Deadlock and Starvation

DRS. KROPP, YOUSSFI, AND STALLINGS

Objectives

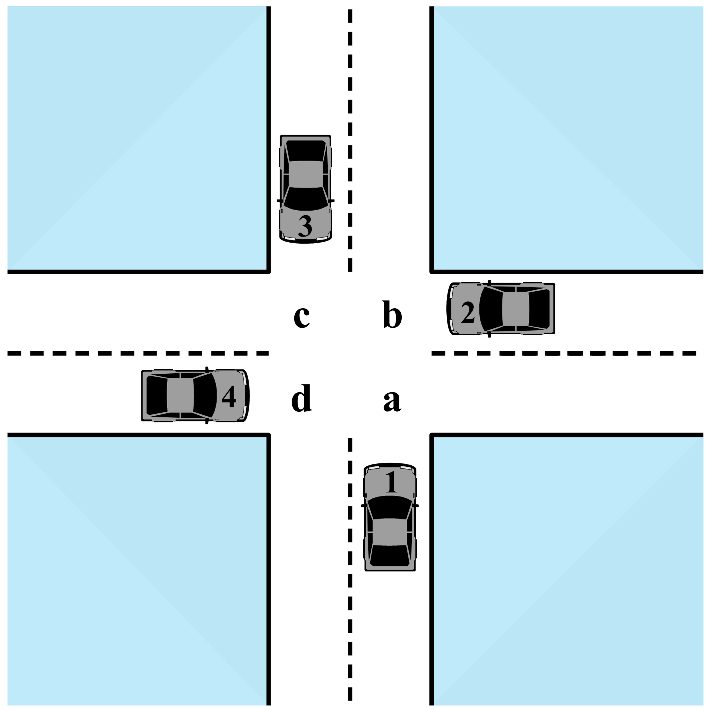
- Define and understand the condition of deadlock
- Deadlock prevention
- Deadlock avoidance (and its difference to deadlock prevention)
- Deadlock detection
- Integrated approach deadlock strategy
- Analyze the dining philosophers' problem

Deadlock definition

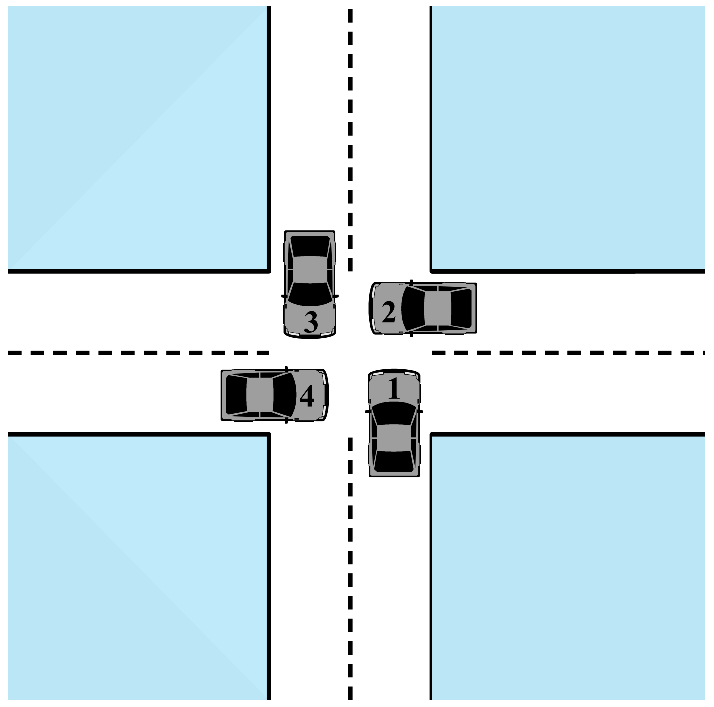
Deadlock Definition

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution

Traffic Deadlock Example



(a) Deadlock possible



(b) Deadlock

Four cars arriving at a four-way stop intersection at the same time:

car 1 needs quadrant a & b

car 2 needs quadrant b & c

car 3 needs quadrant c & d

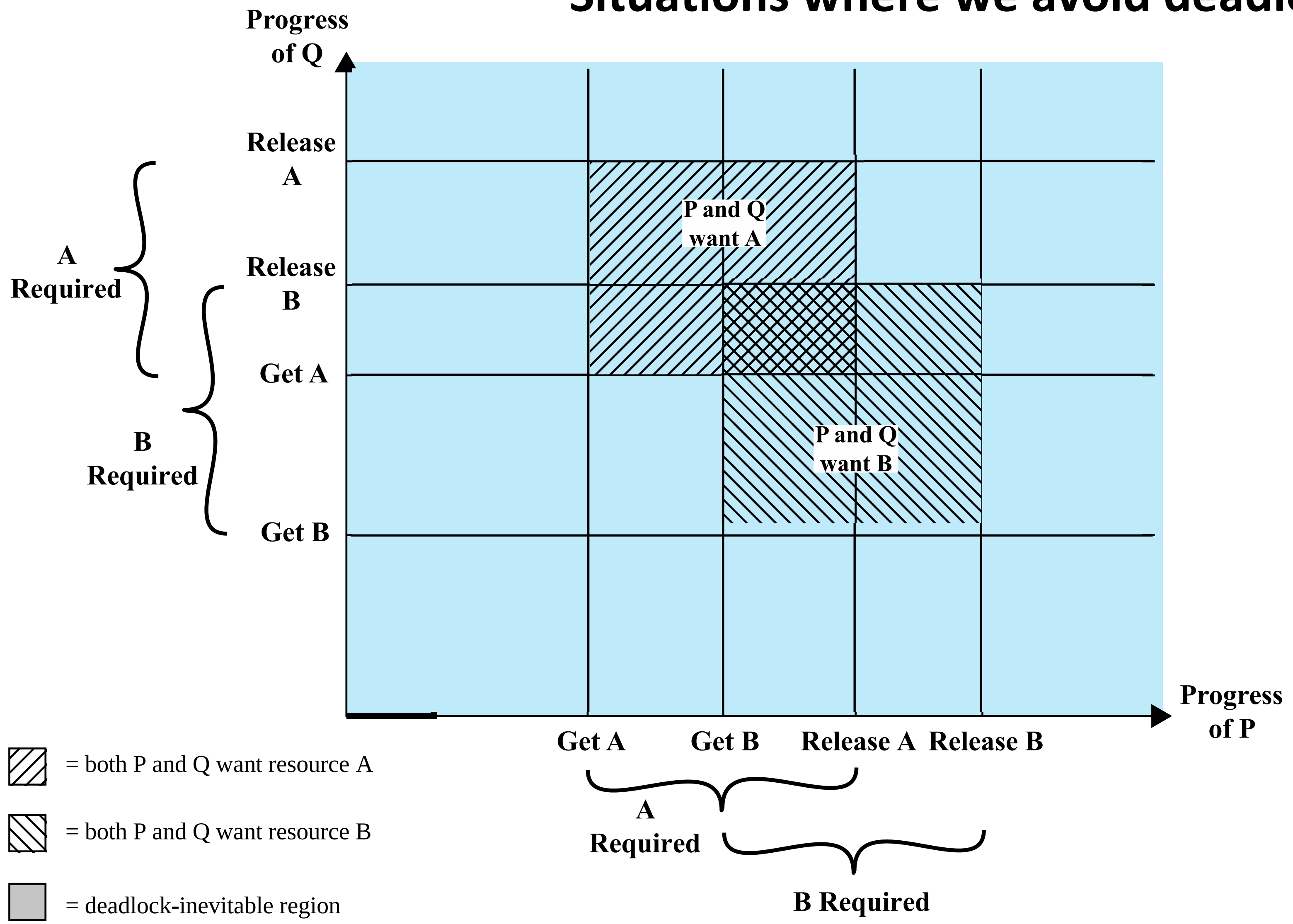
car 4 needs quadrant d & a

Consider the following scenario

- We're running on a uniprocessor system
- Two processes are running, and require two resources: A and B

Process P	**Process Q**
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

Situations where we avoid deadlock



▨ = both P and Q want resource A

▧ = both P and Q want resource B

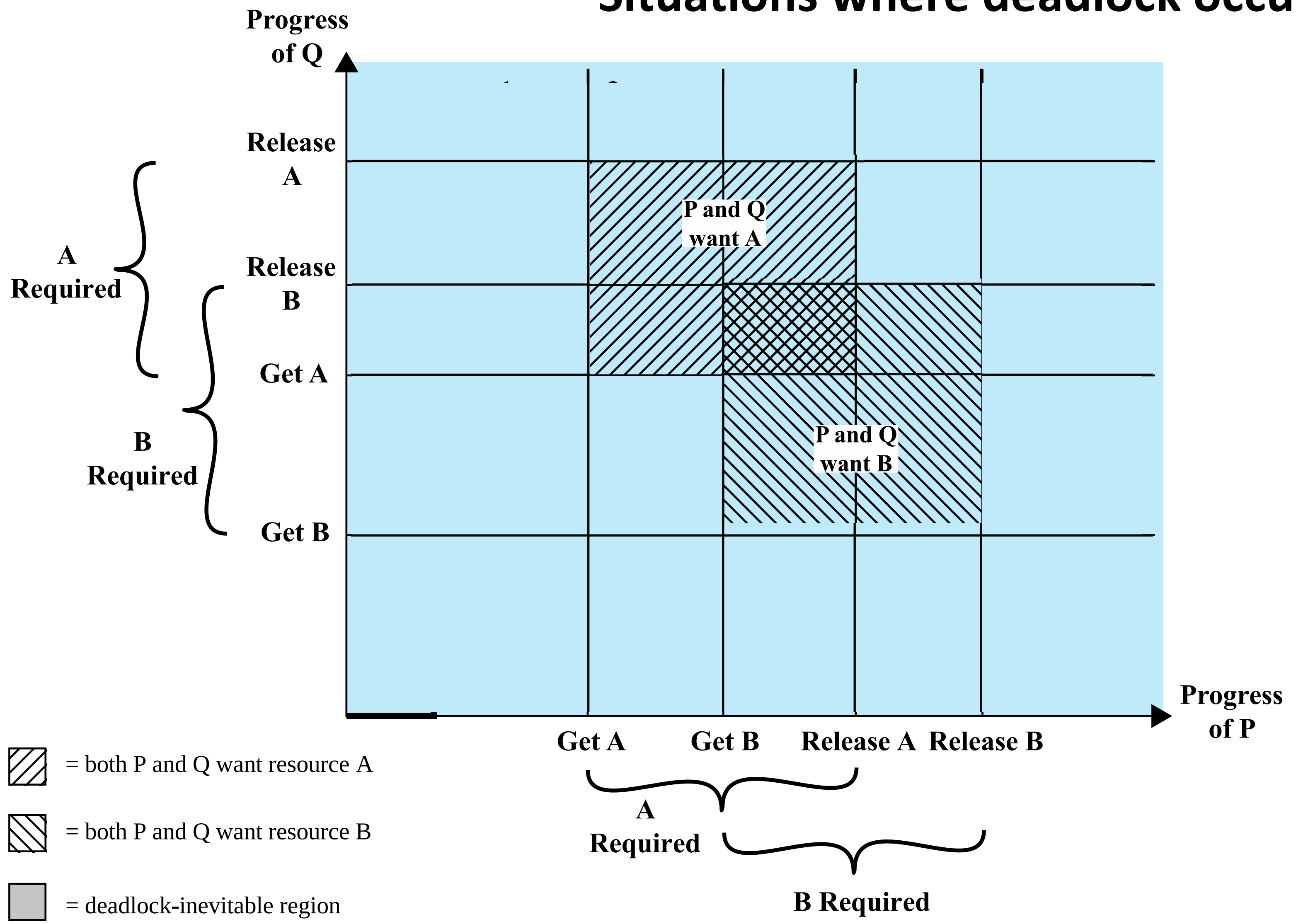
⬜ = deadlock-inevitable region

→ = possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Process P	**Process Q**
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

Joint process diagram

Situations where deadlock occurs



▨ = both P and Q want resource A

▧ = both P and Q want resource B

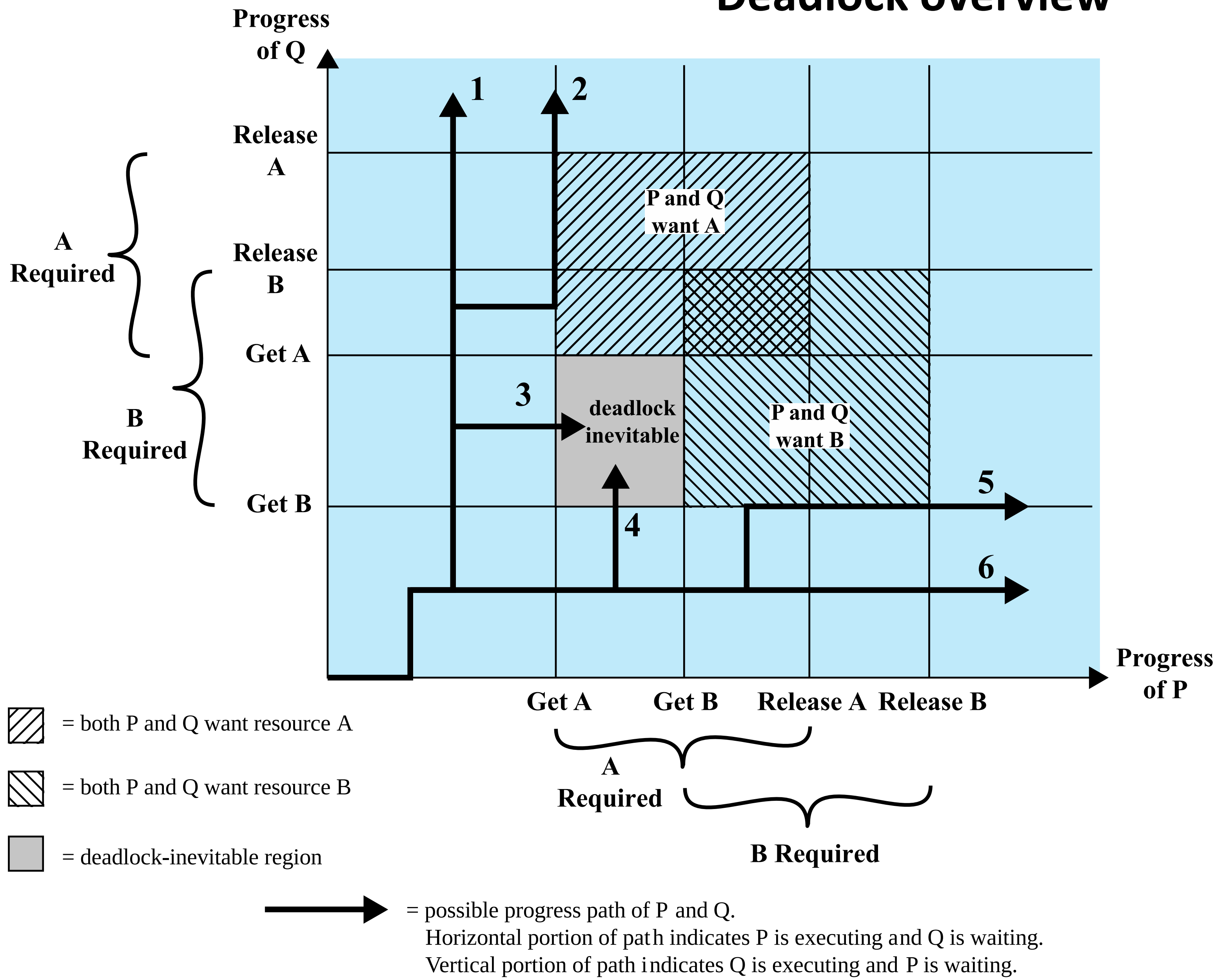
⬜ = deadlock-inevitable region

→ = possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Process P	Process Q
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

Joint process diagram

Deadlock overview



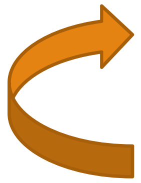
Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
deadlock inevitable
P and Q want B
Progress of P
Get A
Get B
Release A
Release B
A Required
B Required
= both P and Q want resource A
= both P and Q want resource B
= deadlock-inevitable region
= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Process P	**Process Q**
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

Deadlock is only inevitable if this logic stands

Joint process diagram

Considering an alternative scenario

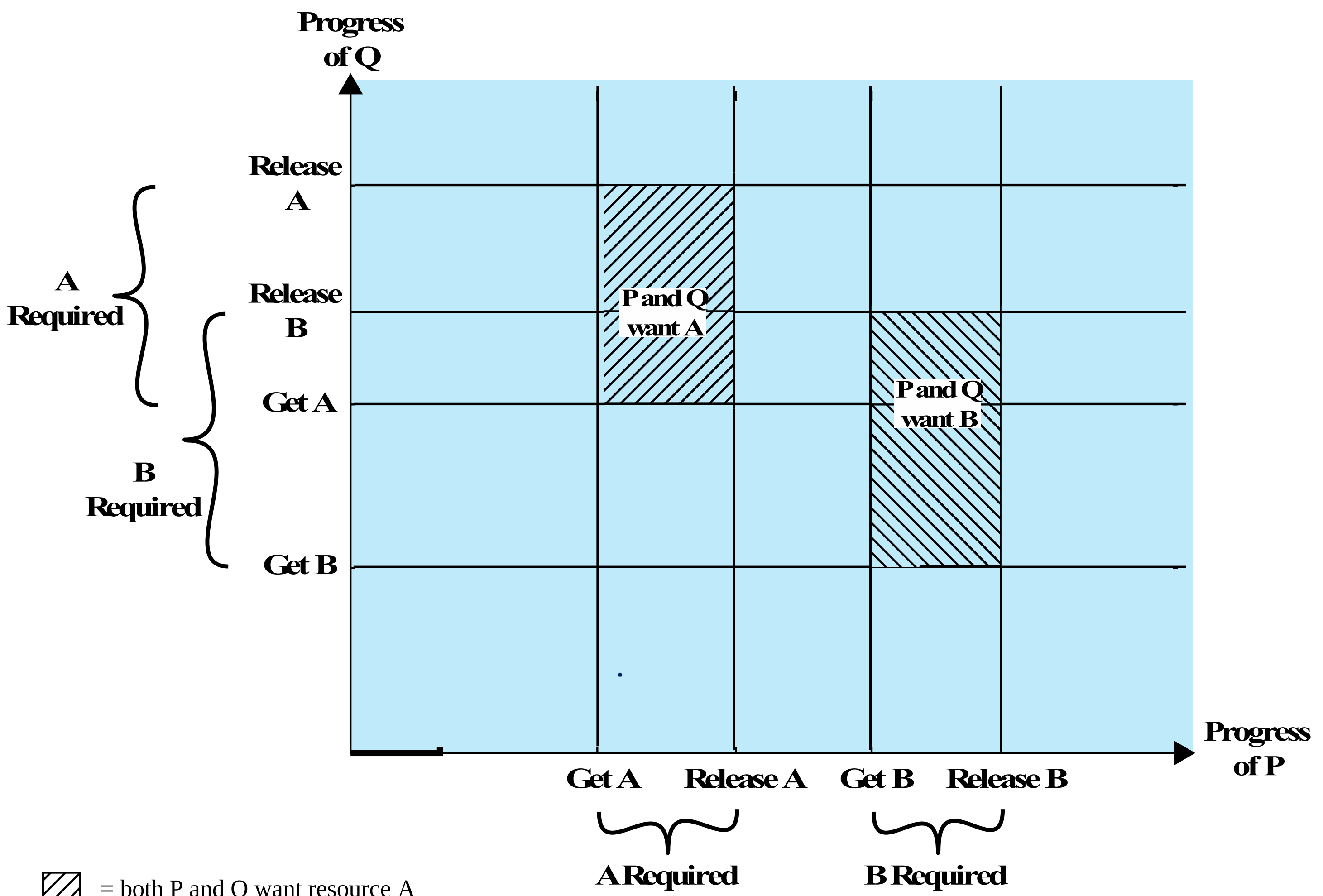


Process P	**Process Q**
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

Considering an alternative scenario

Process P	**Process Q**
…	…
Get A	Get B
…	…
Release A	Get A
…	…
Get B	Release B
…	…
Release B	Release A
…	…

Can deadlock occur with this change?



▨ = both P and Q want resource A

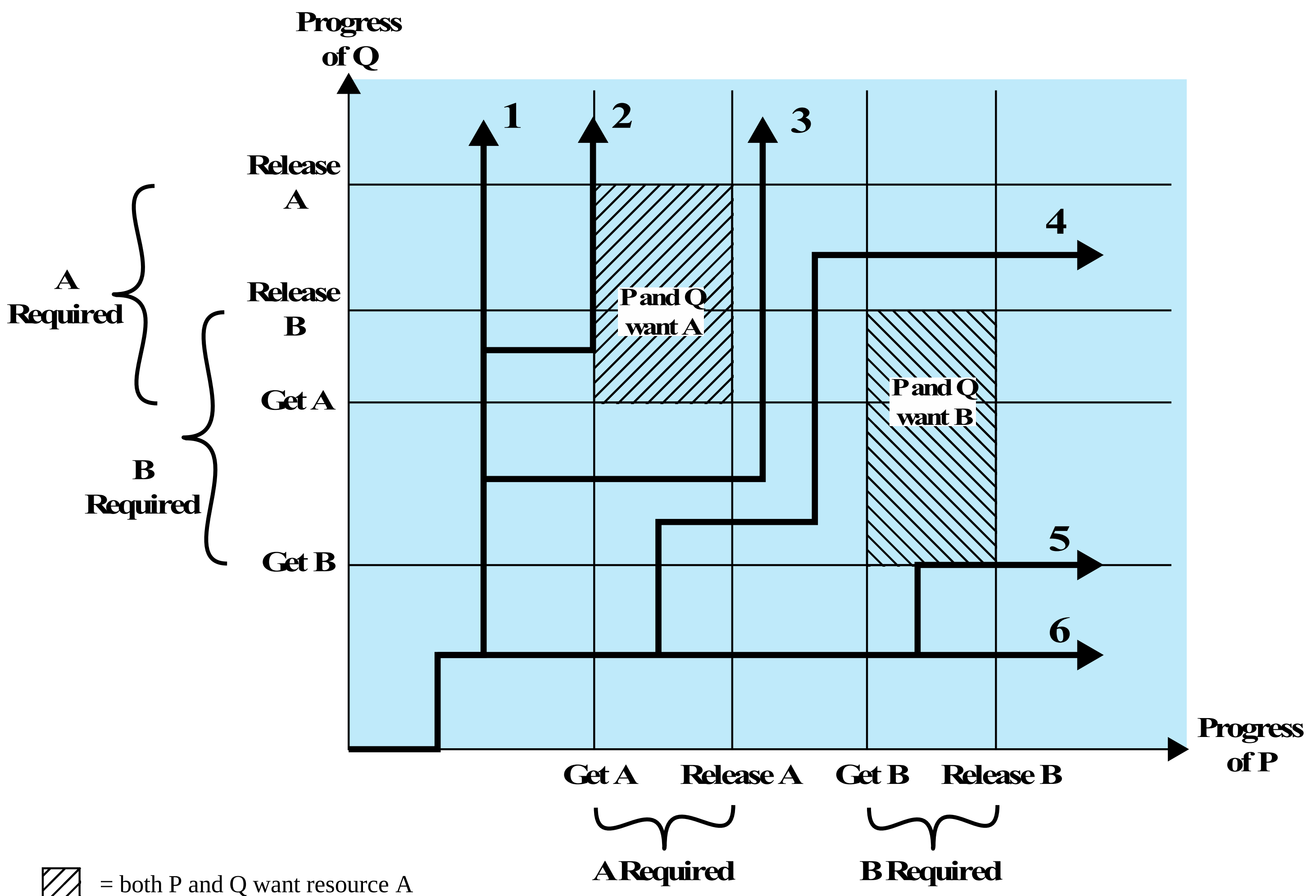
▧ = both P and Q want resource B

→ = possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Process P	Process Q
…	…
Get A	Get B
…	…
Release A	Get A
…	…
Get B	Release B
…	…
Release B	Release A
…	…

Joint process diagram

Can deadlock occur with this change?



▨ = both P and Q want resource A

▧ = both P and Q want resource B

= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Process P	**Process Q**
…	…
Get A	Get B
…	…
Release A	Get A
…	…
Get B	Release B
…	…
Release B	Release A
…	…

Joint process diagram

Resource types

REUSABLE

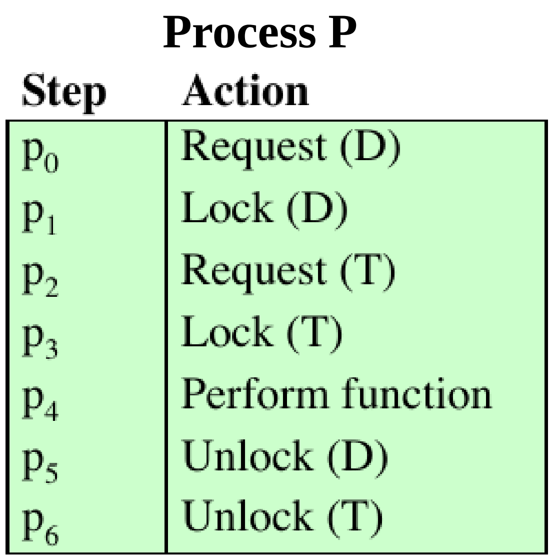
- Can safely be used by only one process at a time
- Once used, the resource can be reused.
- e.g., processors, I/O, main memory, secondary memory, files

CONSUMABLE

- Resources that can be created (produced) and destroyed (consumed)
- e.g., interrupts, signals, messages, data in I/O buffers

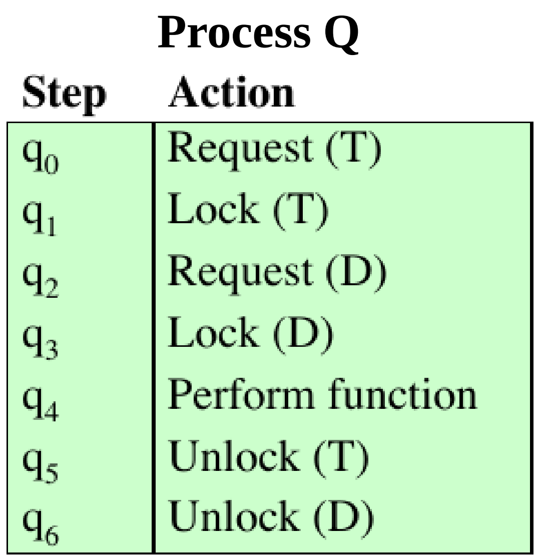
Deadlock with Reusable Resources

- Reusable resources can be used by one process at time and not depleted by use
- Example 1: Two processes accessing a disk drive (D) and a tape drive (T)
- Imagine the sequence of events: p_0 p_1 q_0 q_1 p_2 q_2



Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)



Process Q

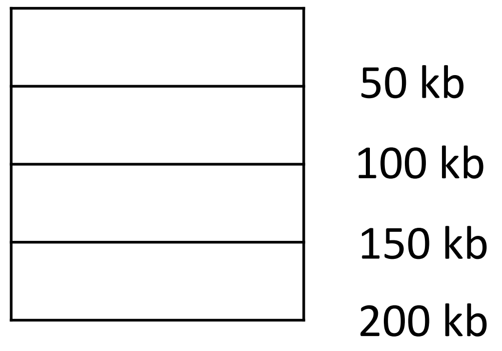
Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Deadlock with Reusable Resources

- Example 2: Space is available for allocation of a maximum of 200KB, and the following sequence of events occur:

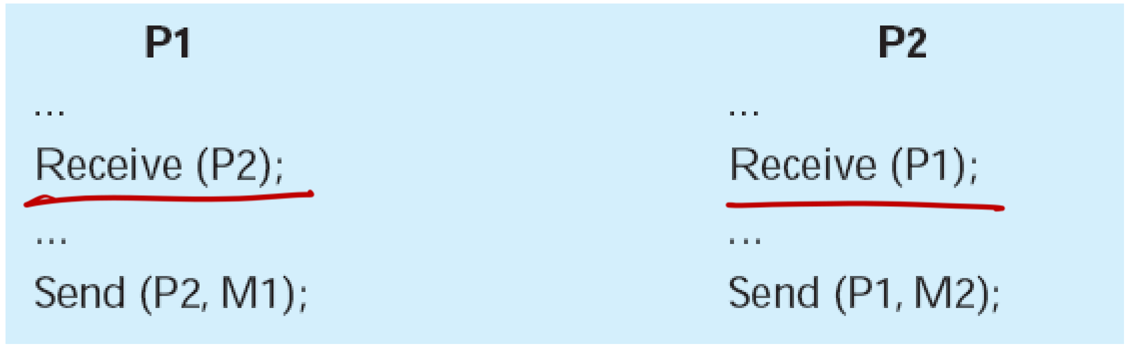
Process 1
. . .
Request 80 Kbytes;
. . .
Request 60 Kbytes;

Process 2
. . .
Request 70 Kbytes;
. . .
Request 80 Kbytes;



Deadlock with Consumable Resources

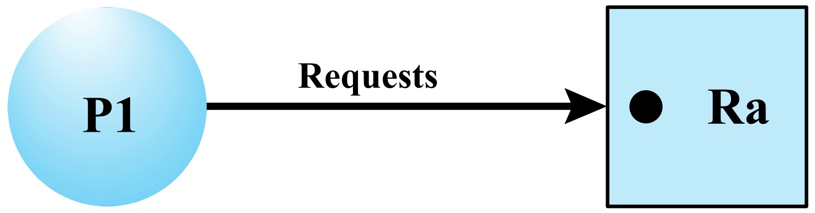
- Consumable resource is one that can be created (i.e., produced) and destroyed (i.e., consumed).
- Consider a pair of processes (P1, and P2). Each process attempts to receive a message from the other process and then send a message to the other process:



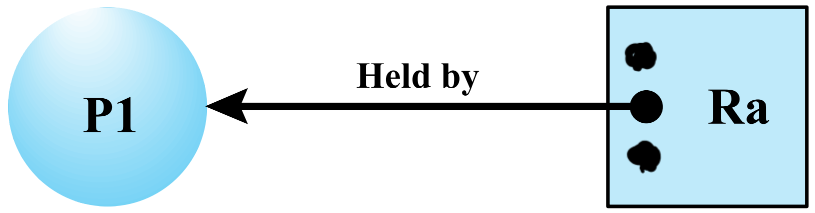
```
      P1                              P2
...                              ...
Receive (P2);                    Receive (P1);
...                              ...
Send (P2, M1);                   Send (P1, M2);
```

- Deadlock occurs if the Receive is blocking

Resource Allocation Graphs



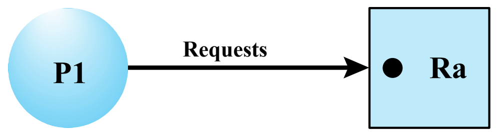
(a) Resouce is requested



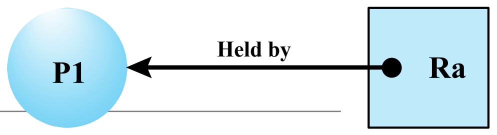
(b) Resource is held

- Square: a resource
- Circle: a process
- Dot: an instance of a resource

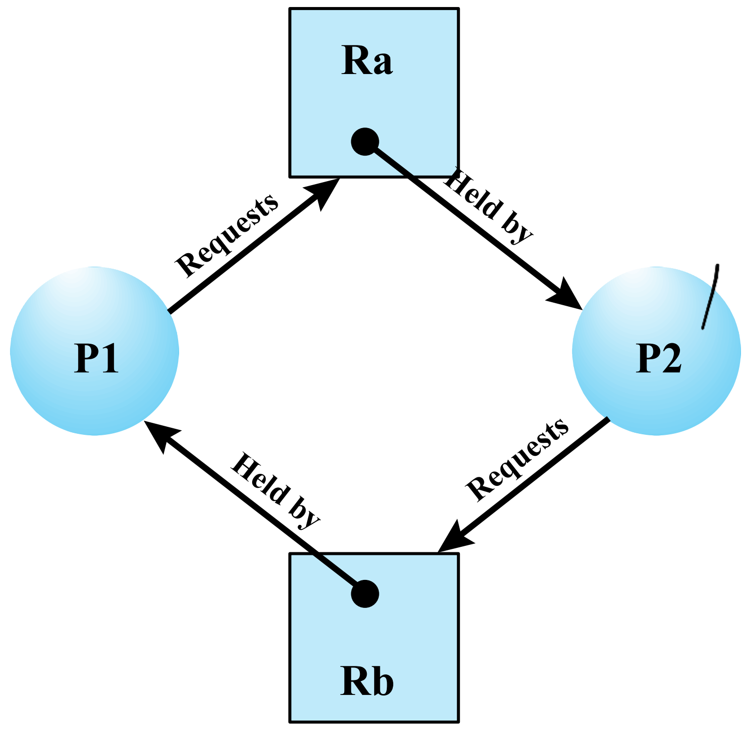
Resource Allocation Graphs



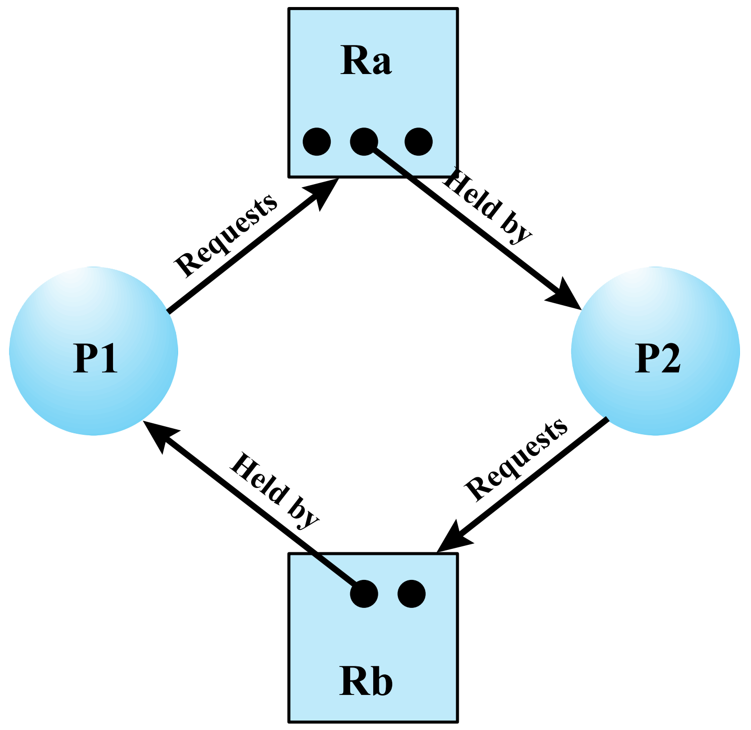
(a) Resouce is requested



(b) Resource is held



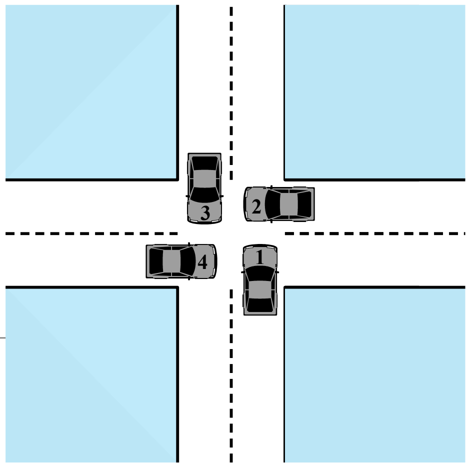
Deadlock

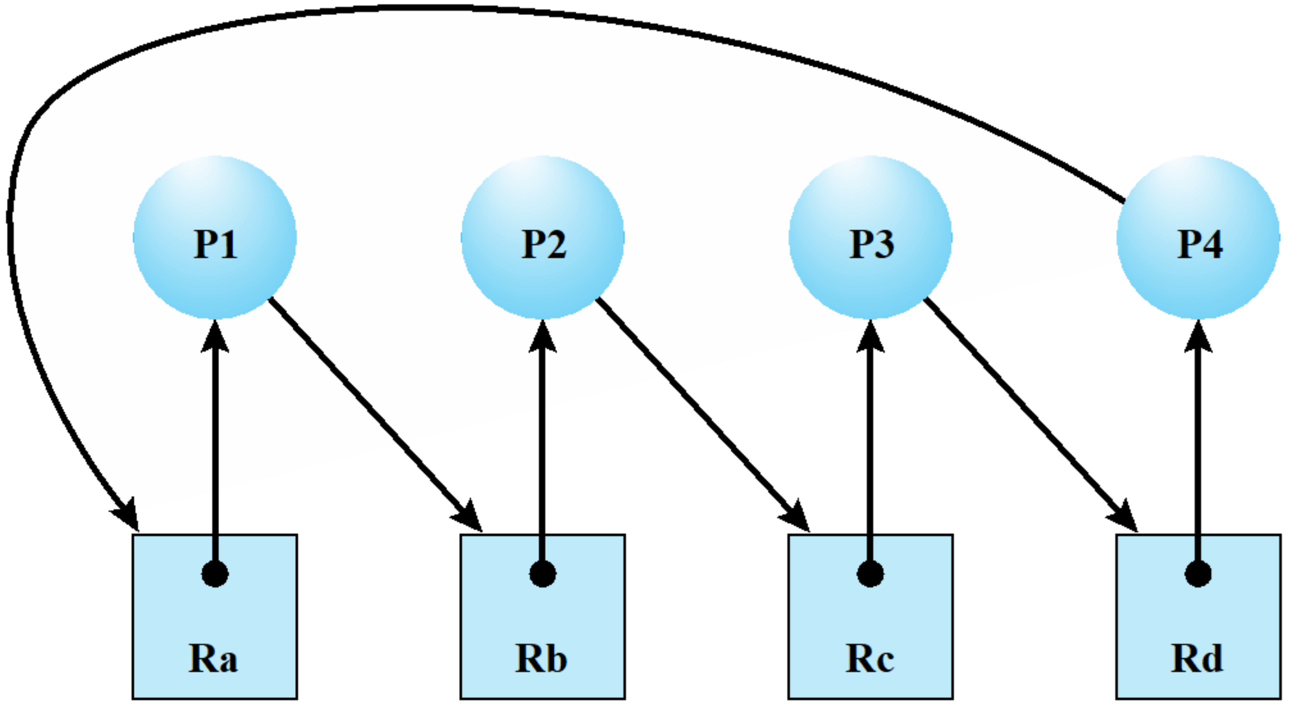


No deadlock

Resource Allocation Graphs

- Example of the traffic deadlock





Conditions for deadlock

CONDITIONS LEADING TO DEADLOCK

- Mutual Exclusion
  - Only one process may use a resource at a time
  - No process may access a resource until that has been allocated to another process
- Hold-and-Wait
  - A process may hold allocated resources while awaiting assignment of other resources
- No Preemption
  - No resource can be forcibly removed from a process holding it

THESE FACTORS LEAD TO…

- Circular Wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

What Can We Do about Deadlocks?

- Prevent Deadlock
  - Adopt a policy that eliminates one of the conditions
  - Can be: direct or indirect
- Avoid Deadlock
  - Deny access to a resource if it leads to deadlock
- Detect Deadlock
  - Attempt to detect the presence of deadlock and take action to recover

Indirect Deadlock Prevention: Can we disable the causes of deadlock?

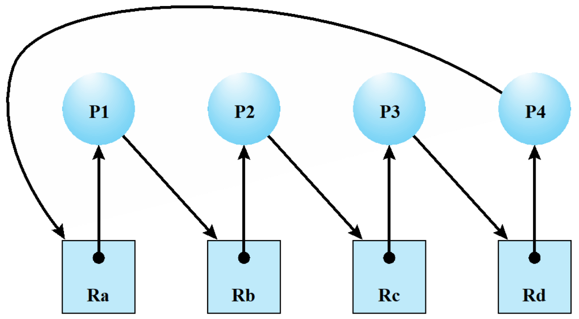
1. Disable mutual exclusion? Nope. Can’t be disabled
2. Disable hold and wait?
   - Requires that process request all of its needed resources at one time and blocking the process until all requests can be granted simultaneously.
   - Drawbacks:
     - Process may be blocked for a long time
     - Resources allocated to a process may remain unused for a considerable time while being denied to other processes
     - Would take a great deal of planning

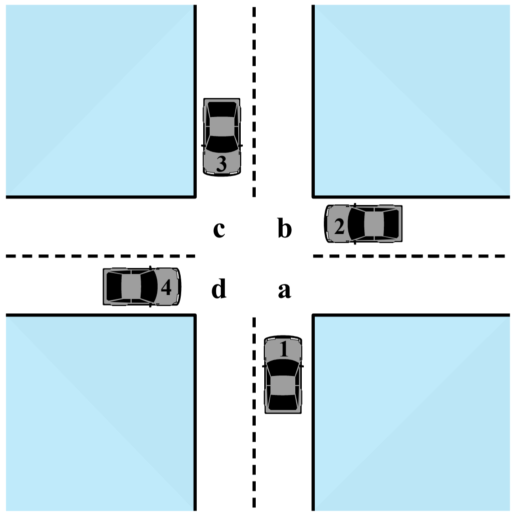
Indirect Deadlock Prevention: Can we disable the causes of deadlock?

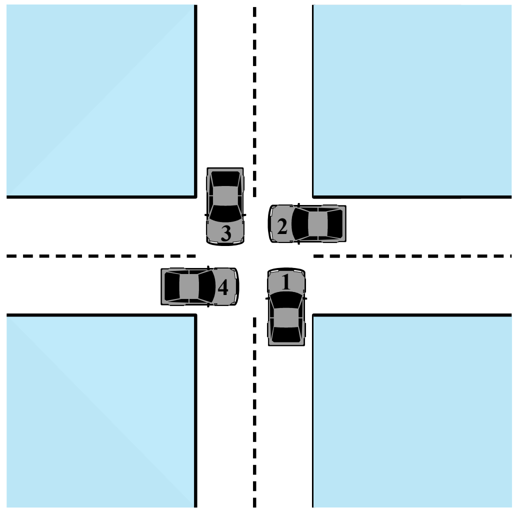
3. Allow preemption?
   - Option 1: If a process holding certain resources is denied a further request, that process must release its original resources and request them again
   - Option 2: or if a resource is held by another process, the OS may preempt the other process and require it to release its resources
   - Drawbacks:
     - Preemption can be applied only to processes of different priorities

Direct Deadlock Prevention: Disabling deadlock causes

4. Circular Wait
   - Define a linear ordering of resource types.
   - Deny resources when their ordering in not linearly increasing or decreasing
   - Example: 4-way intersection, car 4 is denied access to quadrant (d,a), because order is not linearly increasing.







Prevention versus avoidance

- Deadlock *prevention* often works, but makes the system very inefficient
- Deadlock *avoidance* instead allows the four conditions to happen, but does so carefully to avoid deadlocks
  - Also requires knowledge of future process requests
- Two types of avoidance:
  1. Don’t start a process if its demands might lead to a deadlock
  2. Don’t allow a given incremental resource request to a process if this allocation might lead to deadlock

Process/resource states

- n is the number of processes, m is the number of resources
- The **r**esource vector R contains the overall number instances per resource on a system
  - $R = (R_1, R_2, \dots, R_m)$
- The a**v**ailability vector V contains the number of available instances per resource on a system
  - $V = (V_1, V_2, \dots, V_m)$
- **C**laim matrix (the maximum claim for each process of each resource)
  - $\begin{pmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{pmatrix}$ where C_{ij} = maximum requirement of process i for resource j
- **A**llocation matrix
  - $\begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{pmatrix}$ where A_{ij} = current allocation of process i for resource j

Resource vector $(R_1, R_2, \dots, R_m)$

Availability vector $(V_1, V_2, \dots, V_m)$

Claim matrix:

$$\begin{pmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{pmatrix}$$

Allocation matrix:

$$\begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{pmatrix}$$

Process Initiation Denial

- In general, the following must hold
  - $R_j = V_j + \sum_{i=1}^{n} A_{ij}$ for all j
    - i.e., all resources must be either allocated or available
  - $C_{ij} \leq R_j$ for all i, j
    - i.e., No process can claim more than the total amount of resources in the system
  - $A_{ij} \leq C_{ij}$ for all i, j
    - i.e., No process is allocated more resources of any type than the process original claimed to need

Resource vector $(R_1, R_2, \ldots, R_m)$

Availability vector $(V_1, V_2, \ldots, V_m)$

Claim matrix:
$$\begin{pmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{pmatrix}$$

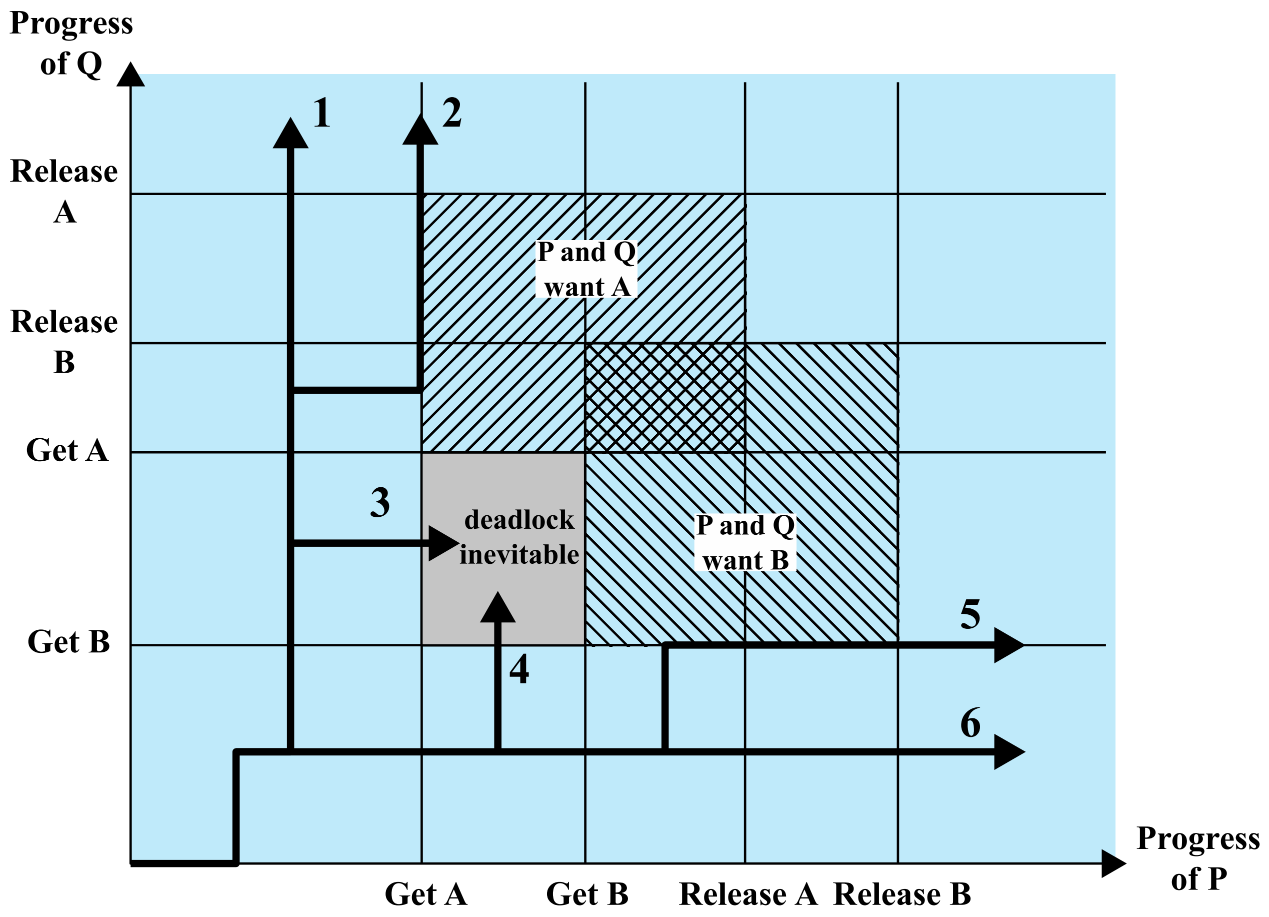
Allocation matrix:
$$\begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{pmatrix}$$

Process Initiation Denial

- So how can we tell if we can safely spawn a process? The following must hold:
  - the *proposed* total amount of claimed resources must be less than or equal to the total available
  - In our math speak, $R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$ for all j
- Does it work well?
  - This works
  - Assumes the worst case scenario
  - Though it's very restrictive

Resource Allocation Denial

- Also referred to as the Banker's algorithm
- The current state of the system can be represented with our R and V vectors, and our C and A matrices
- Safe state: when there is at least one sequence of resource allocations to processes that does not result in a deadlock, i.e. all processes can be run to completion.
- Unsafe state: when there is no sequence of resource allocations



Resource Allocation Denial Example 1

- Example: three resources (R_1, R_2, R_3) and four processes (P_1, P_2, P_3, P_4) with following initial state:
- Is this initial state a “safe” state? How can we calculate this from these values?

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Resource Allocation Denial Example 1

- Example: three resources (R_1, R_2, R_3) and four processes (P_1, P_2, P_3, P_4) with following initial state:
- Is this initial state a “safe” state? How can we calculate this from these values? $C - A$

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

−

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

=

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – **A**

R1	R2	R3
9	3	6

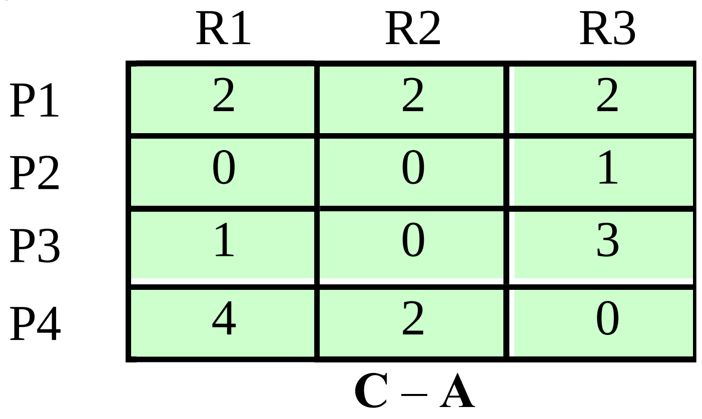
Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

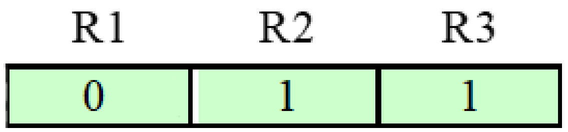
Resource Allocation Denial Example 1

- Example: three resources (R_1, R_2, R_3) and four processes (P_1, P_2, P_3, P_4) with following initial state:
- Is this initial state a "safe" state? How can we calculate this from these values? $C - A$
- What process can precede safely?



	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A



R1	R2	R3
0	1	1

Available vector **V**

Resource Allocation Denial Example 1

- Example: three resources (R_1, R_2, R_3) and four processes (P_1, P_2, P_3, P_4) with following initial state:
- Is this initial state a “safe” state? How can we calculate this from these values? $C - A$

$$\begin{array}{c|ccc} & R1 & R2 & R3 \\ \hline P1 & 3 & 2 & 2 \\ P2 & 6 & 1 & 3 \\ P3 & 3 & 1 & 4 \\ P4 & 4 & 2 & 2 \end{array} - \begin{array}{c|ccc} & R1 & R2 & R3 \\ \hline P1 & 1 & 0 & 0 \\ P2 & 6 & 1 & 2 \\ P3 & 2 & 1 & 1 \\ P4 & 0 & 0 & 2 \end{array} = \begin{array}{c|ccc} & R1 & R2 & R3 \\ \hline P1 & 2 & 2 & 2 \\ P2 & 0 & 0 & 1 \\ P3 & 1 & 0 & 3 \\ P4 & 4 & 2 & 0 \end{array}$$

Claim matrix **C** — Allocation matrix **A** — **C – A**

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Resource Allocation Denial Example 1

- After P_2 is run to completion, its resources can be returned to the pool.

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Resource Allocation Denial Example 1

- Can any of the remaining processes be run to completion?
- Each of the remaining processes can be run to completion
- Therefore, the initial state is safe

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

How does safe/unsafe state help us avoid deadlocks?

- If a process asks for a resource, grant it
- But before continuing, check if the process is in a safe state
- If not, block the process until the state is safe

Resource Allocation Denial Example2 (cont'd)

- Suppose we start with following initial state:

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

- Is this a safe state?
  - Yes, it is safe because P_2 can be run to completion

Resource Allocation Denial Example2 (cont'd)

- If P_1 makes a request for one unit of R_1 and one unit of R_3

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

C - A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

- Is this new state a safe state?
  - No, because none of the processes can be run to completion
  - Thus, P_1 is blocked until the state is safe
  - The new state does not indicate or predict deadlock will happen, but its potential for deadlock

Deadlock Avoidance Advantages and Restrictions

- Advantages
  - It is not necessary to preempt and rollback processes (as in detection)
  - It is less restrictive than deadlock prevention
- Disadvantages and restrictions
  - Maximum resource requirements for each process (i.e., the claim matrix) must be stated in advance
  - Processes under consideration must be independent with no synchronization
  - There must be a fixed number of resources
  - No process may exit while holding resources.

Deadlock Detection

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
- Advantages
  - It leads to early detection
  - The algorithm is simple
- Disadvantages
  - Frequent checks consume processor time

Deadlock Detection Algorithm

- Strategy: Iterate through the processes and determine if they can terminate given the available competing processes.
  - If so, mark them as good
  - If not, leave unmarked
- Those processes that are still unmarked at the end of the algorithm are in a deadlock

Deadlock Detection Algorithm

- Define:
  - ReQuest matrix Q such that Q_{ij} represents the amount of resources of type j requested by process i.
  - Same aVailability vector V as previous algorithms
  - W is a scratch variable
- Initially all processes are unmarked and are in the deadlock set
- Processes are progressively eliminated (marked) from the deadlock set as follows:
  1. Mark each process that has a row in the Allocation matrix as all zeros.
  2. Initialize a temp vector W to V
  3. Find a process row in the Q matrix for which resources are $\leq W$. If no such a row is found, terminate the algorithm
  4. If such a process row is found, mark that process (as not in the deadlock set) and add its Allocation row from A to the W vector
- A deadlock exists if and only if there are unmarked processes at the end of the algorithm

Deadlock Detection Example

- Is there deadlock for the following state?

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

ReQuest Matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

aVailability Matrix V

1. Initial $W = V = (0,0,0,0,1)$
2. P_4 has no allocation , so mark it
3. P_3 requests $\leq W$, so mark P_3 as not in the deadlock set
   - $W = (0,0,0,0,1) + (0,0,0,1,0) = (0,0,0,1,1)$

Terminate the algorithm if P_1, P_2 are still unmarked (i.e., we have deadlock)

Deadlock Recovery Strategies

1. Abort all deadlocked processes
2. Back up each deadlocked process to some previously defined checkpoint and restart all processes
3. Successively abort deadlocked processes until deadlock no longer exists
4. Successively preempt resources until deadlock no longer exists

For (3) and (4), the selection criteria could be one of the following. Choose the process with:

- least amount of processor time consumed so far
- least amount of output produced so far
- most estimated time remaining
- least total resources allocated so far
- lowest priority
- Or a combination of the above

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)
- No philosopher must starve to death (avoid deadlock and starvation)

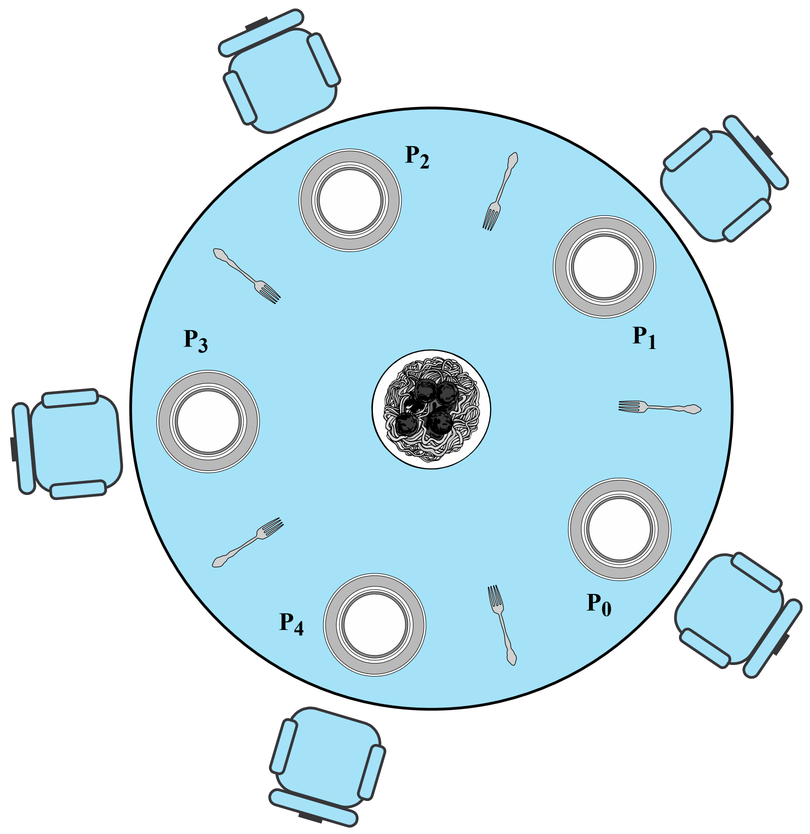
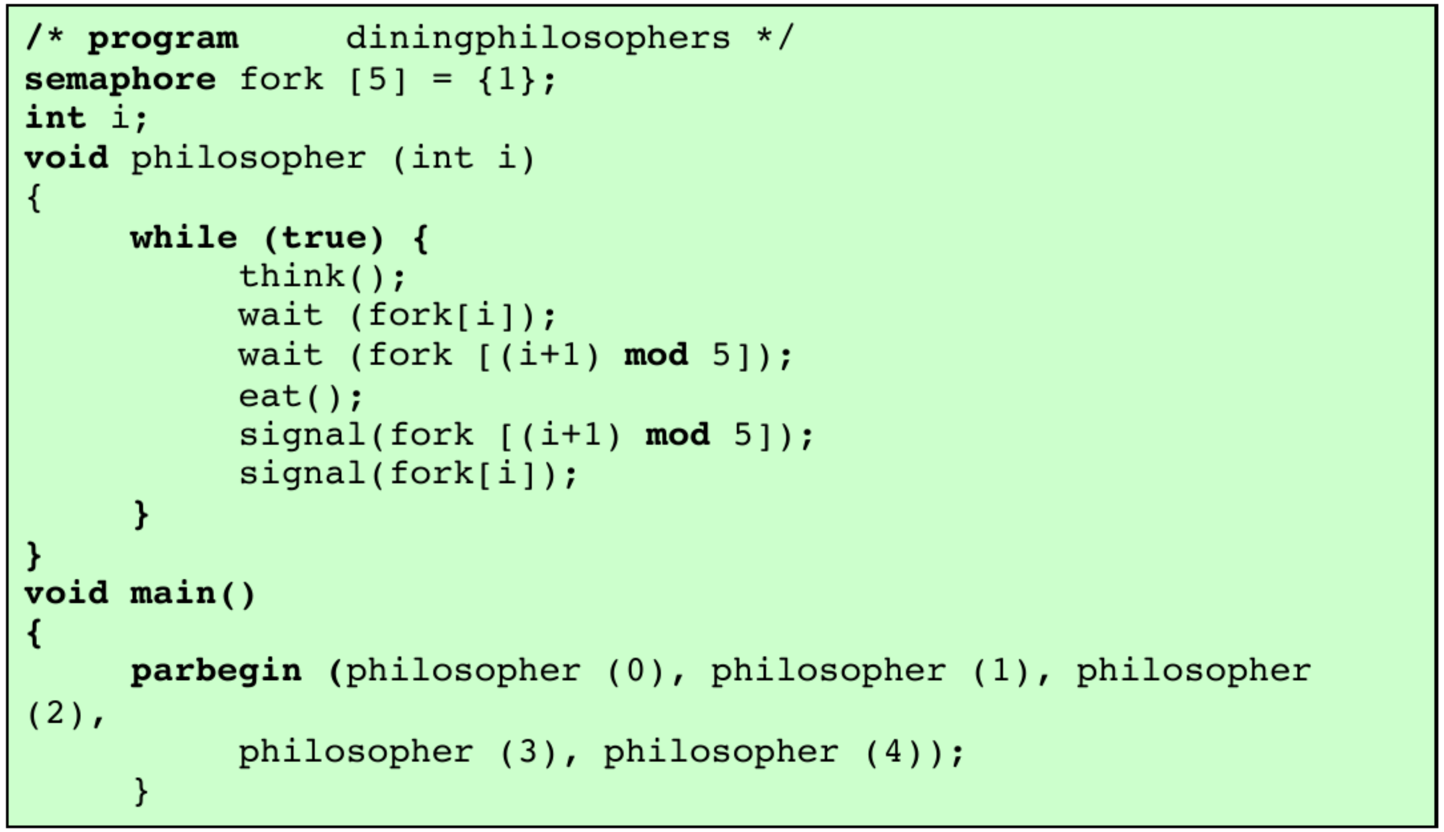


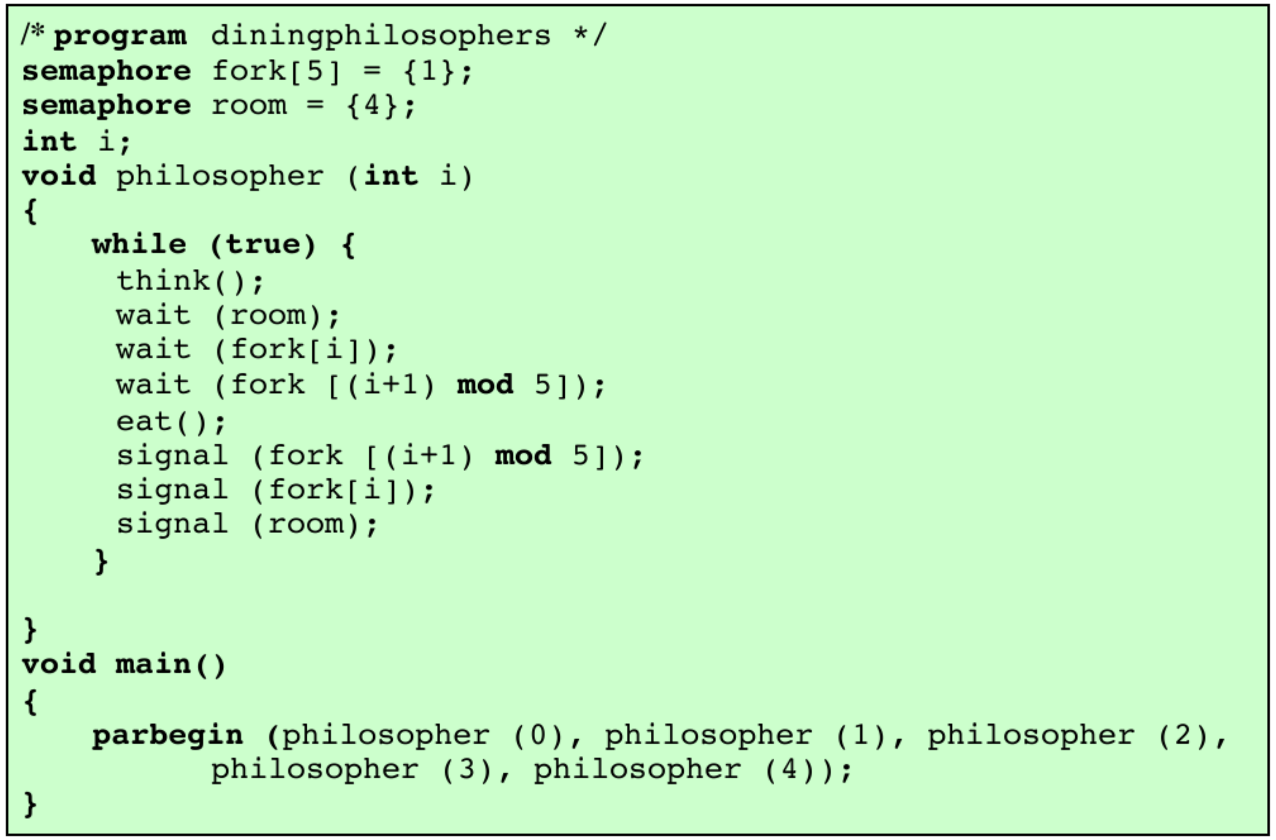
Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem– 1st Solution



```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Dining Philosophers Problem—2nd Solution



```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```