CSE 306 Operating Systems
Deadlock

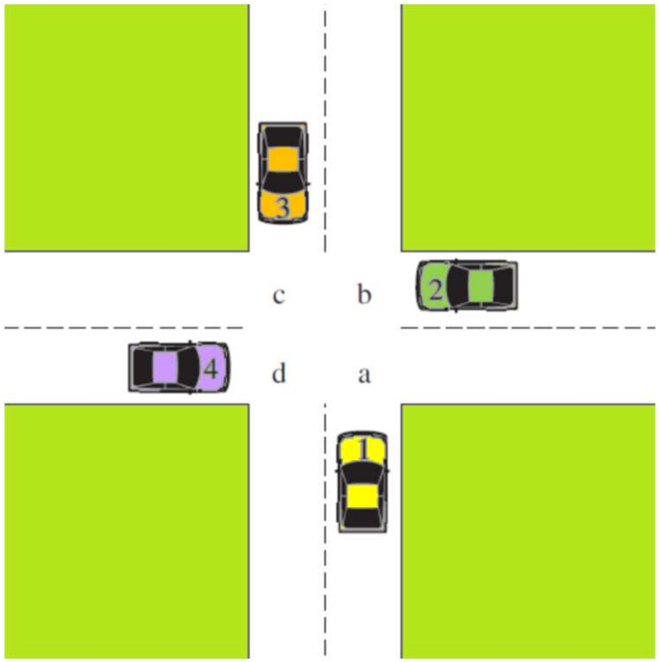
YoungMin Kwon

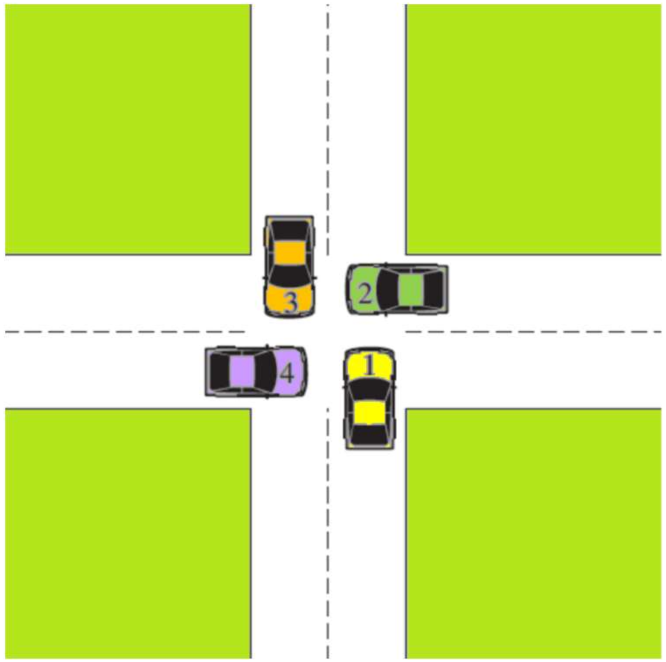
Deadlock

- A set of processes are deadlocked if
  - Each process in the set is blocked and
  - Waiting for an event that can be triggered only from another process in the set

Deadlock

- Illustration of deadlock
  - a, b, c, d are resources and 1, 2, 3, 4 are processes





Joint Progress Diagram

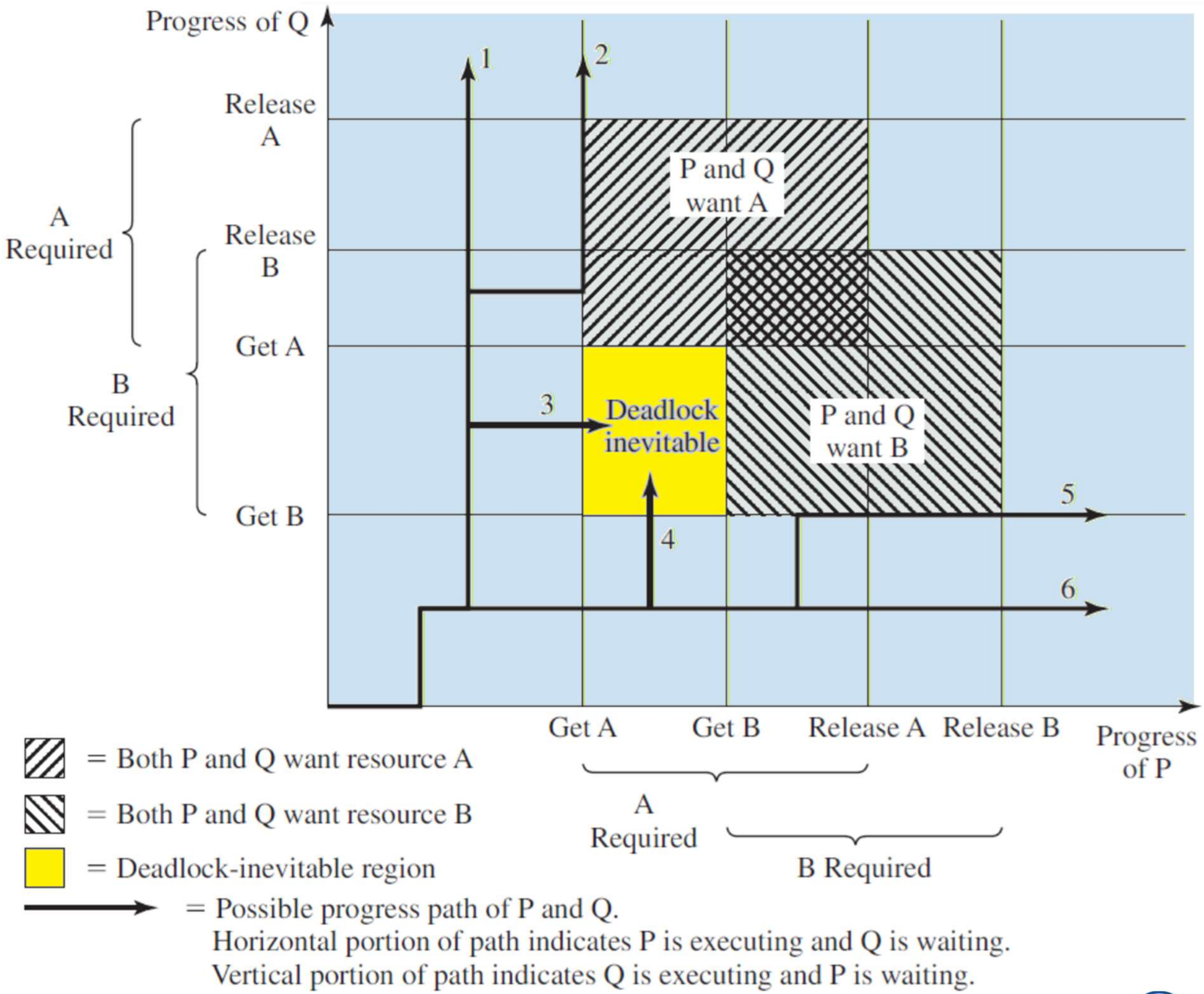
- Illustrates the progress of two processes competing for resources
  - The progress-path moves only from left to right or from bottom to top
- Each process needs exclusive use of both resources
  - Exclusive use forms prohibited regions in the diagram
- Deadlock occurs if the progress-path cannot move

Joint Progress Diagram

- An example of deadlock
  - Two processes P and Q acquire and release resources A and B in the following order

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

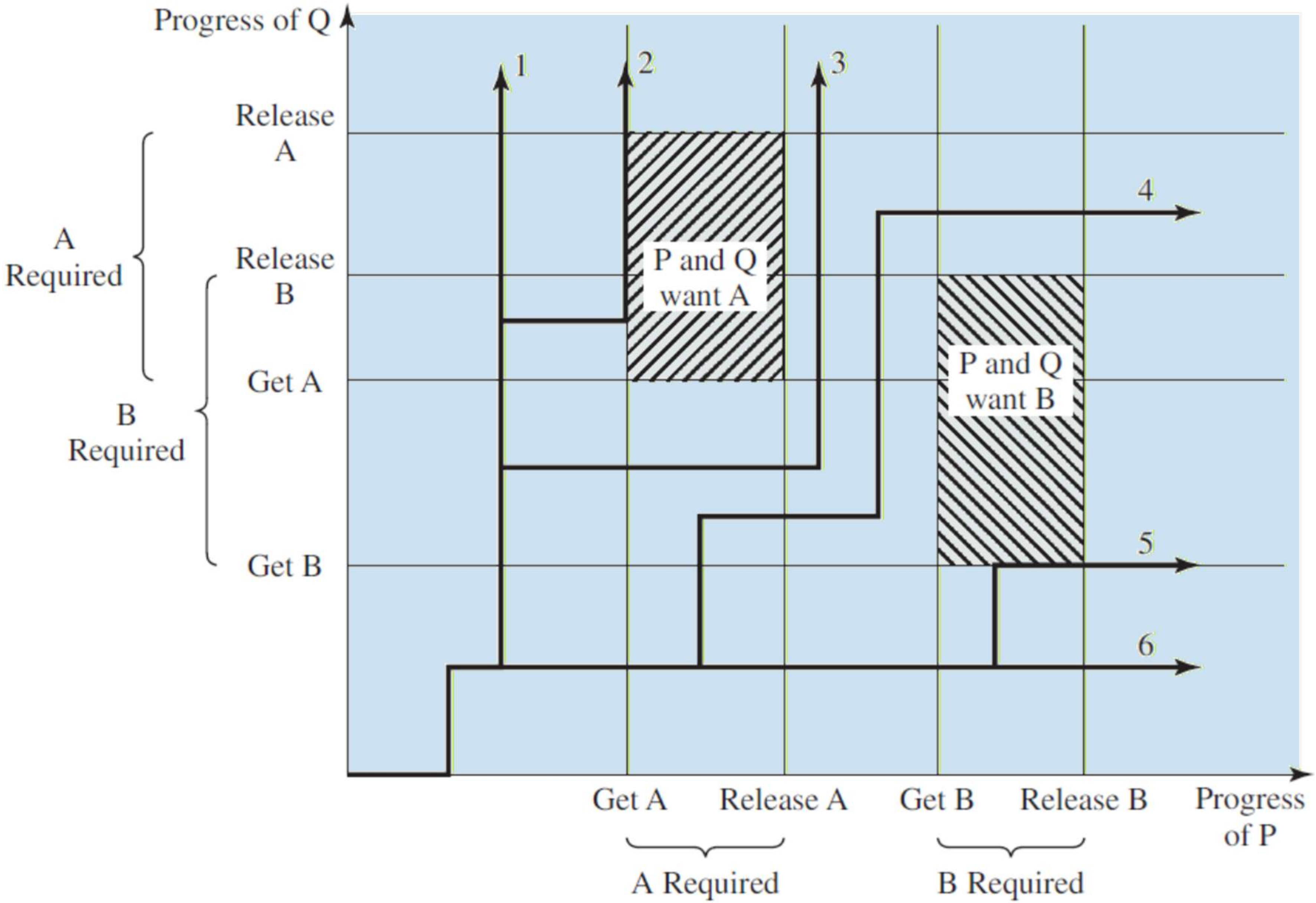
```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```


Progress of Q
1
2
Release A
Release B
Get A
Get B
A Required
B Required
P and Q want A
P and Q want B
3
Deadlock inevitable
4
5
6
Get A
Get B
Release A
Release B
Progress of P
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.

Joint Progress Diagram

- An example of NO deadlock
  - Two processes P and Q acquire and release resources A and B in the following order

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •


Progress of Q
1
2
3
4
5
6
Release A
Release B
Get A
Get B
A Required
B Required
P and Q want A
P and Q want B
Get A
Release A
Get B
Release B
Progress of P
A Required
B Required

Resources

- Reusable resources
  - Resources that can be used by a process at a time and not depleted by that use
  - Processor, I/O channels, memory, device, and data structures (files, DB, semaphores)
  - Deadlock example (200 KB of available memory)

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

Resources

- Consumable resources
  - Resources that can be created and destroyed
  - Interrupts, signals, messages, data in I/O buffers
  - Deadlock example
    - Each process tries to receive a message from the other

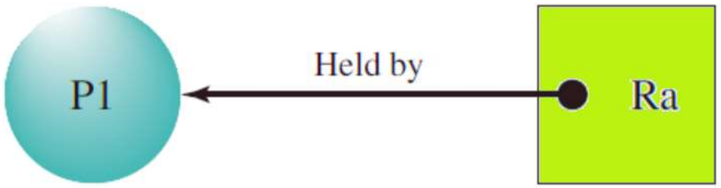
P1	**P2**
…	…
Receive (P2);	Receive (P1);
…	…
Send (P2, M1);	Send (P1, M2);

SUNY Korea

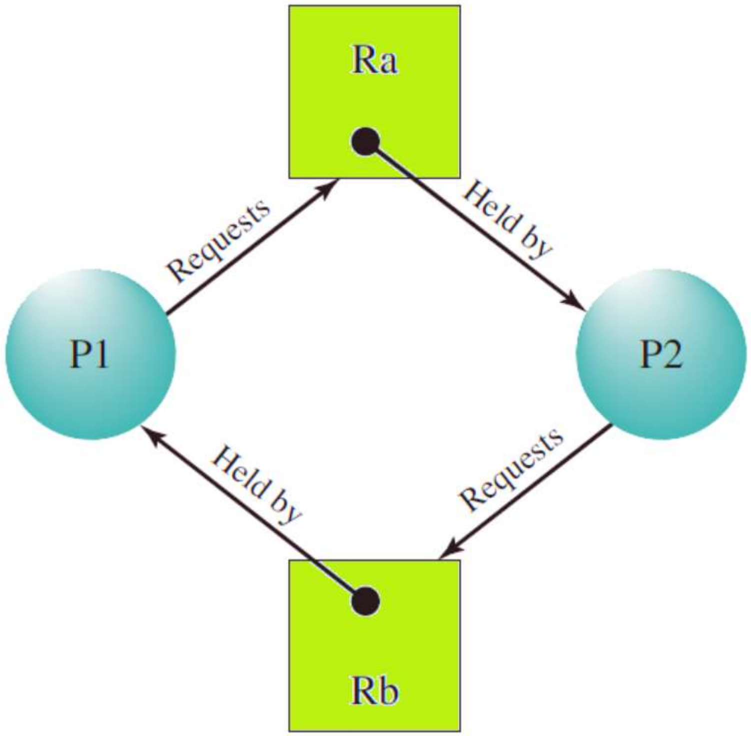
Resource Allocation Graph

- Resource allocation graph
  - Directed graph that depicts the state of the resources and processes
    - Nodes are processes and resources
  - Resource request: the directed edge from the requesting process to the resource

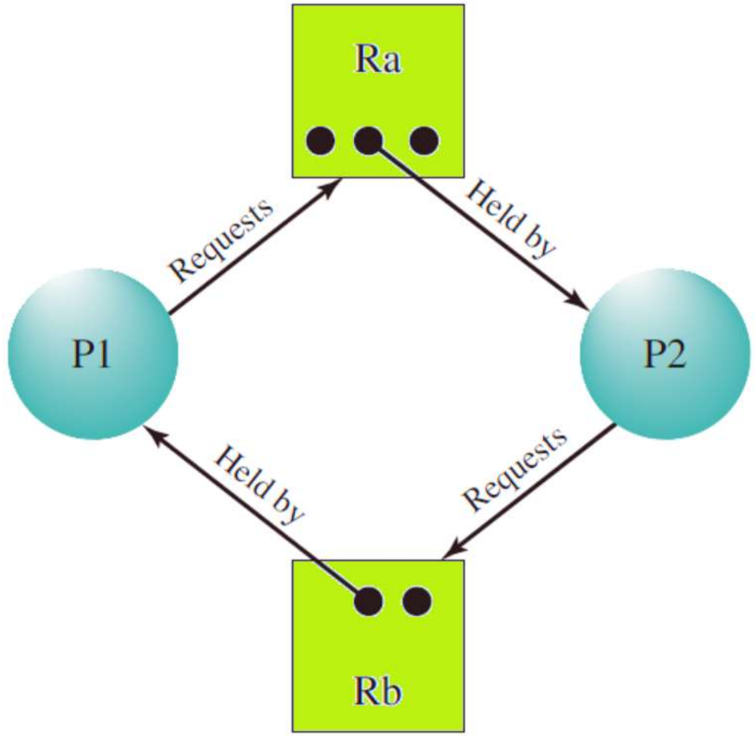
  - Granted resource: the directed edge from the resource to the process



Resource Allocation Graph



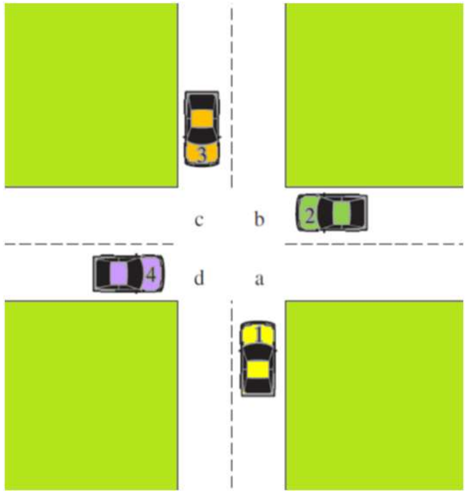
Circular wait: deadlock

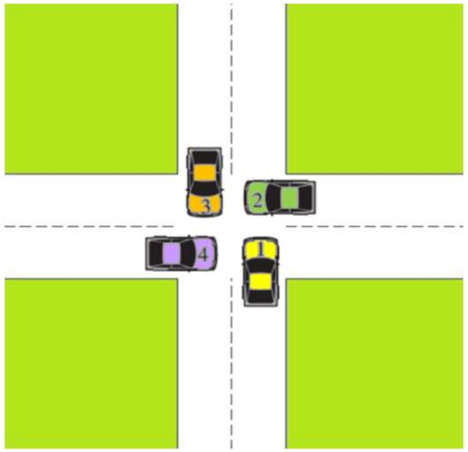


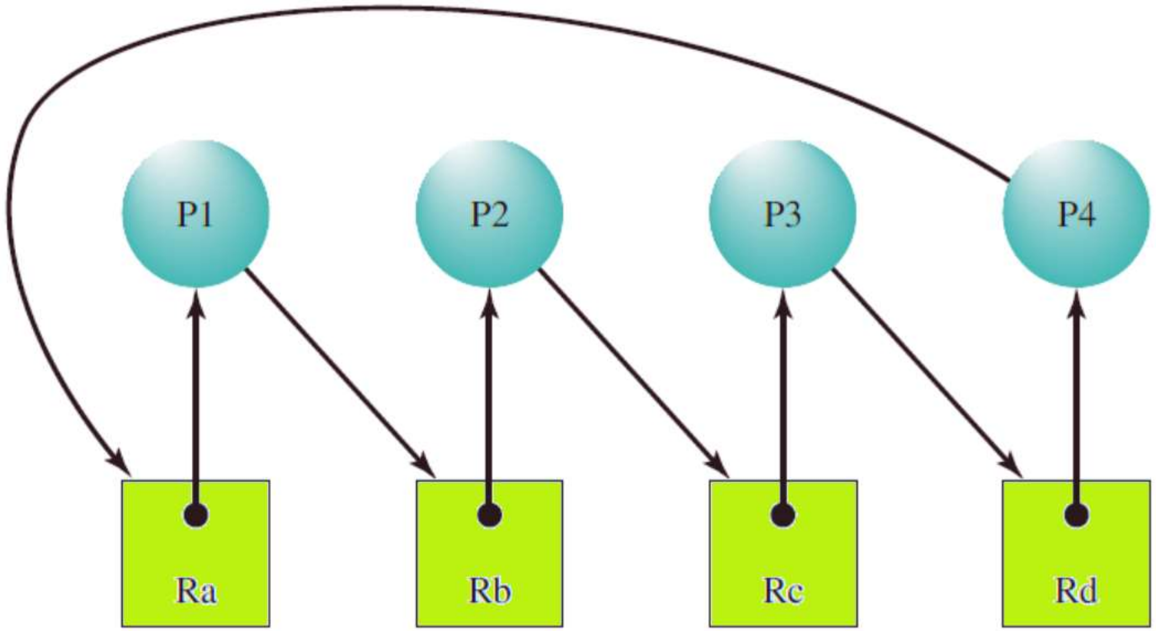
No deadlock:

Ra and Rb are available

Resource Allocation Graph







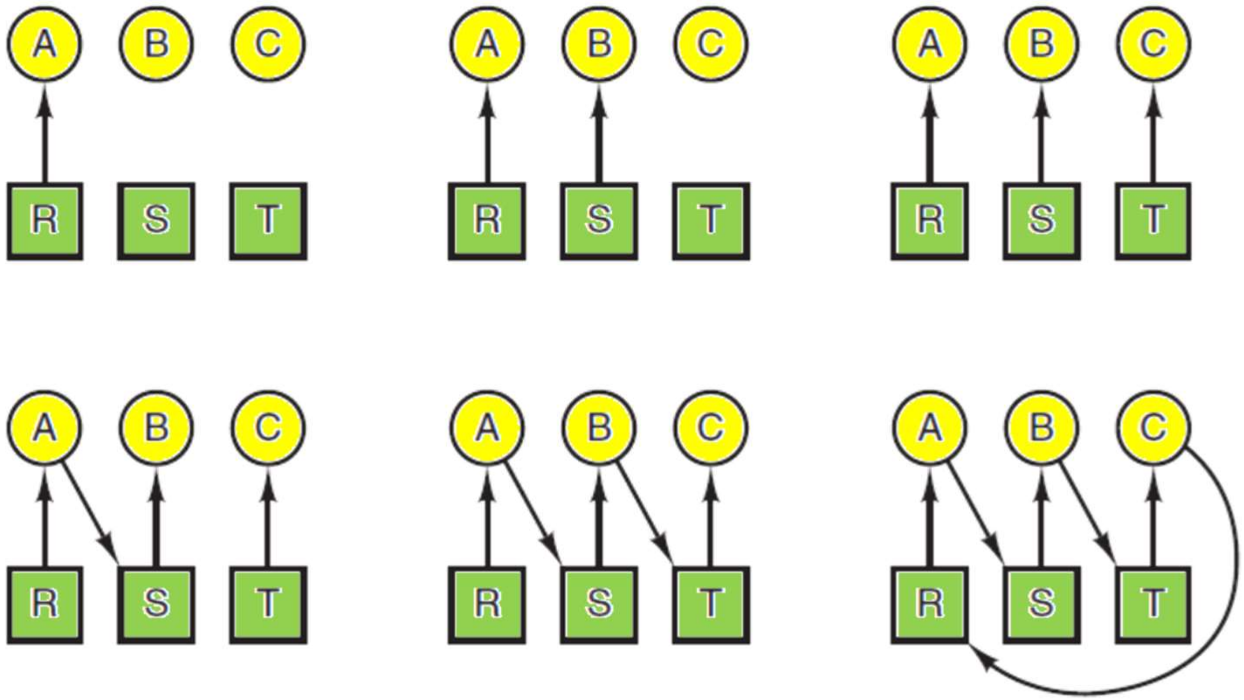
Circular wait: deadlock

Resource Allocation Graph

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

Deadlock sequence

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

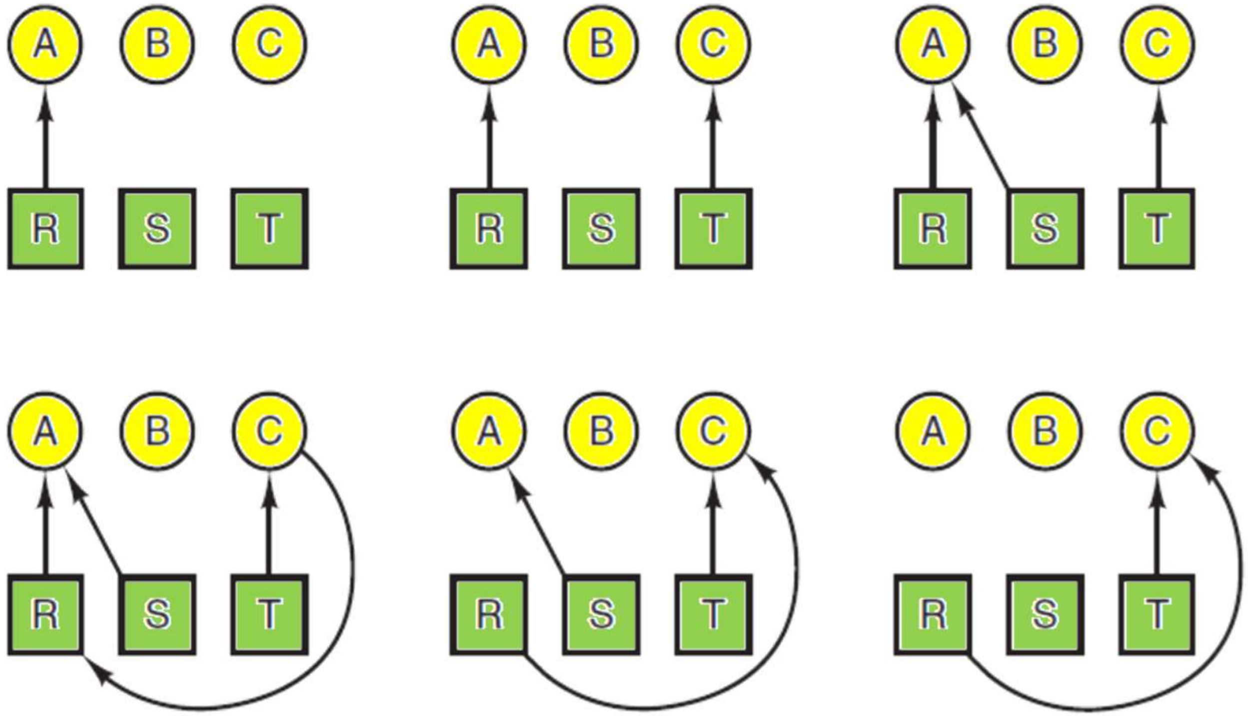


Resource Allocation Graph

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

Deadlock free sequence

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock



4 Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold and wait
  - Processes make requests while holding resources
- No preemption
  - Granted resources cannot be forcefully removed
- Circular wait
  - A closed chain exists in the resource allocation graph

4 Strategies for Deadlock

- Ignore the problem
  - If you ignore it, it will ignore you
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

The Ostrich Algorithm

- The simplest approach
  - Stick your head in the sand and pretend there is no problem
  - Mathematicians' reaction to this problem
    - Unacceptable and deadlock must be prevented at all costs
  - Engineers' reaction to this problem
    - How often the problem is expected?
    - How often the system crashes?
    - How serious the deadlock is?

Deadlock Prevention

- Design a system such that the possibility of deadlock is statically excluded
  - Preventing one of the 4 deadlock conditions

- Mutual exclusion
  - In general, this condition cannot be disallowed

Deadlock Prevention

- Hold and wait
  - Require processes to make request for all necessary resources together
  - Inefficiency of hold and wait
    - Processes may need to wait long when it can make progresses with some of the resources
    - Resources may be held for a long time without being used

Deadlock Prevention

- No preemption
  - Make processes release all resources held if a further request is denied
  - Alternatively, resources held by a process are released if they are requested by other processes
    - Works when no two processes have the same priority
  - Need to save and restore process states

Deadlock Prevention

- Circular wait
  - Define a linear order ($\succ$) of resources
  - If a process has a resource of order R, it can make requests only for resources of order R’ such that R’ $\succ$ R
  - Like the hold and wait prevention, circular-wait-prevention strategy can be inefficient
    - Unnecessarily slowing down processes and denying resource access

Deadlock Avoidance

- More concurrency than the prevention strategies
  - Allows the first three conditions
  - Dynamically decide whether the current resource request, if granted, will potentially lead to a deadlock

- Two approaches
  - Do not start a process if its demand may lead to a deadlock
  - Do not grant a resource request if it may lead to a deadlock (Banker's algorithm)

Process Initiation Denial

- Consider a system of n processes and m types of resources
  - Two vectors Resource, aVailable, and two matrices Claim, and Allocation

Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

Process Initiation Denial

- Relations among Resource, aVailable, Claim, and Allocation

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j	All resources are either available or allocated.
2. $C_{ij} \le R_j$, for all i,j	No process can claim more than the total amount of resources in the system.
3. $A_{ij} \le C_{ij}$, for all i,j	No process is allocated more resources of any type than the process originally claimed to need.

- Deadlock avoidance policy

Start a new process P_{n+1} only if

$$R_j \ge C_{(n+1)j} + \sum_{i=1}^{n} C_{ij} \quad \text{for all } j$$

Banker's Algorithm

- A system state comprises
  - Resource, aVailable, Claim, and Allocation
- Safe state
  - There is a sequence of resource allocations that can make all processes run to complete (without deadlock)
  - A process i can run to completion if

$$C_{ij} - A_{ij} \leq V_j, \quad \text{for all } j$$

- Unsafe state
  - A state that is not safe

Banker's Algorithm (Safe 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	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

- Decide whether this initial state is safe
- P2 can run to complete

Banker's Algorithm (Safe State?)

	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

- P2 is complete; fill the 2nd row of C and A with 0
- Update V with the resources held by P2
- Now, P1 can run to complete

Banker's Algorithm (Safe State?)

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
7	2	3

- P1 is complete; fill the 1st row of C and A with 0
- Update V with the resources held by P1
- P3 can run to complete

Banker's Algorithm (Safe State?)

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

Claim matrix C

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

Allocation matrix A

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

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

- P3 is complete; fill the 3rd row of C and A with 0
- Update V with the resources held by P3
- P4 can run to complete
- As all processes can run to complete the initial state is safe

Banker's Algorithm

- Banker's algorithm
  - Grant resources only when the resulting state will be safe

Banker's Algorithm Example 1

	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

- Given the state, if P2 requests for 1 R1 and 1 R3

Banker's Algorithm Example 1

	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

- Above is the resulting state if the request is granted
  - The resulting state is the same as the initial state of the previous example
  - Grant the resources because the resulting state is safe

Banker's Algorithm Example 2

	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

- Given the state, if P1 requests for 1 R1 and 1 R3

Banker's Algorithm Example 2

	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	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
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
0	1	1

Available vector V

- Above is the resulting state if the request is granted
  - This state is unsafe as no process can run to complete
  - Thus, the request should not be granted

Deadlock Detection

- Deadlock detection strategy
  - Do not limit resource requests
  - Periodically check if there is a deadlock
    - Assuming that the current requests are all that are needed for processes to complete
    - Check if the current requests can be satisfied by the available resources
  - If a deadlock is detected recover from it

Deadlock Detection Algorithm

- Mark each process that has zero row vector in A
  - A process not holding a resource cannot be a part of deadlocked processes

- Initialize a temporary vector W (copy V to W)
- Find an unmarked process i such that the i^{th} row of a request matrix Q is less than or equal to W
  - **reQest matrix**: Q_{ij} is the amount of resources of type j requested by process i
  - Terminate the algorithm if no such process is found
- If such a row is found
  - Mark process i
  - Add i^{th} row of A to W and go to the 3^{rd} step

- Any unmarked processes are deadlocked processes

Deadlock Detection Example

	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

Available vector

- Mark P4 because P4 has no allocated resources
- Copy V = [0 0 0 0 1] to W
- Because the 3rd row of Q is less than or equal to W
  - Mark P3 and update W as
  - W = W + [0 0 0 1 0]
= [0 0 0 1 1]
- Terminate because no other unmarked process has a row in Q that is less than or equal to W
- P1 and P2 are unmarked and they are deadlocked

Recovery

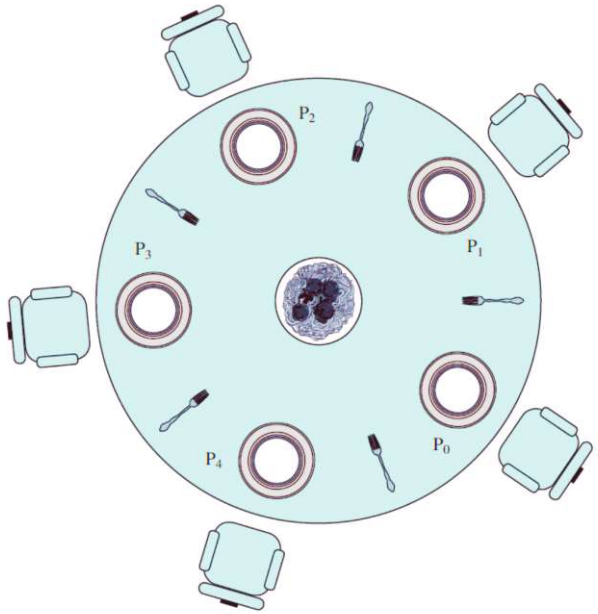
- Abort all deadlocked processes
  - One of the most common approaches

- Rollback all deadlocked processes to some previously defined checkpoint and restart

- Successively abort deadlocked processes until deadlock is removed

- Successively rollback processes to a checkpoint and restart until deadlock is removed

```c
// Dining Philosophers Problem
#define N 5
typedef struct {
    int id;
    sem_t *left;
    sem_t *right;
} Philosopher;

void *thread_func(void *vargp) {
    Philosopher *p = (Philosopher*)vargp;
    int i;
    for(i = 0; i < 100; i++) {
        fprintf(stderr, "%d: thinking\n", p->id);
        fprintf(stderr, "%d: getting left\n", p->id);
        sem_wait(p->left);
        fprintf(stderr, "%d: getting right\n", p->id);
        sem_wait(p->right);
        fprintf(stderr, "%d: eating\n", p->id);
        fprintf(stderr, "%d: putting left\n", p->id);
        sem_post(p->left);
        fprintf(stderr, "%d: putting right\n", p->id);
        sem_post(p->right);
    }
}
```



```c
int main() {
    pthread_t tid[N];
    sem_t stick[N];
    Philosopher p[N];
    int i;
    for(i = 0; i < N; i++) {
        sem_init(stick+i, 0/*pshared*/, 1/*value*/);
        p[i].id = i;
        p[i].left  = &stick[i % N];
        p[i].right = &stick[(i+1) % N];
    }

    for(i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, thread_func, &p[i]);
    for(i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    for(i = 0; i < N; i++)
        sem_destroy (stick+i);
    return 0;
}

//in gdb, try info threads, thread #, bt
```