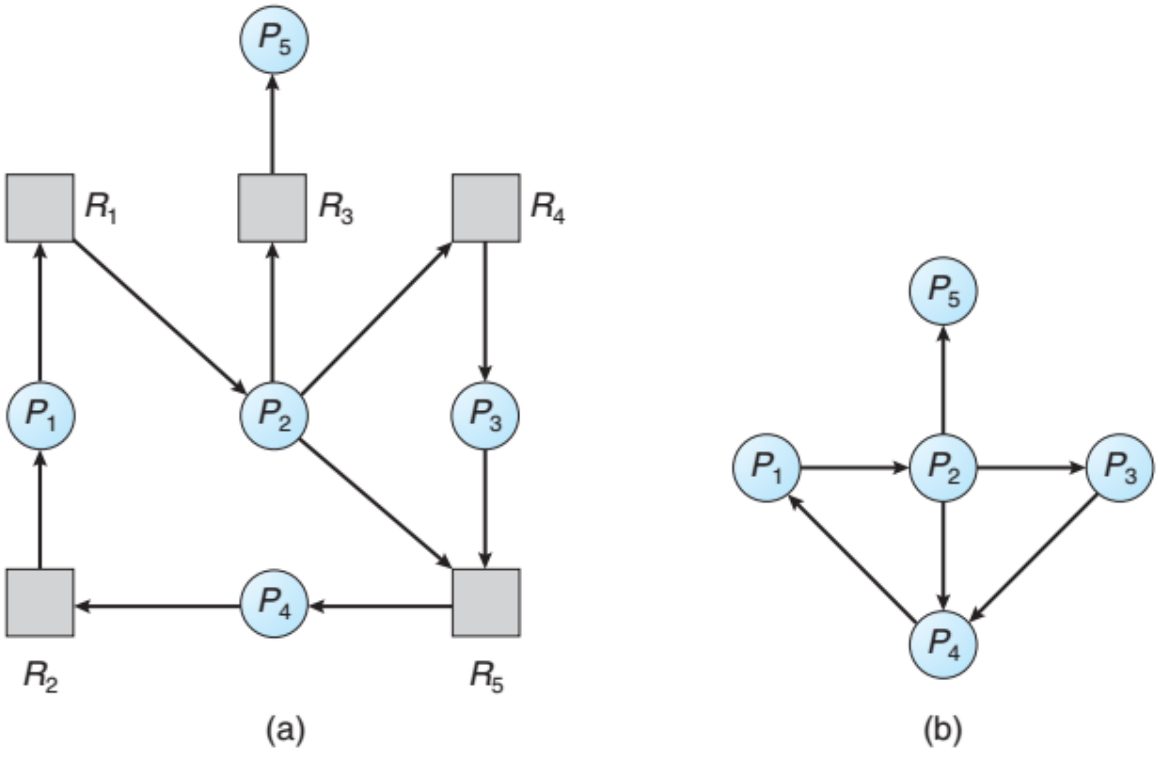
Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred
  - An algorithm to recover from the deadlock.

- we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

1 Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq.
- For example we present a resource-allocation graph and the corresponding wait-for graph.



(a) Resource-allocation graph. (b) Corresponding wait-for graph.

- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.

2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

Recovery from Deadlock

- ✓ When a detection algorithm determines that a deadlock exists, several alternatives are available.
- ✓ One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- ✓ Another possibility is to let the system recover from the deadlock automatically.
- ✓ There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes**. This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

- Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated.

- This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one.

- Many factors may affect which process is chosen to terminate, including:
  1- What the priority of the process is
  2- How long the process has computed and how much longer the process will compute before completing its designated task
  3- How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
  4- How many more resources the process needs in order to complete
  5- How many processes will need to be terminated

2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

2. Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.