Deadlock analysis

Based upon ch.6 in Pankaj's book

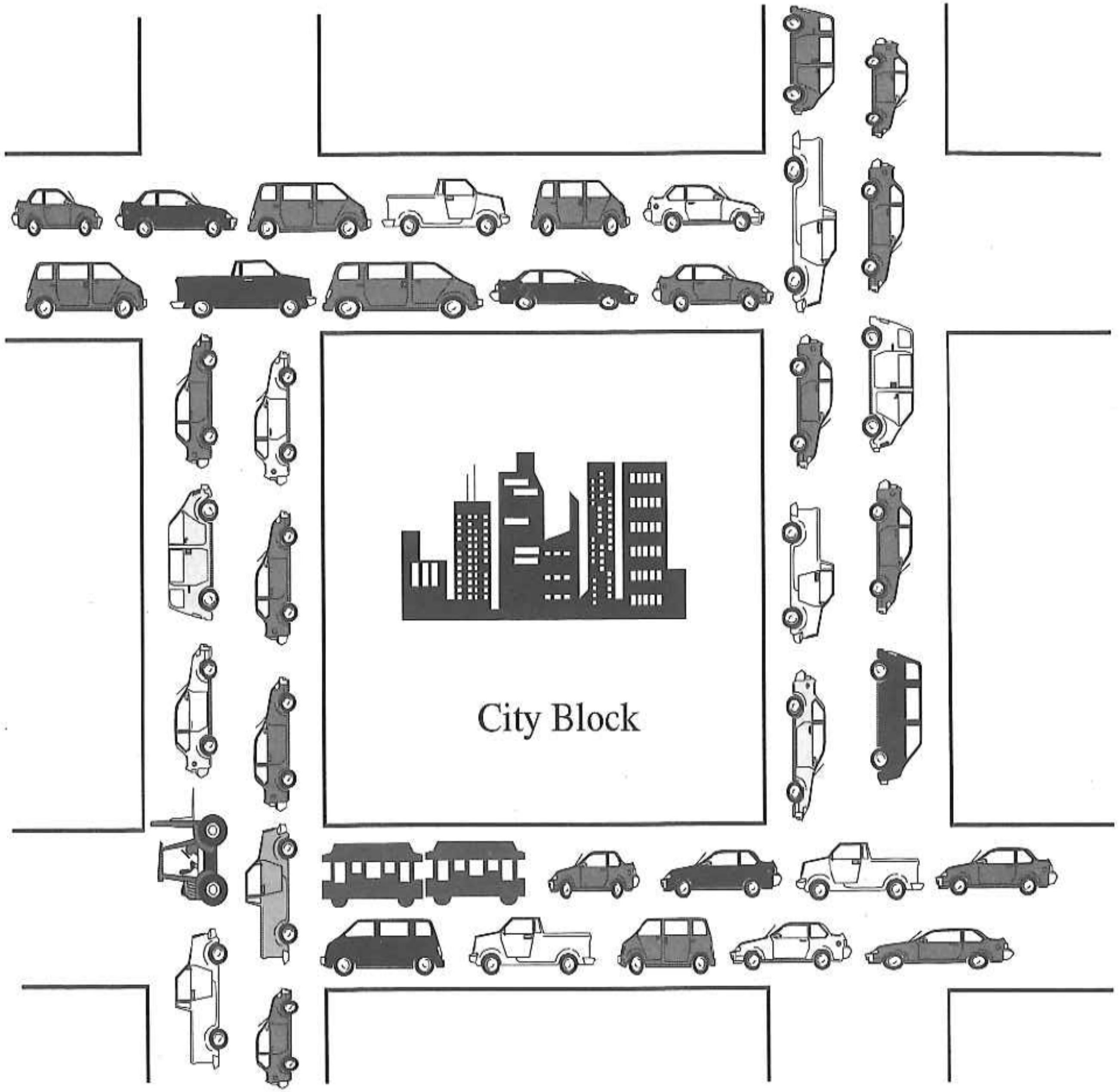
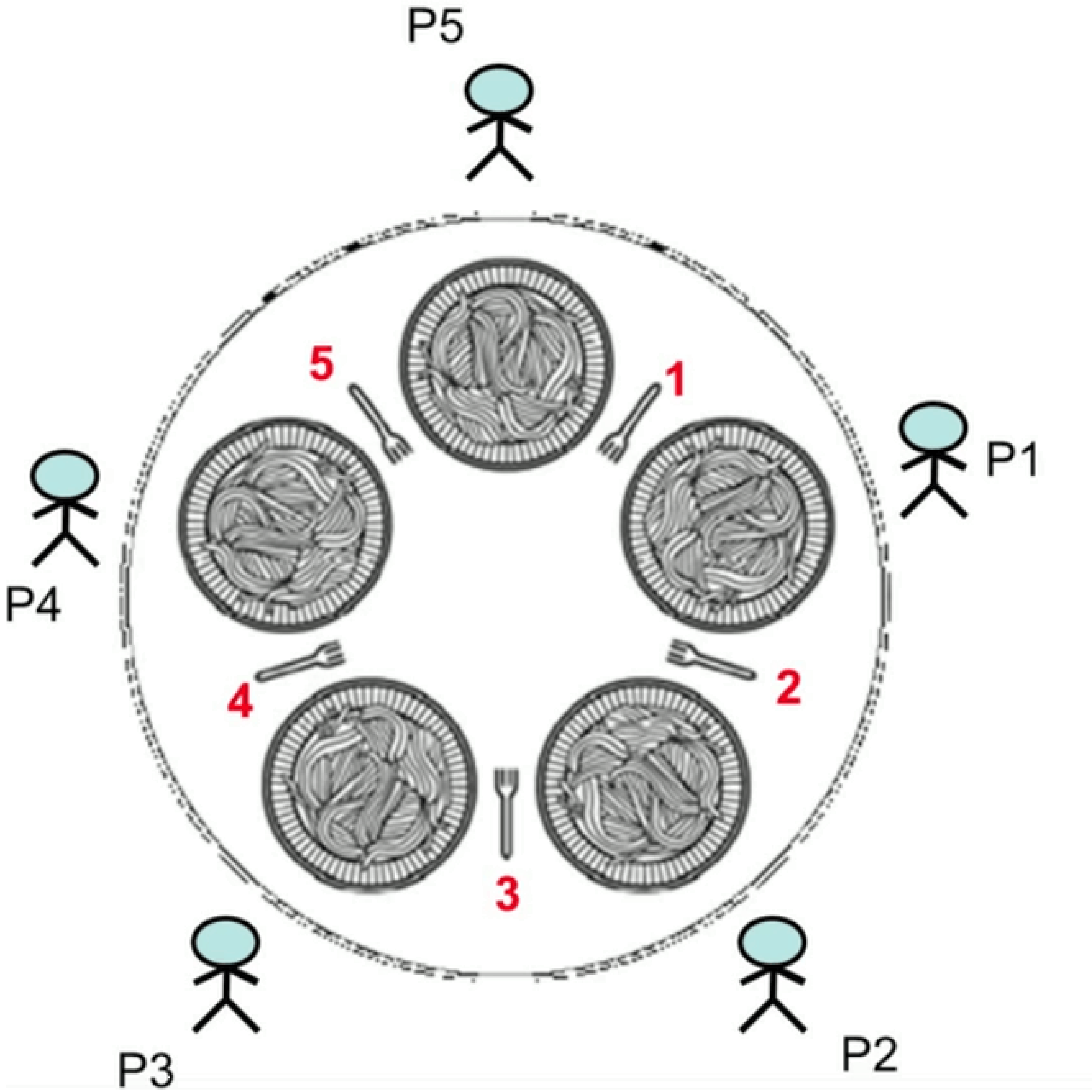


Figure 6–1. Traffic Gridlock.

Deadlock in the classic dining philosophers problem

- Philospher life cycle
  - Take right fork
  - Take left fork
  - Eat
  - Put left fork
  - Put right for
  - Think
  - Start all over again
- Deadlock when each philopher has his/her right fork only



6.3 Conditions for Deadlock

The early work of Coffman et al. [1971] showed that in order for deadlock to occur, four conditions must exist:

1. **Mutual Exclusion**: A thread can seize exclusive control of an object, and no other thread can have access to it.
2. **Hold and Wait**: A thread can hold locked resources while waiting to acquire more.
3. **No Preemption**: Once a certain resource is held by a thread, it cannot be involuntarily reassigned to another thread.
4. **Circular Wait**: Two or more threads can hold some resources and await some held by other threads. If this dependency is circular, the threads will all be waiting for others to release needed resources, and none will make progress.

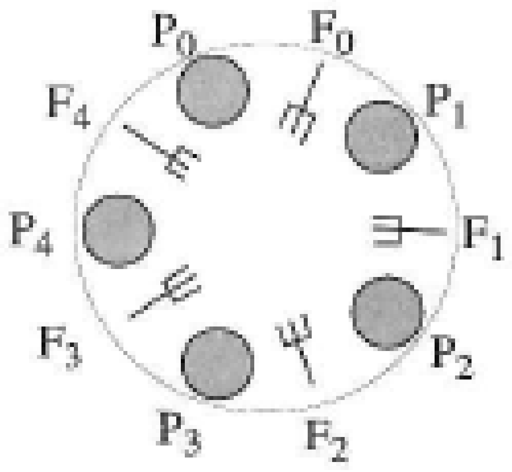


Figure 6–3. The Dining Philosophers' Table.

6.5 Handling Deadlocks

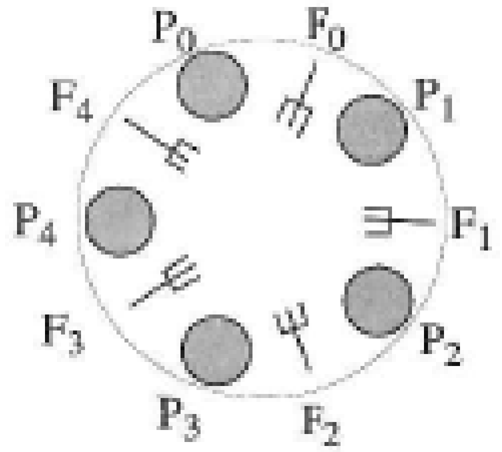
In general, there are several ways to deal with deadlocks:

- **Ignoration**: Handling deadlock is costly, and sometimes it is not practical to spend much computing resource to handle it, particularly in systems and situations where deadlock possibility is small. Since deadlock is a benign and easily detectable problem, in some cases one may choose not to handle deadlock in a program, and then manually break the cycle (usually by terminating one or several deadlocked threads or processes) when a deadlock is detected.
- **Detection**: There are algorithms to search for cycles of deadlocked threads and processes. When such cycles are detected, we can sometimes preempt some threads to release their resources, roll back their execution state, or simply terminate them.
- **Prevention**: Through a set of rigid rules and restrictive resource allocation requirements, we can prevent deadlock from occurring by precluding any one of the four conditions for deadlock at every point in a program.
- **Avoidance**: We can implement algorithms and services to carefully manage resource allocation in order to avoid getting into a potential deadlock situation. For efficiency, this solution does not rule out all possibility of deadlock like the *deadlock prevention* strategy; it simply monitors the state of the system's resource allocation and circumvents deadlock when imminent by being very conservative about resource allocation. Djikstra's *banker's algorithm* is a classic example of this approach.

6.6 Deadlock prevention – attack one of the 4 conditions

- **Prevent hold-and-wait:** Each thread must request all resources at once, and it cannot proceed until all resources are granted.
- **Allow preemption:** If a thread holding a resource is denied further resource requests, it must release all resources that it holds and request them later with the additional resource.
- **Prevent circular wait:** Impose a linear ordering on all resources R_j such that the order of resource request and allocation follows this ordering. For example, if a task has been allocated resources of type R_j, it can subsequently request only resource R_{j+1} or, later in the ordering, R_1 to R_n.

NO!: Actually attacking the hold-and-wait condition with 2-phase locking



- Condition 1 mutual exclusion can also be attacked by making a resource asynchroneous – as an example: printer spoolers

6.7 deadlock detection and recovery: detection

- Maintain a ressource graph on run-time:
  - T1 incoming arrow means T1 has R1
  - T1 outgoing arrow means T1 requests R2
- Detection of deadlock is finding cycles – running time $O(N^2)$

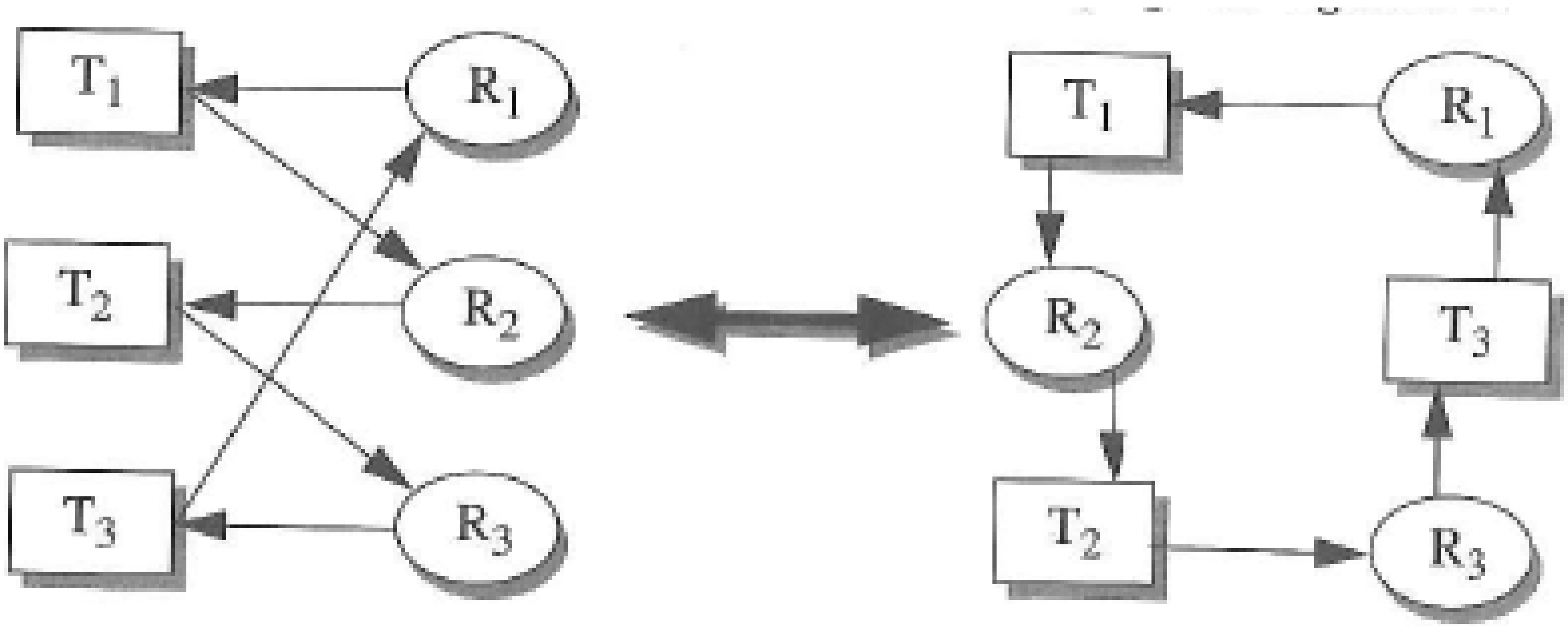
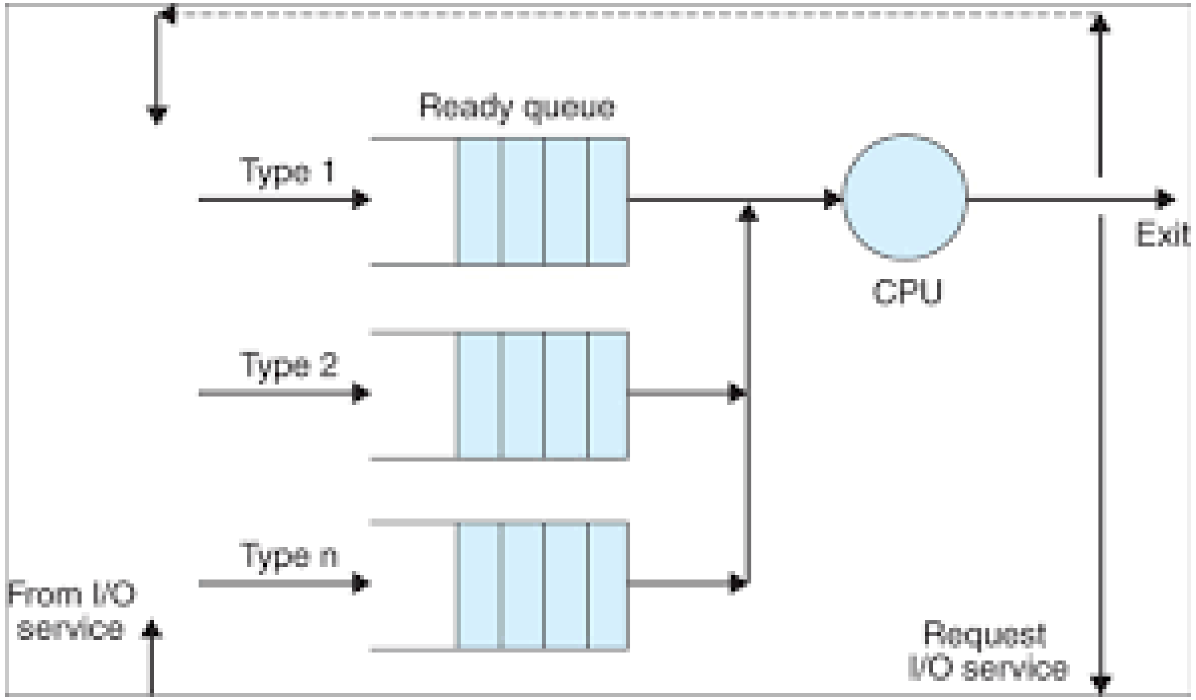


Figure 6–5. Two Views of the Same *Resource Allocation Graph.*

Or an even simpler - detection in a multi-level priority scheduler:

- When a deadlock occur then the watch-dog (idle-task) on lowest priority level wakes up.
- The watch-dog understands that a deadlock has occurred and takes action (see next slide)



6.7.3 deadlock recovery – 3 ways

- Termination
  - Kill tasks then other tasks can get their ressources
  - Easy, brutal
- Ressource preemption
  - Difficult to deal with for tasks
- Rollback from log
  - The log has major and perhaps also minor synchronization points
  - Safe but costly approach

6.8 deadlock avoidance

- Safe and unsafe states
  - Avoid unsafe states

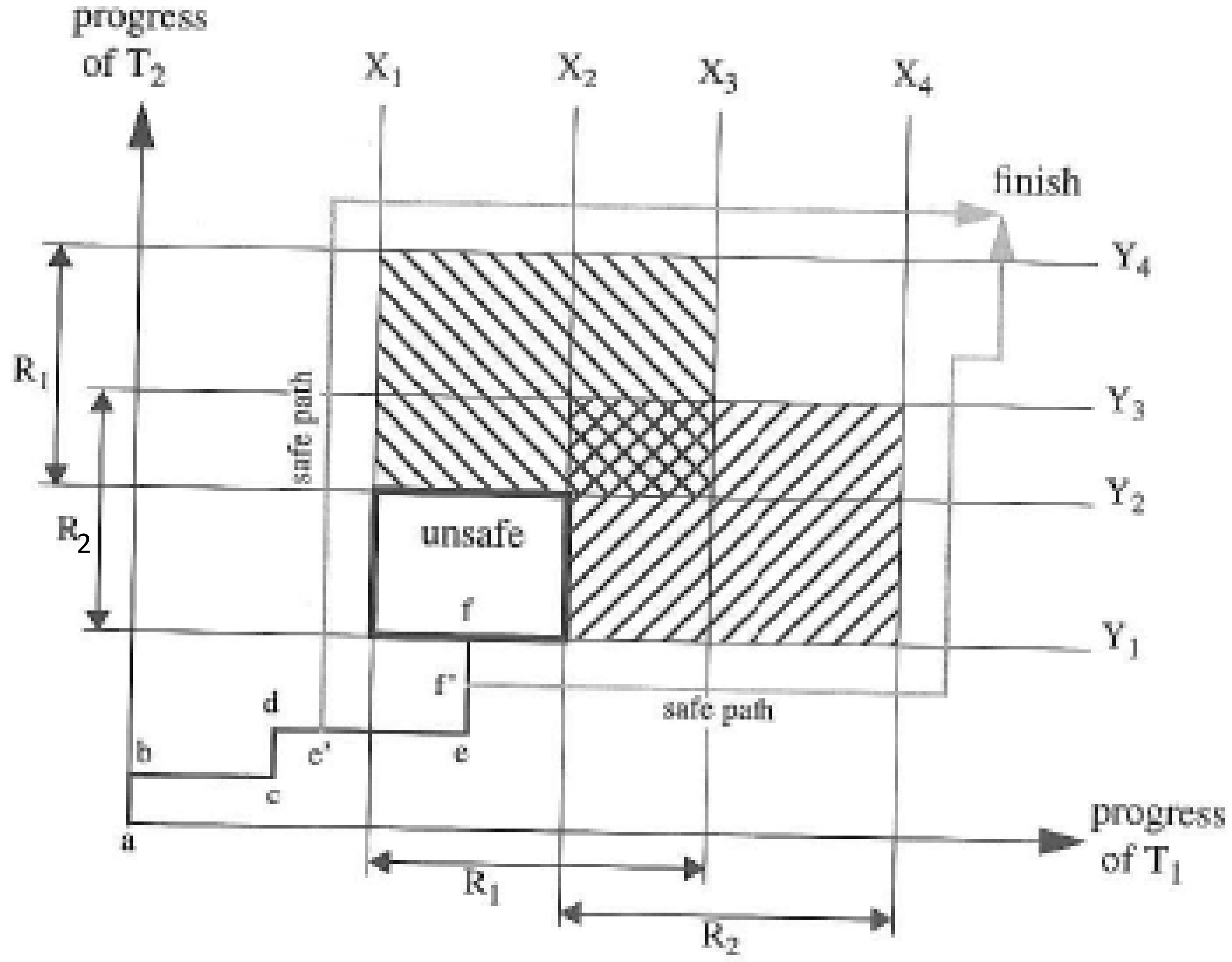


Figure 6–9. Safe and Unsafe States of System Progress.

6.8 deadlock avoidance

- Bankers algorithm : introduce a clever ressource manager

	Bob	Joe	Eve	Bank
Loan so far	1000000	1900000	100000	
needs	200000	300000	2400000	
House price	1200000	2200000	2500000	250000
To whome? BoB				
To whome? Joe				
To whome? Eve				