CSE 660: Introduction to Operating Systems

Deadlock

Presentation F

Deadlock Problem

- Here we consider a deadlock involving hardware resources.
- A set of processes is deadlocked if each process in the set is blocked and waiting for a resource that only another process in the set can release.
- System model:
  - – Resource types R_1, R_2, . . ., R_m, *e.g. CPU cycles, memory space, I/O devices, …*
  - – Each resource type R_i has W_i instances.
  - – Each process utilizes a resource as follows:
    - **– request for a resource is made; if the request is not granted, a process is blocked (and waits) until a resource is granted.**
    - **– usage of a resource for some time; after resource being granted.**
    - **– release of resource; when not any more needed.**

Deadlock: Example

- In a given system, we have one printer and one plotter, and two processes A and B.

Process A	Process B
(1) -	(8) -
(2) Request (printer)	-
-	(9) Request (plotter)
(3) -	(10) -
(4) Request (plotter)	(11) Request (printer)
-	-
(5) Release (printer)	(12) -
(6) Release (plotter)	-
(7) -	(13) Release (printer)
-	(14) Release (plotter)
-	(15) -

Scenario 1: (1)-(7), (8)-(15) ==> No problem (no deadlock)

Scenario 2:

(1)-(3), interrupt, (8)-(11), B blocked, (4), A blocked ==> Deadlock

Deadlock Characterization

- Deadlock may arise if the following four conditions hold simultaneously:
  - **Mutual exclusion**: only one process at a time can use a resource.
  - **Hold and wait**: a process holding at least one resource is blocked and waiting to acquire additional resources held by other processes.
  - **No preemption**: a resource can be released only voluntarily by the process holding it, i.e. the process completes resource related tasks before releasing the resource.
  - **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

- Note above conditions are necessary but not sufficient for deadlock to occur.

Resource-Allocation Graph

- Resource allocation graphs are useful in analyzing deadlock situations.
- A graph includes a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two types:

  $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

  $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system; each resource type R_i has W_i instances.
- The set of edges E includes two types of edges :
  - request edge – directed edge $P_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

If graph contains no cycles ⇒ no deadlock.

If graph contains a cycle ⇒

- if only one instance per resource type, then deadlock.
- if several instances per resource type, possibility of deadlock.

Resource-Allocation Graph: Example

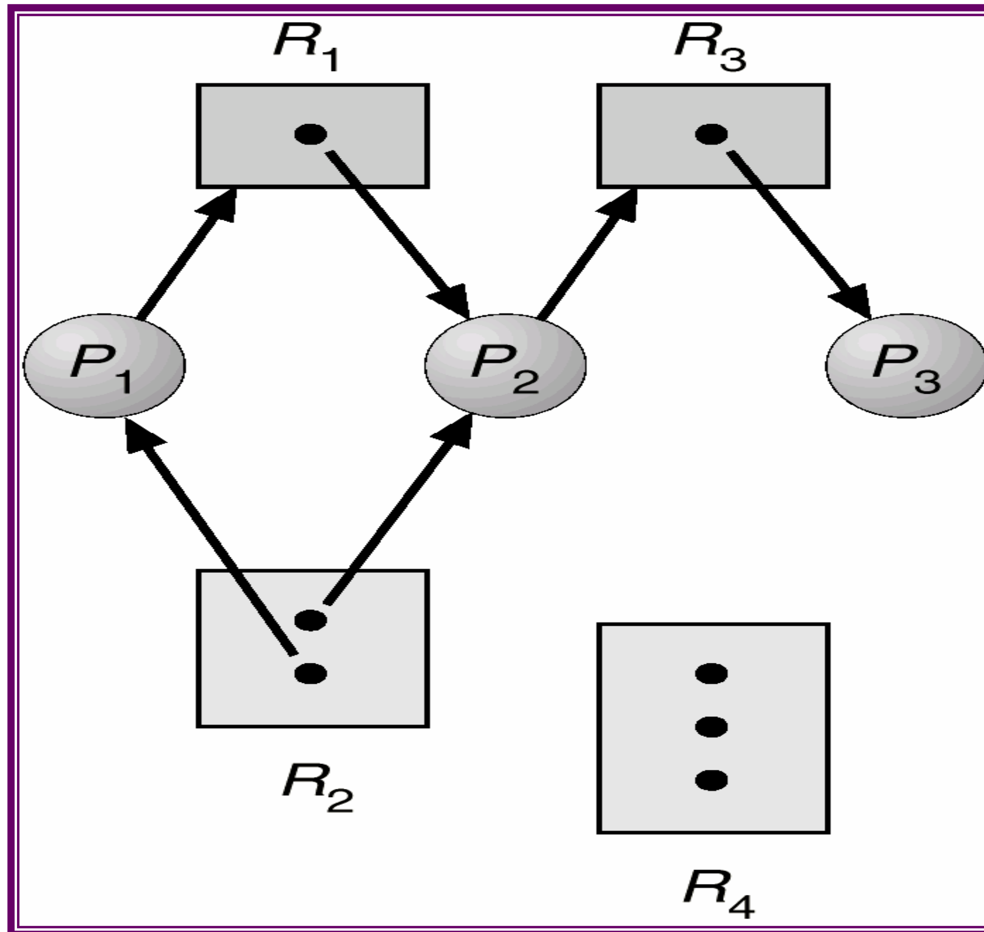


Figure 7.2

No cycle → no deadlock

Resource-Allocation Graph With Cycle (1)

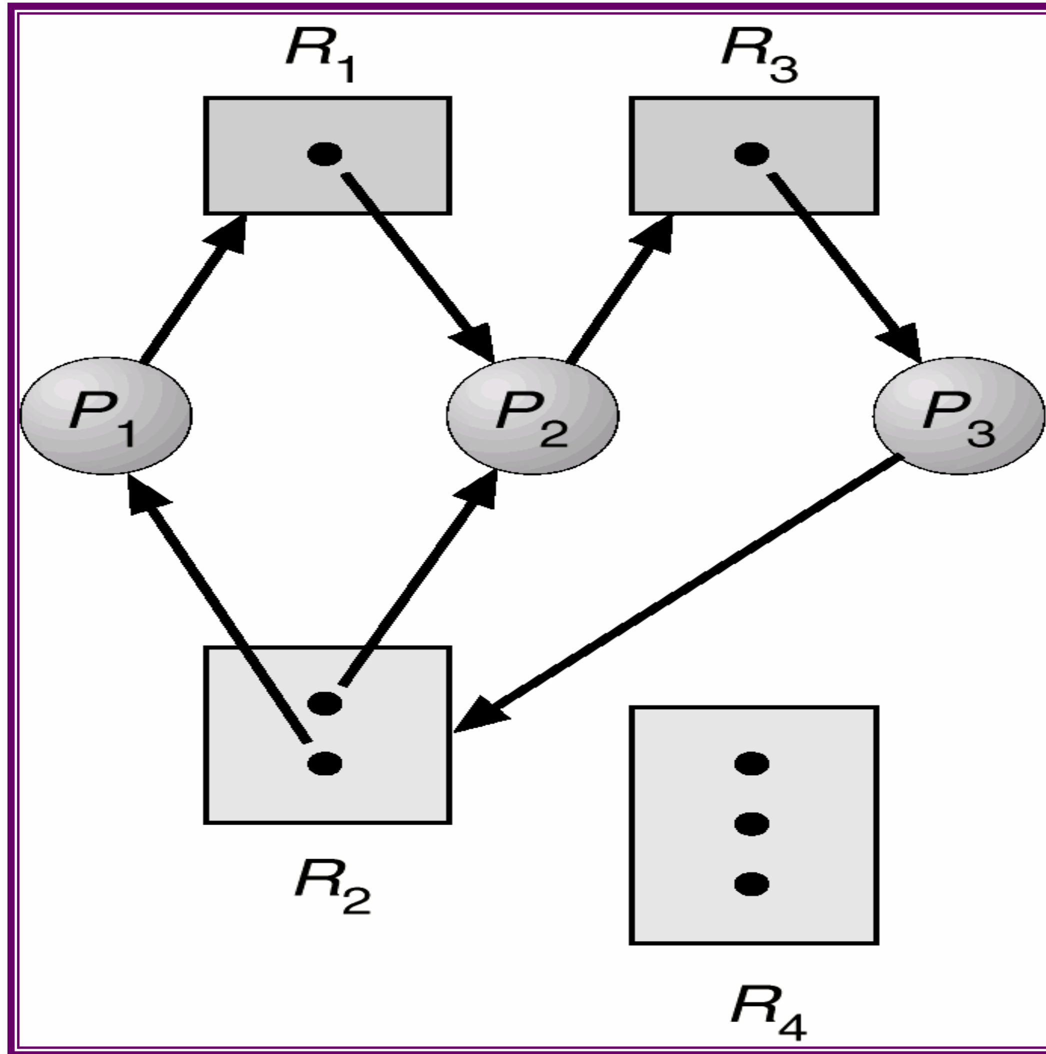


Figure 7.3

Deadlock exists

Resource-Allocation Graph With Cycle (2)

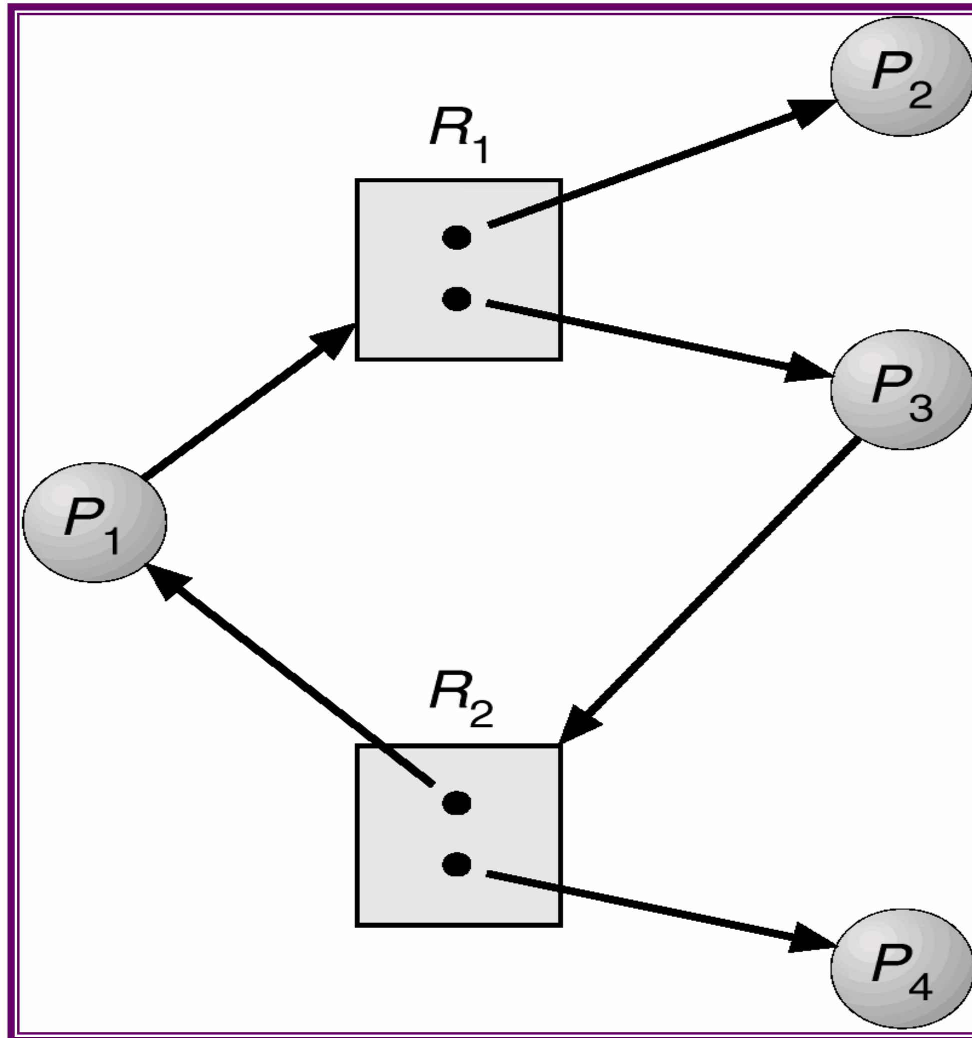


Figure 7.4

No deadlock

Methods for Handling Deadlocks

- Deadlock prevention: Ensure that the system will *never* enter a deadlock state by making that at least one of the necessary deadlock conditions does not hold.
- Deadlock avoidance: Ensure that the system will *never* enter a deadlock state since O.S. never granting an unsafe request for resources; O.S. has to have some advanced information how resources are to be requested.
- Deadlock detection and recovery: Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems; (including UNIX?)

Deadlock Prevention

- Mutual Exclusion – not required for sharable resources; must hold for non-sharable resources.

  Solutions for some types of non-sharable resource:

  – spool if you can, e.g. printer or plotter,

  – windows introduced to share one monitor.

  Unix does all of those, thus it takes some care of deadlock.

- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Requires a process to request and be allocated all its resources before it begins execution, or
  - Allow a process to request resources only when the process has none.

  Because of low resource utilization or/and starvation, none of those approaches promising.

Deadlock Prevention (Cont.)

- No Preemption – usually not promising, although applicable to resources whose state can be easily saved and restored later, e.g. CPU registers and memory space.

- Circular Wait – an interesting and useful algorithm exists, that can be also applied in some different situations.
  - The algorithm imposes a total ordering of all resource types, i.e. each resource type has its unique number;
  - Processes can require resources any time but any request may be asking only for a resource numbered higher than any of currently held resources.
  - If this this rule is followed by everybody, a circular wait is not possible, thus a deadlock is prevented. Note that the operating system can detect a process that doesn't follow the rule.

Deadlock Avoidance

- Deadlock avoidance algorithms normally require that the system has some additional *a priori* information about process behavior. The simplest and most useful model requires that each process declares in advance the *maximum number* of resources of each type that it may need during its execution.
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state or moves it into unsafe state*.
  - System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is a safe sequence, if for each P_i the resources that P_i may request can be satisfied by currently available resources plus resources held by all the P_j, with $j<i$.

Deadlock Avoidance (Cont.)

- If P_i resource needs are not immediately available, then P_i can wait until all P_j $(j<i)$ have finished and terminated.
- When all P_j are finished, P_i can obtain needed resources, executes, returns allocated resources, and terminates.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Deadlock avoidance ⇒ ensure that a system never enters an unsafe state.

Banker's algorithm

- Definitions for a system with *n* processors and *m* resources:

$$\text{Max - Avail matrix } A = \begin{pmatrix} a_1 & a_2 & \dots & a_m \end{pmatrix}$$

$$\text{Max - Claim matrix } B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix}$$

$$\text{Current Allocation matrix } C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nm} \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{pmatrix}$$

$$\text{Current Avail matrix } D = \begin{pmatrix} d_1 & d_2 & \dots & d_m \end{pmatrix} = A - \sum_{k=1}^{n} C_k$$

$$\text{Current Need matrix } E = \begin{pmatrix} e_{11} & e_{12} & \dots & e_{1m} \\ e_{21} & e_{22} & \dots & e_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ e_{n1} & e_{n2} & \dots & e_{nm} \end{pmatrix} = B - C = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

$$\text{Request vector } F_i = \begin{pmatrix} f_{i1} & f_{i2} & \dots & f_{im} \end{pmatrix}$$

Algorithm steps

- Step 1 - tentative accept of request

  $D := D - F$ // update Current Avail matrix D

  $C_i := C_i + F_i$ // update Current Alloc vector C_i

  $E_i := E_i - F_i$ // update Current Need matrix E_i

- Step 2 - safe-state checking test

  see next slide in which

  - ✓ freemoney = D
  - ✓ loan[i] = C_i
  - ✓ need[i] = E_i

- Step 3 - if test positive definitive accept of request, otherwise roll back the updates of step 1

Banker at work: safe-state checking

```
While  (last_iteration_successful)
   last_iteration_successful = false ;
   for i = 1 to N do
      if (finishdoubtful[i] AND need[i] ≤ freemoney) then  // need = claim - loan, how
                                                           // much process i still needs

           finishdoubtful[i] =  false ;                    // process can finish because
           last_iteration_successful  =  true              // need ≤ free resources

           free money = free money + loan[i] ;             // process is able to finish thus
                                                           // it will repay the loan back

      end if
   end for
End while

if (free money == capital) then safe!
        else not safe! ;
```

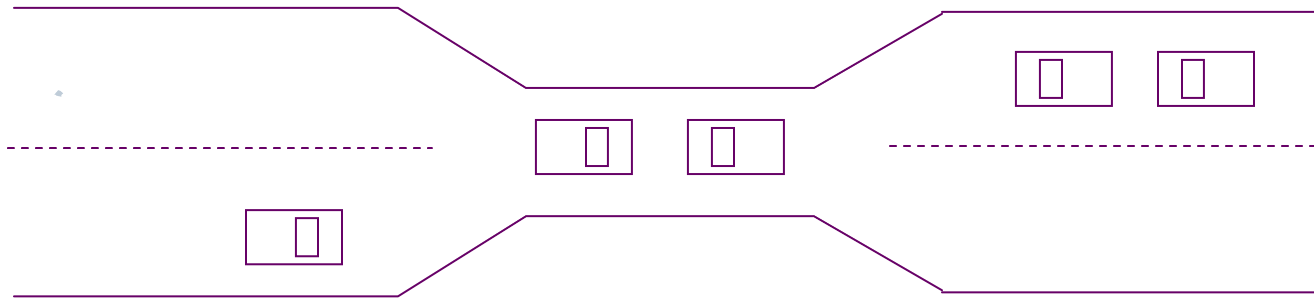
Recovery From Deadlock

- Process Termination:
  a. Abort all deadlocked processes.
  b. Abort one process at a time until the deadlock cycle is eliminated. In which order should we choose to abort?
    - **Priority of the process.**
    - **How long process has computed, and how much longer to completion.**
    - **Resources the process has used.**
    - **Resources process needs to complete.**
    - **How many processes will need to be terminated.**
- Rollback – return to some safe state, and restart processes for that state.

Deadlock Detection and Recovery

- Allow a system to enter deadlock state.
- Deadlock detection algorithm needed to detect deadlock, and then some recovery scheme has to apply.
- When, and how often, to invoke a deadlock detection algorithm?
- If a deadlock detection algorithm is not invoked on time, there may be many cycles in the resource-allocation graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Bridge Crossing Example



- Bridge traffic only in one direction at a time.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible, meaning that some cars wait for a long time.

Deadlock Problem

- In an electronic fund transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying the account to be credited, the account to be debited and an amount of money. The process first locks both accounts and transfers the money, releasing the locks when done.
- When a process attempts to lock an account already locked by another process, it will be blocked and it will stay blocked until the account is released.
- With many processes running in parallel, there is a very real danger that having locked the account x, a process will be unable to lock the account y (thus it will be blocked) because y has been locked by another process now waiting (and it is blocked) for x. Deadlock!!!
- Devise a schema that is deadlock free.

Traffic Deadlock

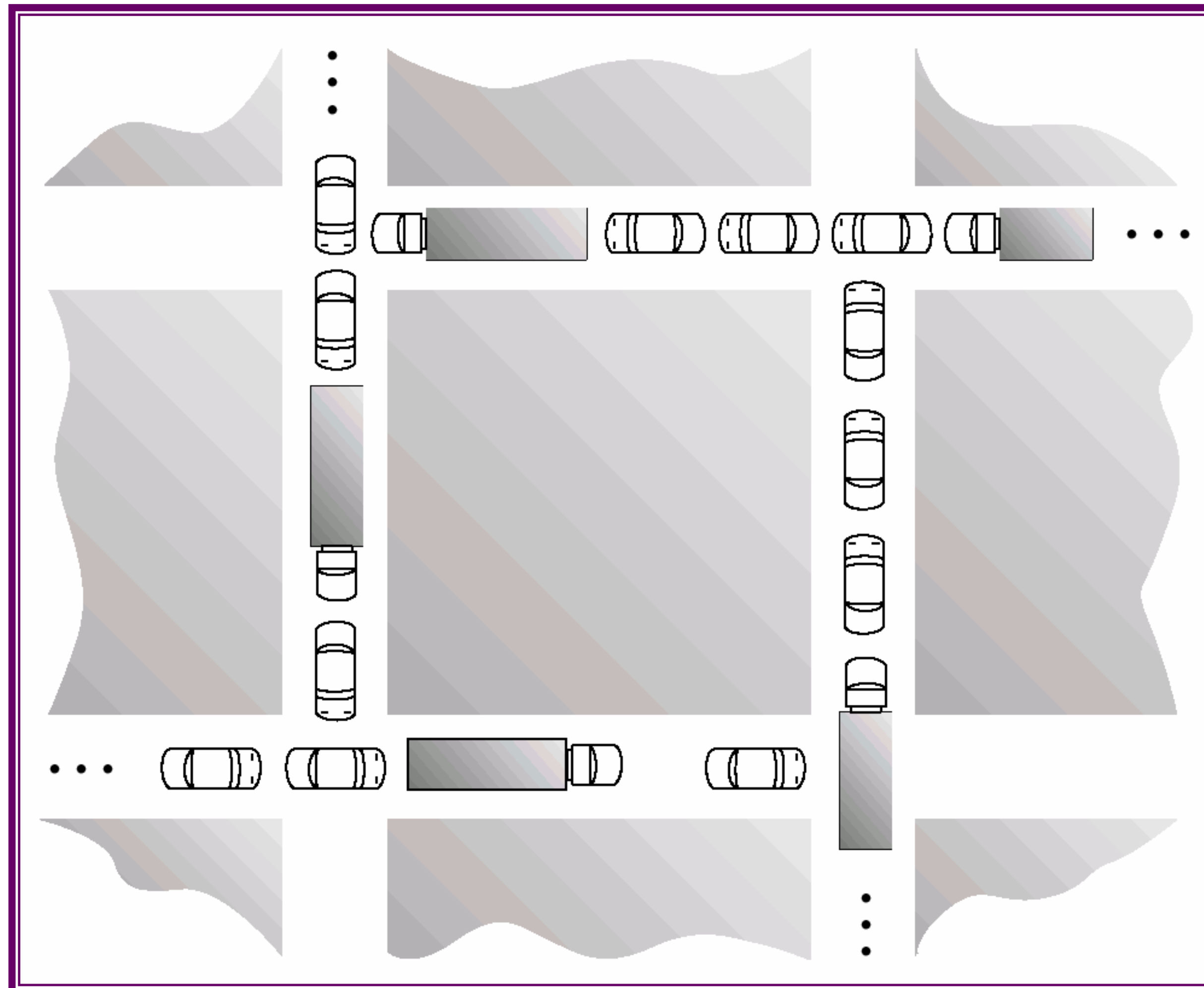


Figure 7.9