FreeBSD provides an order-verifier for locks
  - It records lock usage order
  - And then later enforces the recorded order
  - Pretty simple to implement

Deadlock Avoidance

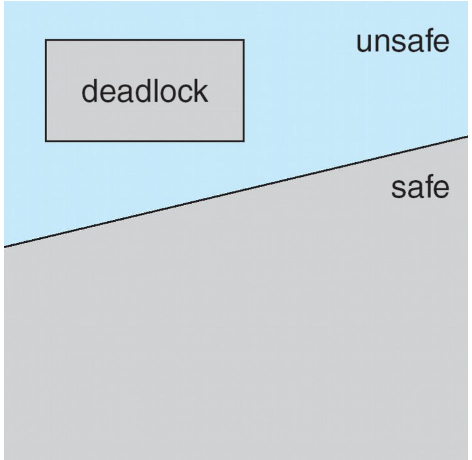
- Idea: if I know what resources a process will need in the future, perhaps I can anticipate deadlocks
- A simple and useful model: each process declares the maximum number of resource of each type that it may need
- Resource state:
  - The number of available resources in each type
  - The number of assigned resources in each type
  - The maximum number of resources of each type for each process
- Goal: ensure that we are always in a safe state

Safe State

- Definition of a safe state: there exists a sequence <P_1, P_2, …, P_n> of ALL the processes is the systems such that
  - for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
  - That is (for $j < i$):
    - If P_i resource needs are not immediately available, then P_i must wait until all P_j have finished
    - When each P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and eventually terminate
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on...
  - Such a sequence is called a safe sequence
- A state without a safe sequence is called unsafe

Safe State

- Theorem:
  - If there is a deadlock, then the state is unsafe
  - If the state is unsafe, then there may be a deadlock



- Goal: never enter an unsafe state, period
  - And conservatively preclude non-deadlocked unsafe states

Example from Section 7.5.1

- 12 resources of the same type, 9 initially assigned
- 3 processes:

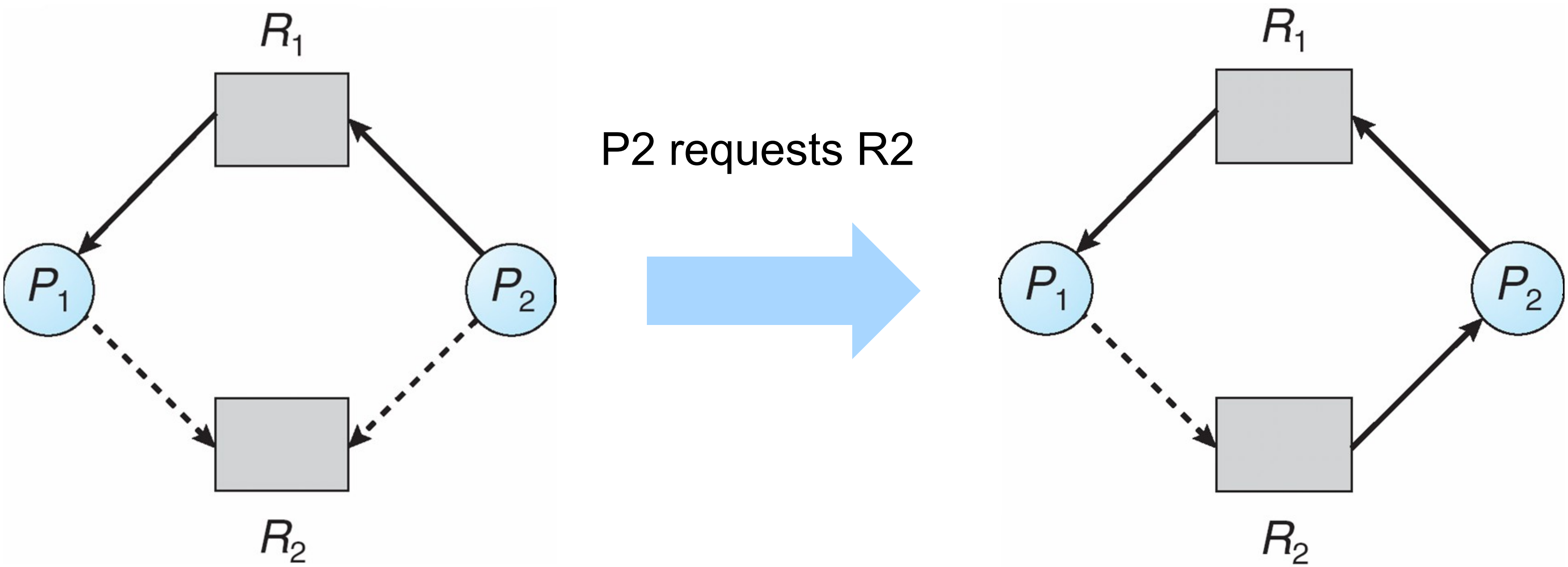
	Maximum Need	Currently Holds
P_0	10	5
P_1	4	2
P_2	9	2

- A safe sequence: <P_1, P_0, P_2>
- If one gives 1 resource to P_2, then we get to an unsafe state
  - P_2 holds 3, P_1 gets and releases 2, then neither P_0 nor P_2 can get everything they need
- Looking at state safety could be done using a brute-force (high-complexity) algorithm

Graph-based Avoidance Alg.

- If each resource type has only one instance, then it is easy to avoid deadlocks
  - A more complex algorithm called the "Banker's algorithm" must be used for multiple instances
- Build a resource allocation graph, but add claim edges
  - edges that correspond to potential future resource needs (all of them)
    - depicted with a dashed line
  - when a resource is assigned, replace the claim edge with an assignment edge
- Grant a resource allocation only if it does not create a cycle in the resource allocation graph
  - The cycle may contain claim edges
  - Detecting a cycle in a graph with n vertices can be done in n^2 time

Graph Example



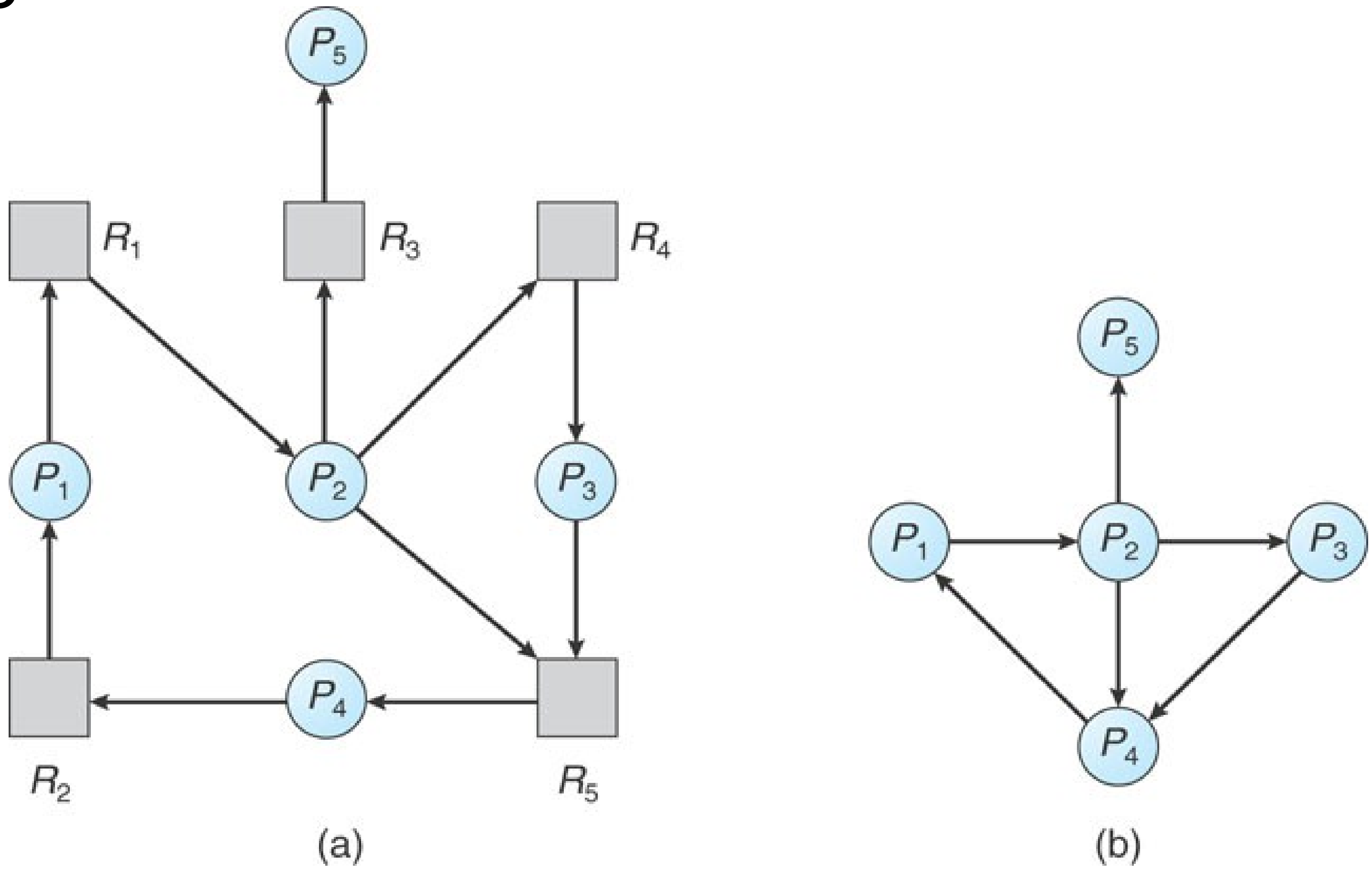
- There is a cycle in the graph
- The request is denied
    - It could lead to a deadlock

Deadlock Detection-Recovery

- Detection-Recovery:
  - Allow system to enter a deadlock state
  - Detect the deadlock state
  - Take some appropriate action to recover
- In the case of one instance per resource type, detection is simple:
  - Build the resource allocation graph
  - Run an $O(n^2)$ cycle-detection algorithm
- Otherwise a more complex algorithm is needed
  - Uses ideas from the Banker's algorithm (see Section 7.5.3 if interested)

Deadlock Detection Example

- Figure 7.9



Deadlock Detection-Recovery

- How often should one run the detection algorithm?
  - Run it often: expensive, but good if deadlocks are frequent
    - extreme: for each resource request, in which case one knows which process “caused” the deadlock
  - Run it rarely: cheap, but bad if deadlocks are frequent
    - and it will be difficult to tell which process “caused” the deadlock

Deadlock Detection-Recovery

- What about recovery?
- Two kinds of actions
  - Process termination
  - Resource preemption
- Process termination
  - Kill all deadlocked processes
    - May be wasteful
  - Kill one process at a time until the deadlock disappears
    - High overhead because deadlock detection algorithm is run at each step (but the system was frozen anyway)
  - Killing a process could be tricky
    - The process may be in the middle of something that would leave an inconsistent state, that must be fixed

Deadlock Detection-Recovery

- Resource Preemption
  - Selecting a victim: which resource/process needs to be preempted
  - Rollback: when preempting a resource from a process, that process must be rolled back
    - Simple solution: restart the process from scratch
    - May require inconsistent state cleanup
  - Starvation: ensure that one process doesn’t see its resource preempted from it forever

Conclusion

- Three methods
  - (i) Deadlock prevention/avoidance
  - (ii) Deadlock detection-recovery
  - (iii) Do nothing and let users deal with it
- The solutions we have discussed for (i) and (ii) above are interesting
- Most argue that none of them covers all the bases
- One could combine them all and be effective, but at the cost of much increased Kernel complexity
- Therefore, in practice, it’s option (iii)