CIS 5512 - Operating Systems
Synchronization and Deadlock

Professor Qiang Zeng

Previous class

- Restroom problem
- Bar problem
- Enforcing execution order
- Single-slot producer-consumer problem
- Multi-slot producer-consumer problem

Barrier Problem

Barrier problem

- Goal: given a number, N, of processes, each process has to wait at some point of its program until all processes reach the point
- Implement the API Barrier(), which is called by each process
  - The N-1 processes block until the last one calls it

Solution

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
Barrier() {
  down(mutex)
  count += 1
  if (count == n)
    for (i = 0; i < n; ++i)
      up(barrier)
  up(mutex)

  down(barrier)
}
```

Another solution

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()
```

Is it possible that two processes both arrive here and find "count == n"

A: It is possible. But extra up() operations will not cause errors. Certainly, you can move the "if" statement into mutex-guarded region

Readers-Writers Problem

Readers-Writers Problem

- Problem statement:
  - *Reader* threads only read the object
  - *Writer* threads modify the object
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object
- Occurs frequently in real systems, e.g.,
  - Online airline reservation system
  - Multithreaded caching Web proxy

Solution

Shared

:

```
int readcnt;    /* Initially = 0 */
semaphore r, whole; /* Initially = 1 */
```

Writers

:

```
void writer(void)
{
    while (1) {
        down(whole);

        /* Critical section */
        /* Writing here */

        up(whole);
    }
}
```

Solution

What if the "whole" lock is already acquired by the writer, and the first reader comes in?

Readers:

```
void reader(void)
{
   while (1) {
      /*Increment readcnt*/
      down(r); /*Only one reader a time*/
         readcnt++;
      if (readcnt == 1) /* First reader in */
         down(whole); /* Lock out writers */
      up(r);

      /* Read; mutliple readers may be here */

      /*Decrement readcnt*/
      down(r);
      readcnt--;
      if (readcnt == 0) /* Last out */
         up(whole); /* Let in writers */
      up(r);
   }
}
```

Previous class…

What is a binary semaphore? A binary semaphore can only be used as a mutex?

A mutex is a lock for mutual exclusion. A binary semaphore can be used

(1) as a mutex for mutual exclusion
(2) for synchronization of concurrent use of resources
(3) for enforcing the order of operations of processes

Summary of the uses of Semaphore

- Mutual exclusion (using binary semaphores)
- Synchronizing the use of shared resources, e.g.,
  - The single-slot restroom problem
  - The bar problem
  - The producer-consumer problem
  - The counter of the semaphore should be initialized to the # of resources available
- Enforcing order, e.g.,
  - Operation O1 in Process P1 has to occur after O2 in P2

Relations between Condition Variable & Monitor

- A Monitor may contain zero or more CVs
  - Very often, procedures in Monitor rely on CVs to implement complex synchronization
  - Recall that a CV has to be used with a lock; a Monitor can provide the lock, so you do not have to explicitly use a lock for employing a CV in a Monitor
- The use of CVs is not limited to Monitors
  - E.g., Pthread library provides CVs but not Monitors

Condition variable VS Semaphore

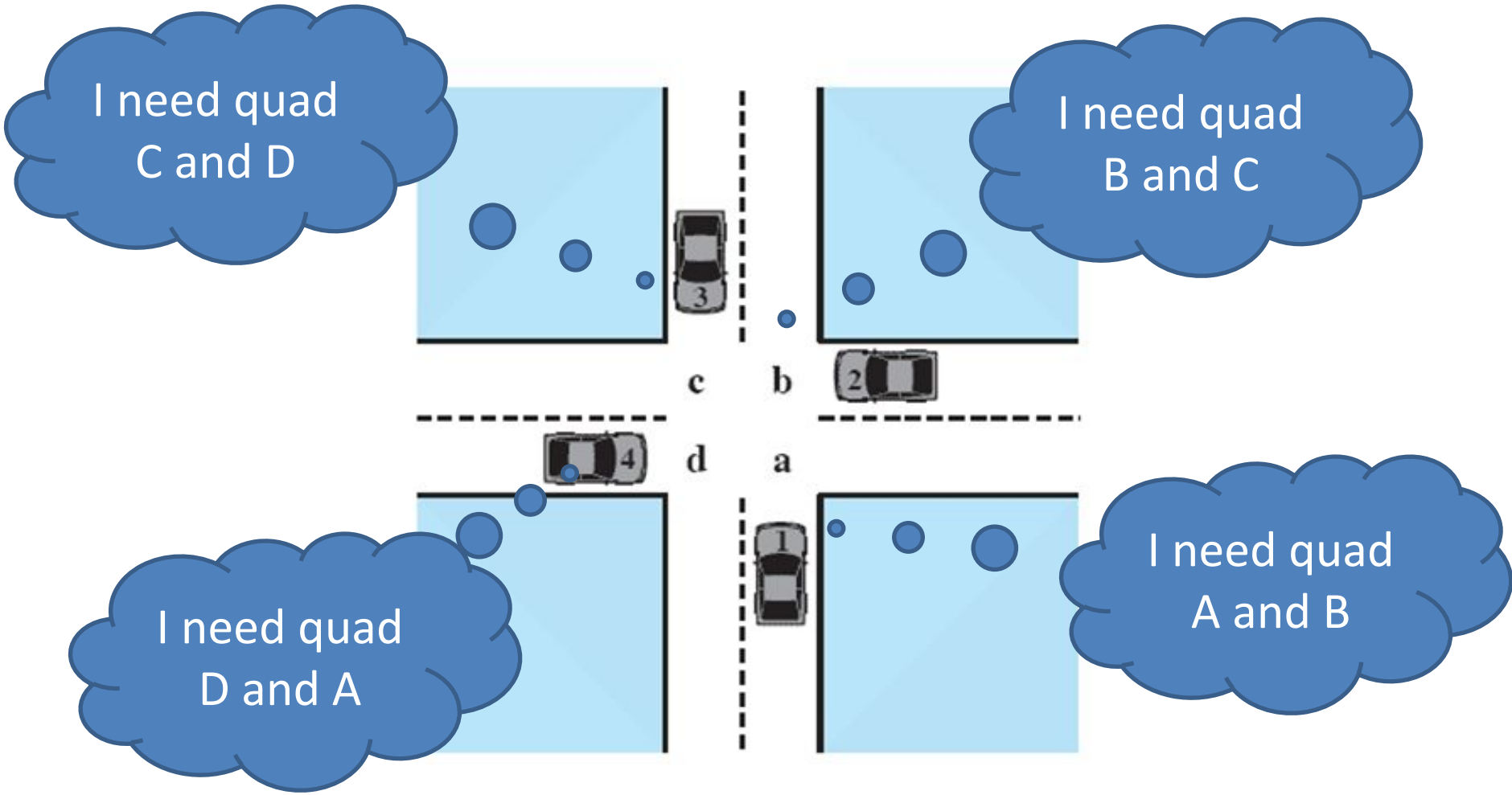
- A CV has to work with a lock (e.g., the lock provided by a monitor), while a Semaphore does not
- Condition Variables allow broadcast() operation, while Semaphores do not
- A Semaphore has a counter and a wait queue, while a Condition Variable only has a wait queue
  - You need to initialize the counter when using a Semaphore. A Condition Variable has no notion of "the number of resources"
  - If there are no processes in the wait queue
    - The up() operation of a semaphore will increment the counter
    - The signal() operation of a CV will have no effect (i.e., the "signal" gets lost)

Deadlock

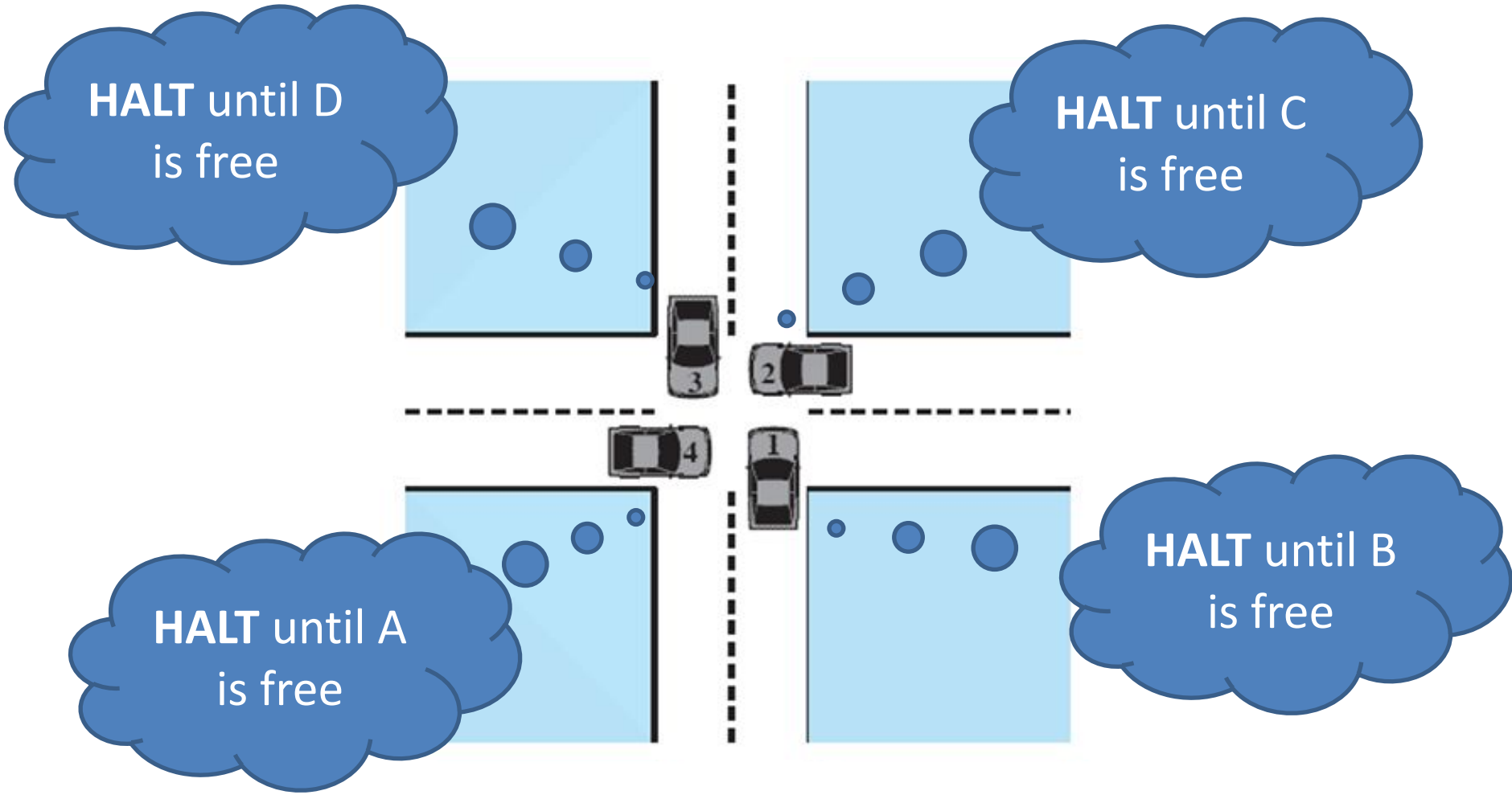
- A *set* of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set

Some Slides Courtesy of Dr. William Stallings

Potential Deadlock



Actual Deadlock



Resource Categories

Reusable

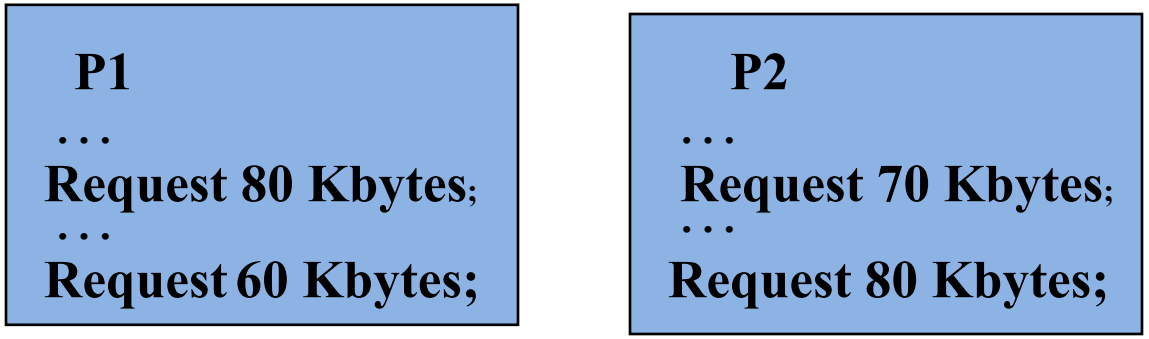
- can be safely used by only one process at a time and is not depleted by that use
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information in I/O buffers

Example of Deadlock: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:



- Deadlock occurs if both processes progress to their second request

Example of Deadlock: waiting for messages

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

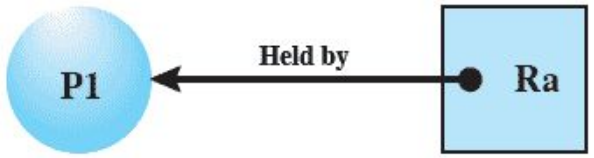
```
S1 = 1; s2 = 1;
```

```
P1:

P(s1)
V(s2)
```

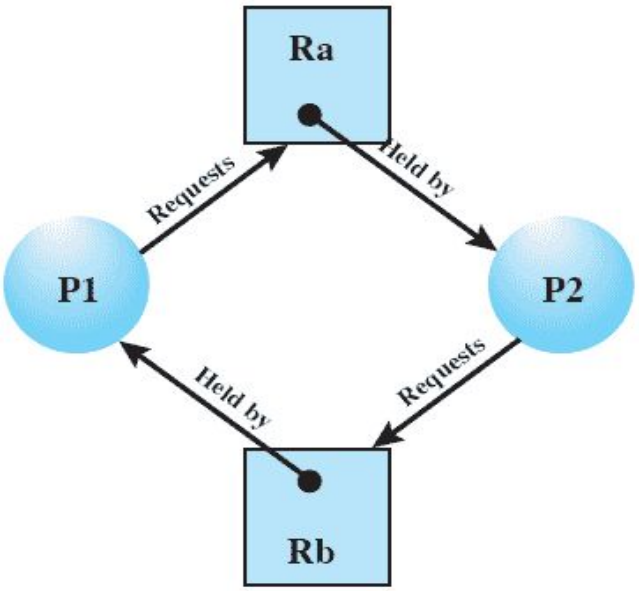
```
P2:

P(s2)
V(s1)
```

Resource Allocation Graph

(a) Resouce is requested



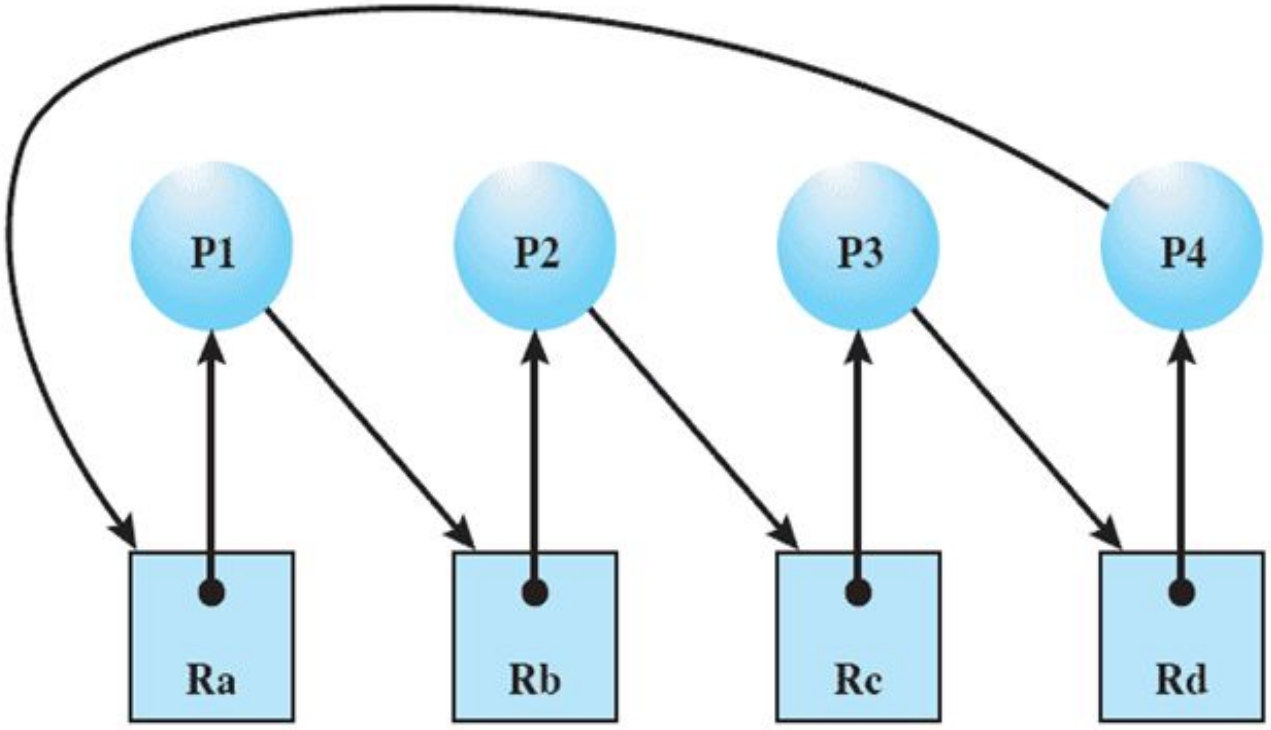
(b) Resource is held



(c) Circular wait

There is a circle in the graph, which indicates deadlock

Resource Allocation Graph describing the traffic jam



Resource Allocation Graph

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• A process cannot access a resource that has been allocated to another process	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

Prevent Deadlock

- adopt a policy that eliminates one of the conditions

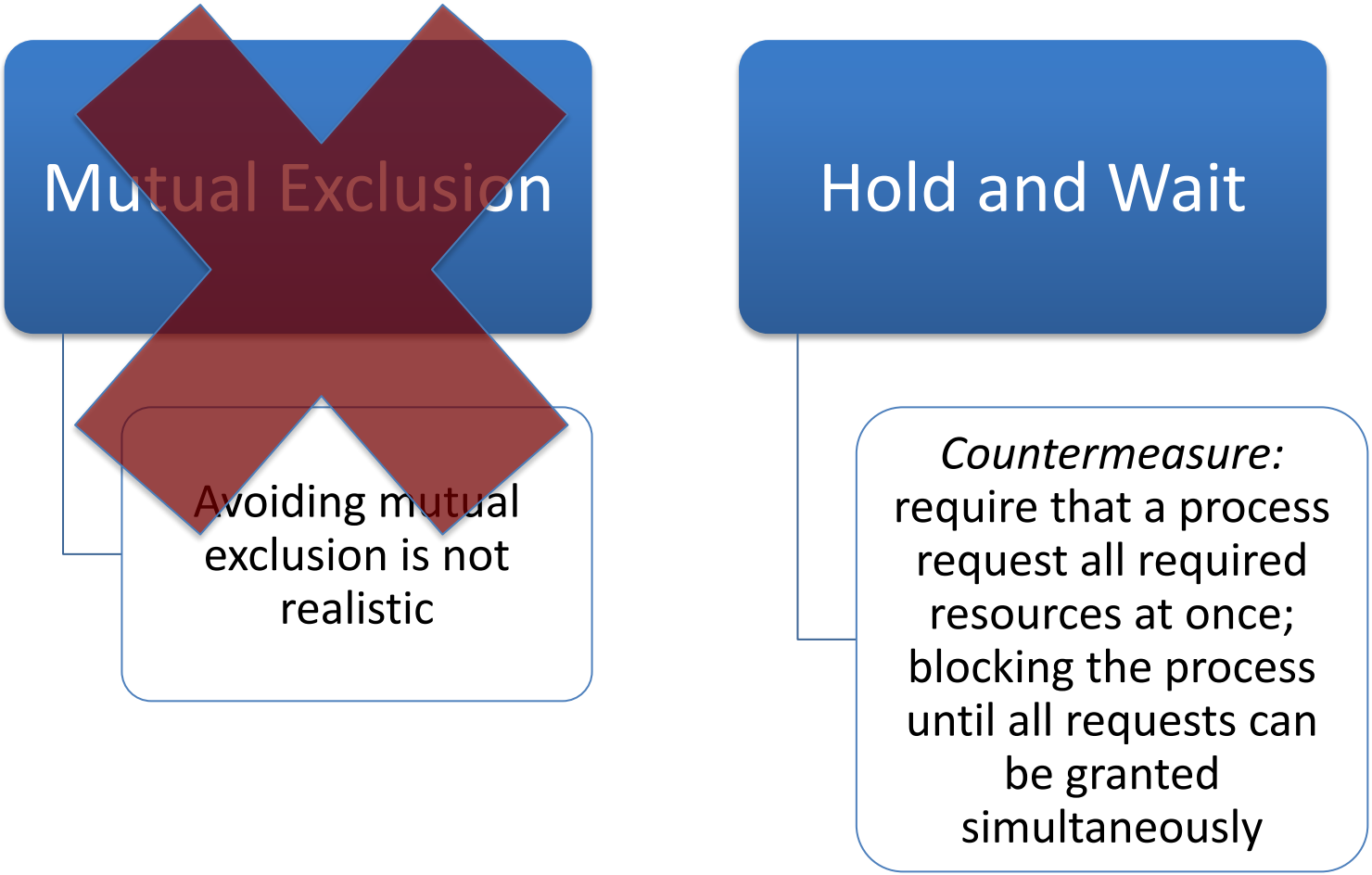
Avoid Deadlock

- make the appropriate dynamic choices based on the current state of resource allocation

Detect Deadlock

- attempt to detect the presence of deadlock and take action to recover

Deadlock Condition Prevention



Deadlock Condition Prevention

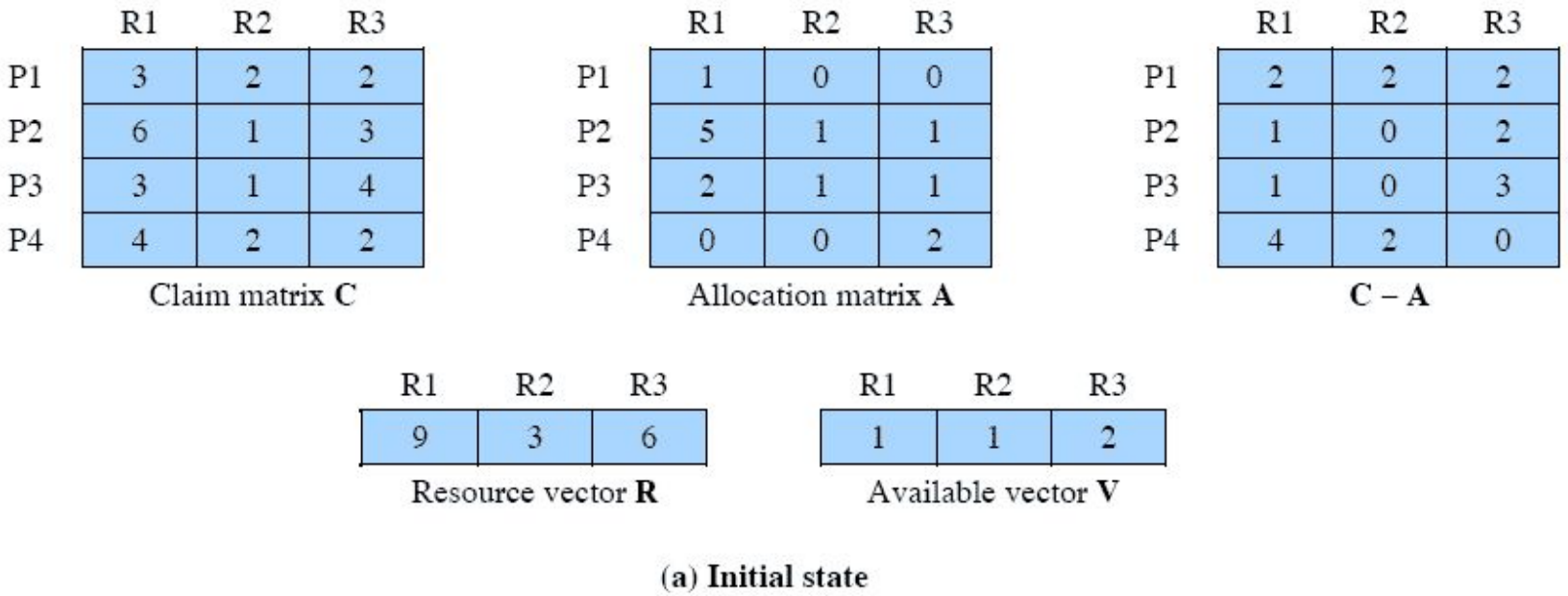
- No Preemption
  - *Countermeasure*: if a process holding certain resources is denied a further request, that process must release its original resources and request them again
- Circular Wait
  - *Countermeasure*: define a linear ordering of resource numbers; if a process has been allocated a resource of number *R* , then it may subsequently request only those resources of numbers following *R* in the ordering.
  - *Why does this work?*
    - *Think about the Resource Allocation Graph*

Deadlock Avoidance

- Deadlock prevention breaks one of the deadlock conditions through rules, which are defined before execution, while deadlock avoidance is enforced during execution
- A decision is made dynamically whether the current resource allocation request will lead to an unsafe state
- Requires knowledge of future process requests
- We will examine some examples

Example

- State of a system consisting of 4 processes and 3 resources
- Allocations have been made as follows



	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**

(a) Initial state

Determination of a Safe State

- P2 requests one of R1 and one unit of R3
- Should this request be granted?
- Banker's algorithm: assume this request is granted, then check whether the resulted state is safe
- A state is **safe** if there is at least one sequence of resource allocations that satisfies all the processes' needs

	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

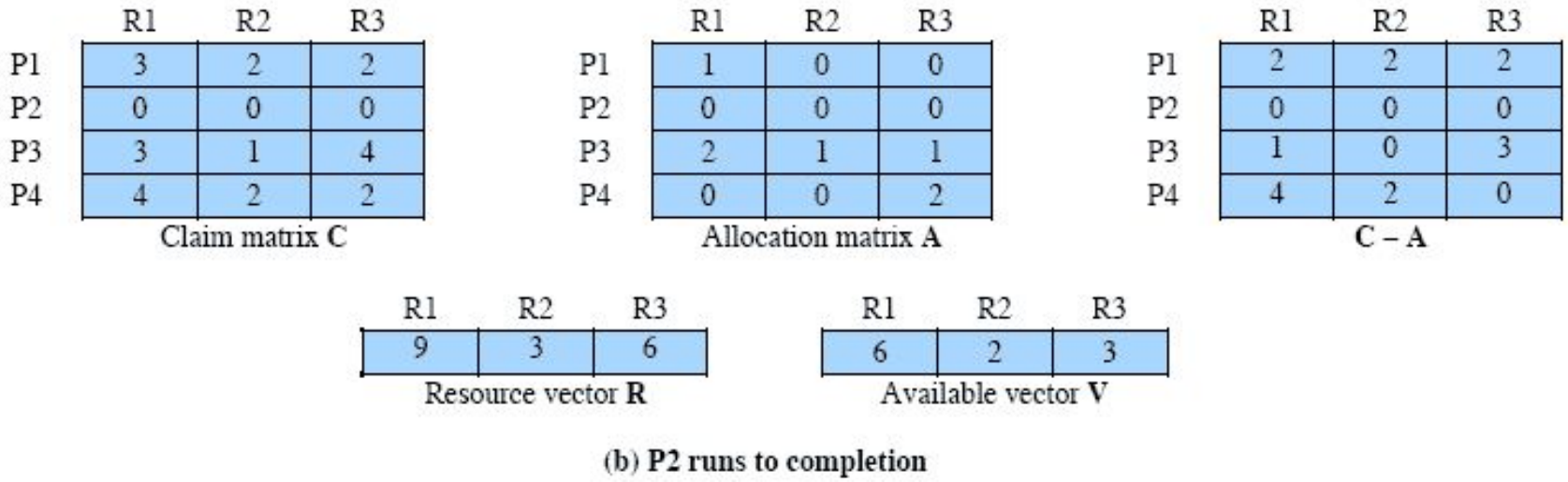
Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Is this a safe state?

P2 Runs to Completion



	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

Old Available vector (0, 1, 1) + Resources released by P2 (6, 1, 2) = Updated available vector(6, 2, 3)

P1 Runs to Completion

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

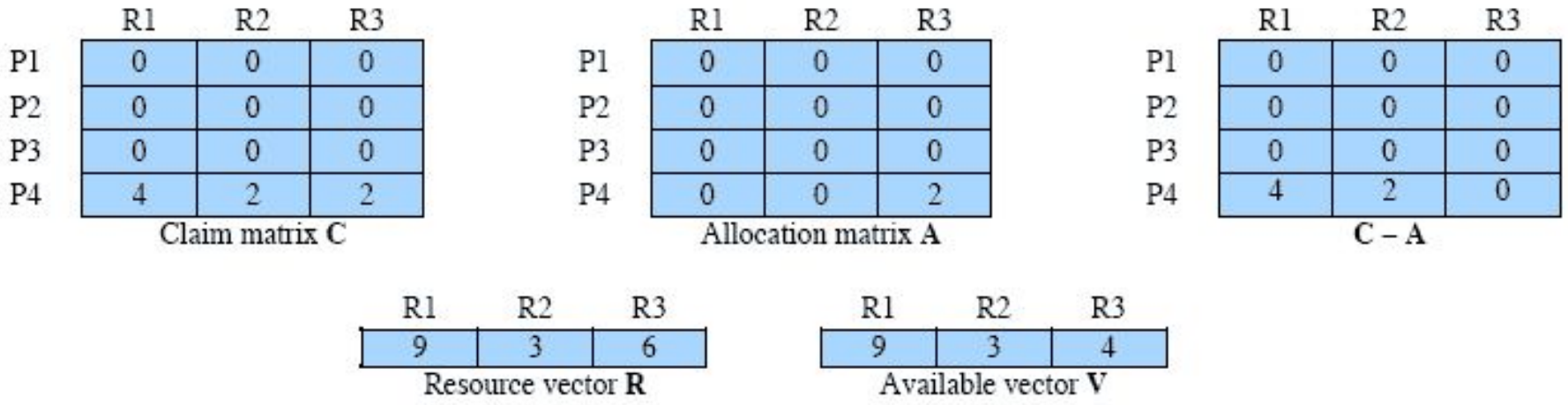
R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Old Available vector (6, 2, 3) + Resources Released by P1 (1, 0, 0) = Updated available vector(7, 2, 3)

P3 Runs to Completion



(d) P3 runs to completion

Thus, the state defined originally is safe

Determination of an Unsafe 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

P1 requests for one more R1 and one more R3

R1	R2	R3
9	3	6

Resource vector **R**

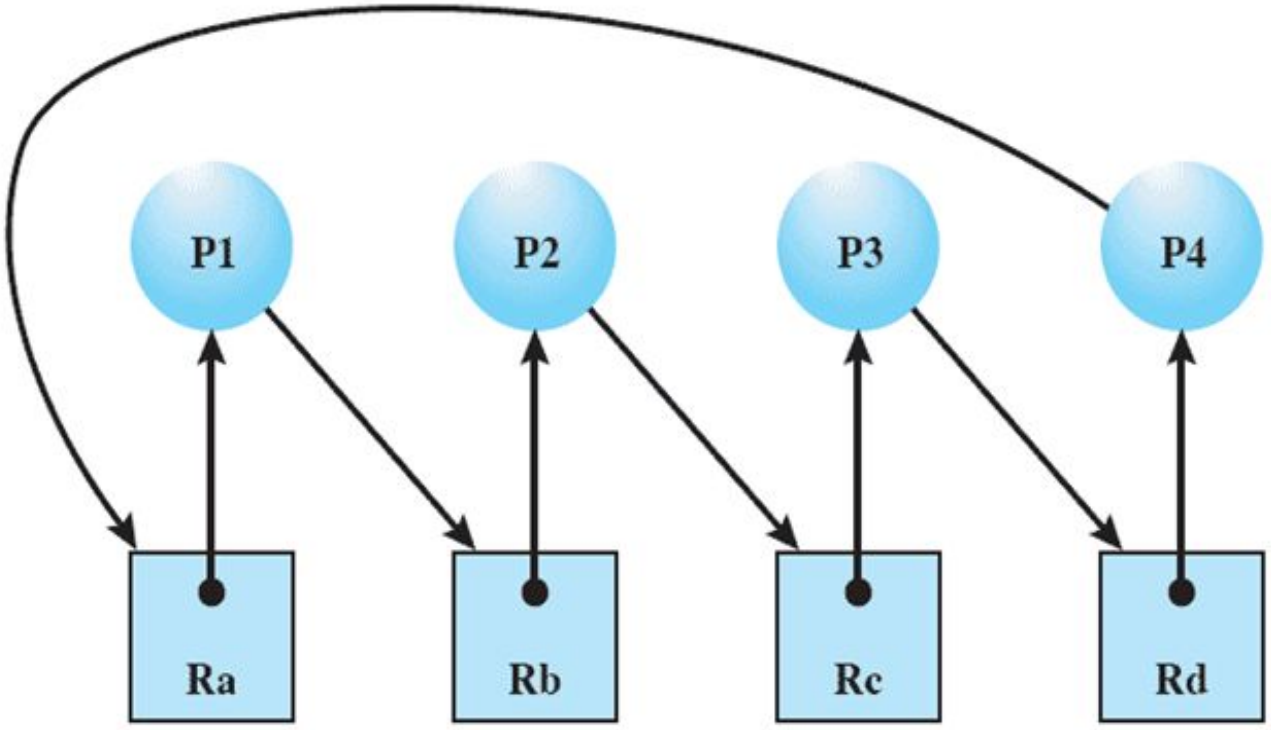
R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

The request should not be granted, because it leads to an unsafe state

Deadlock detection



Resource Allocation Graph

Recovery strategies

- Kill one deadlocked process at a time and release its resources
- Kill all deadlocked processes
- Steal one resource at a time
- Roll back all or one of the processes to a checkpoint that occurred before they requested any resources, then continue
  - Difficult to prevent indefinite postponement

Recovery by killing processes

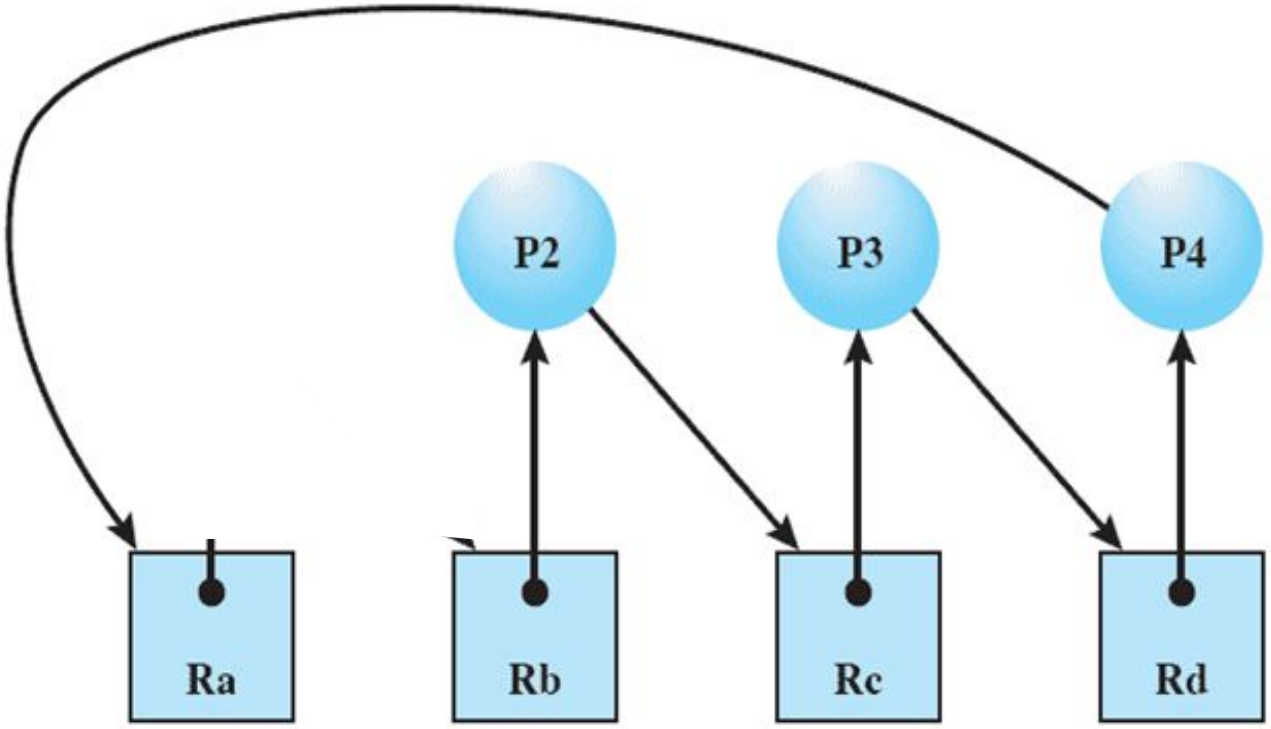
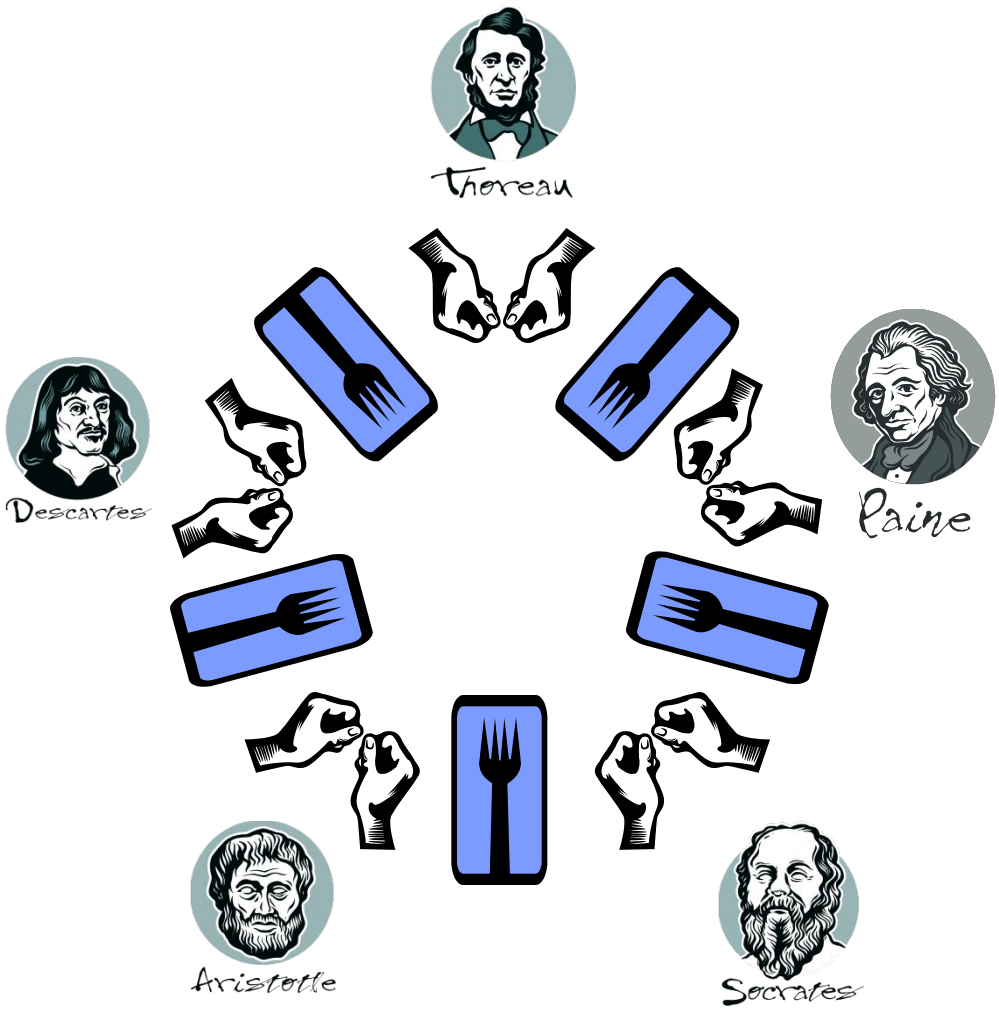


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

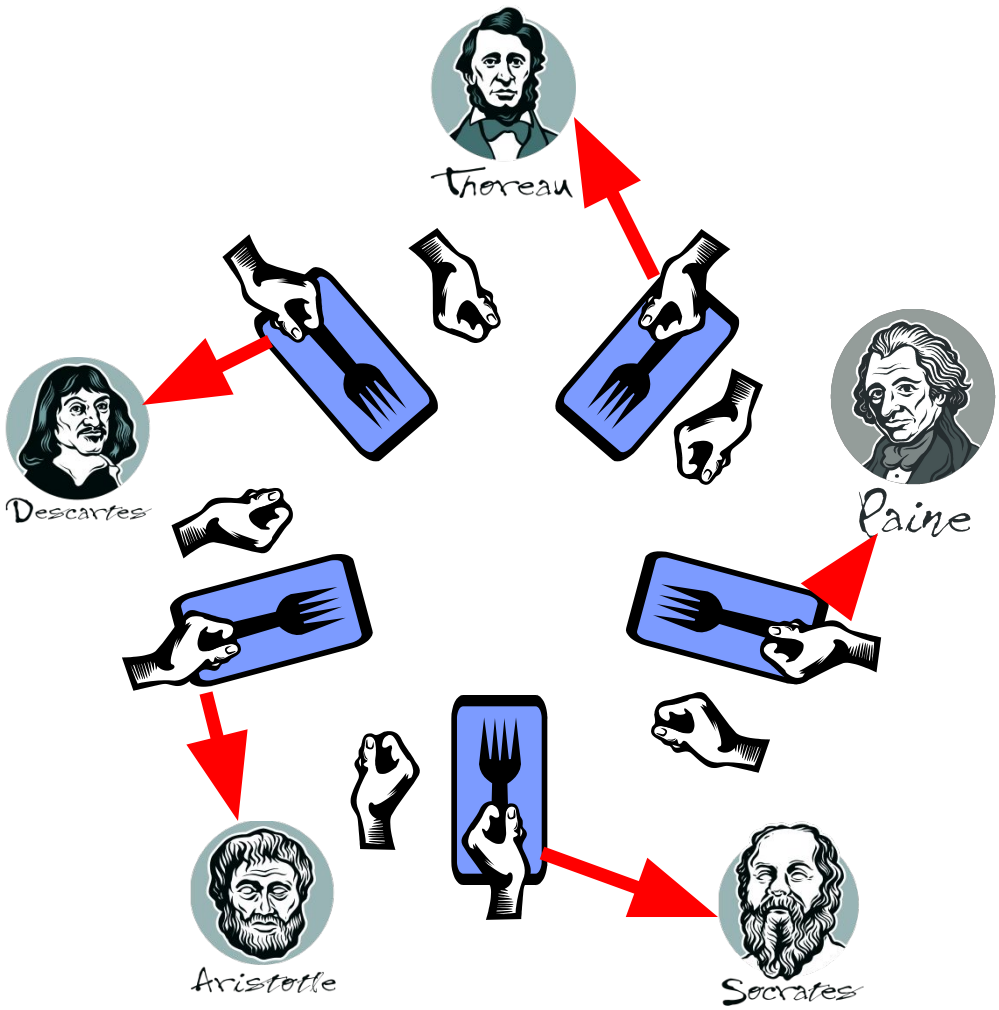
Dining Philosophers: failed solution with deadlock

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```



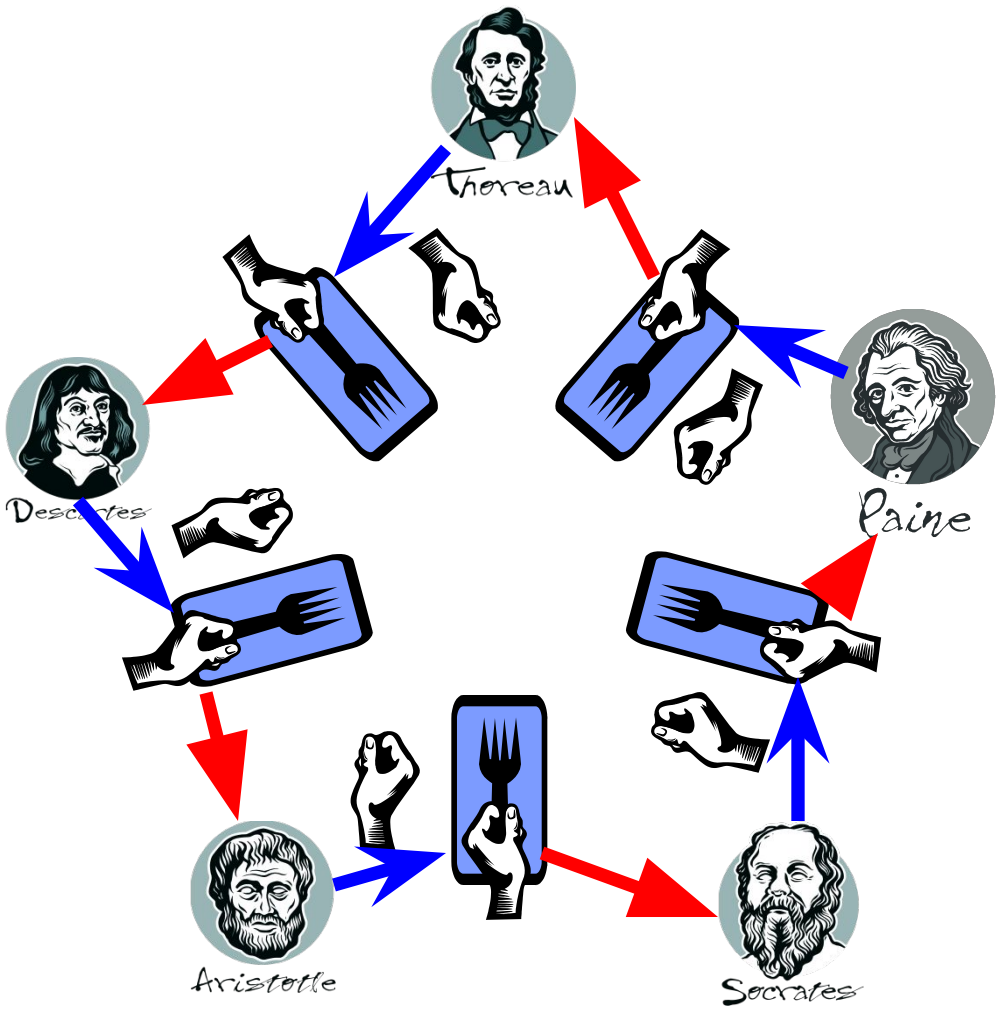
Dining Philosophers: failed solution with deadlock

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```



Dining Philosophers: failed solution with deadlock

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```

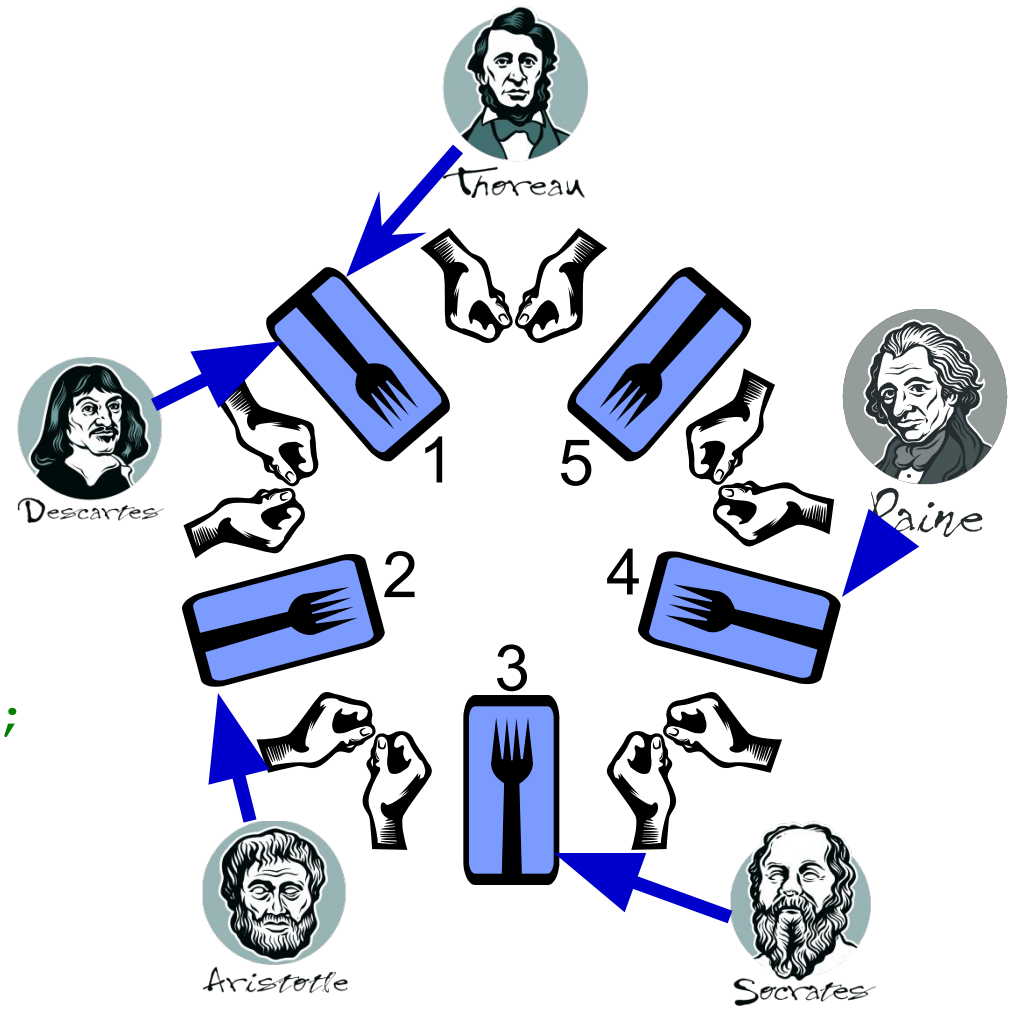


Dining Philosophers solution with numbered resources

Instead, number resources

First request lower numbered fork

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```

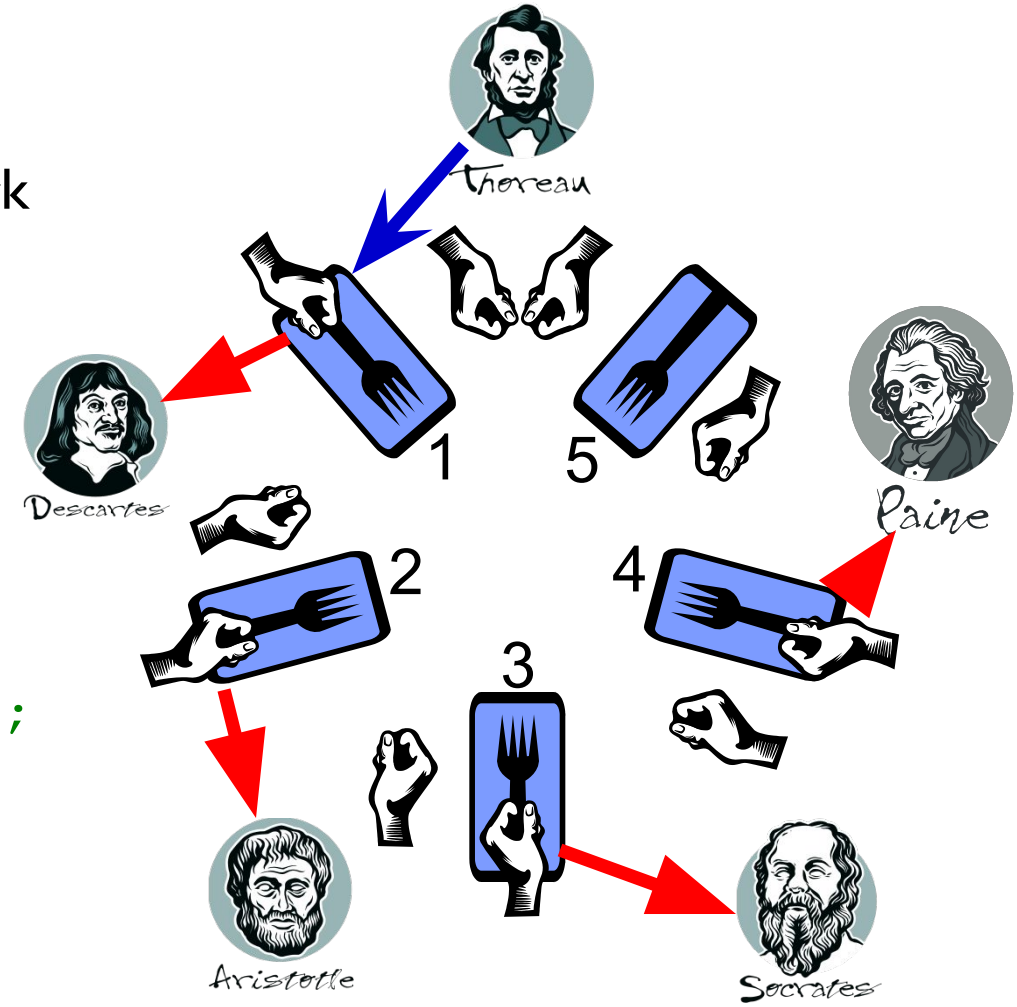


Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```

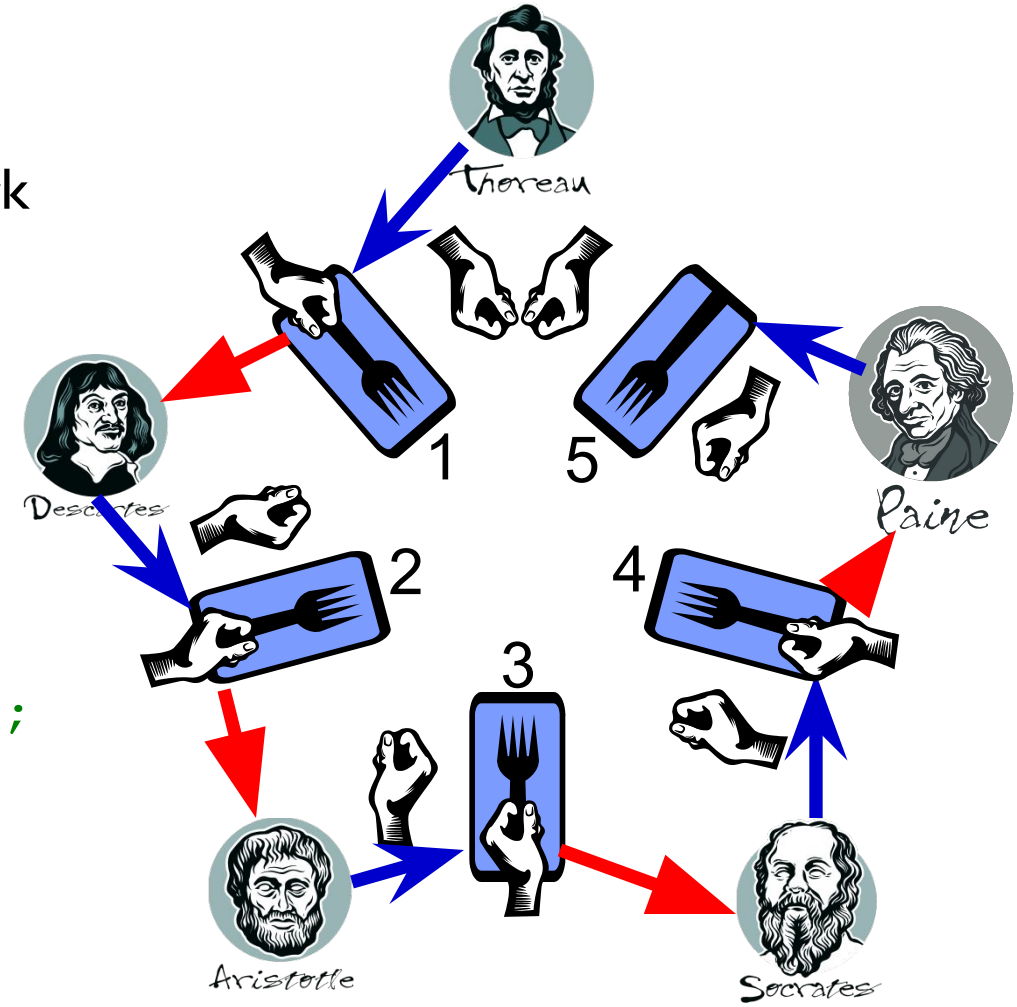


Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```

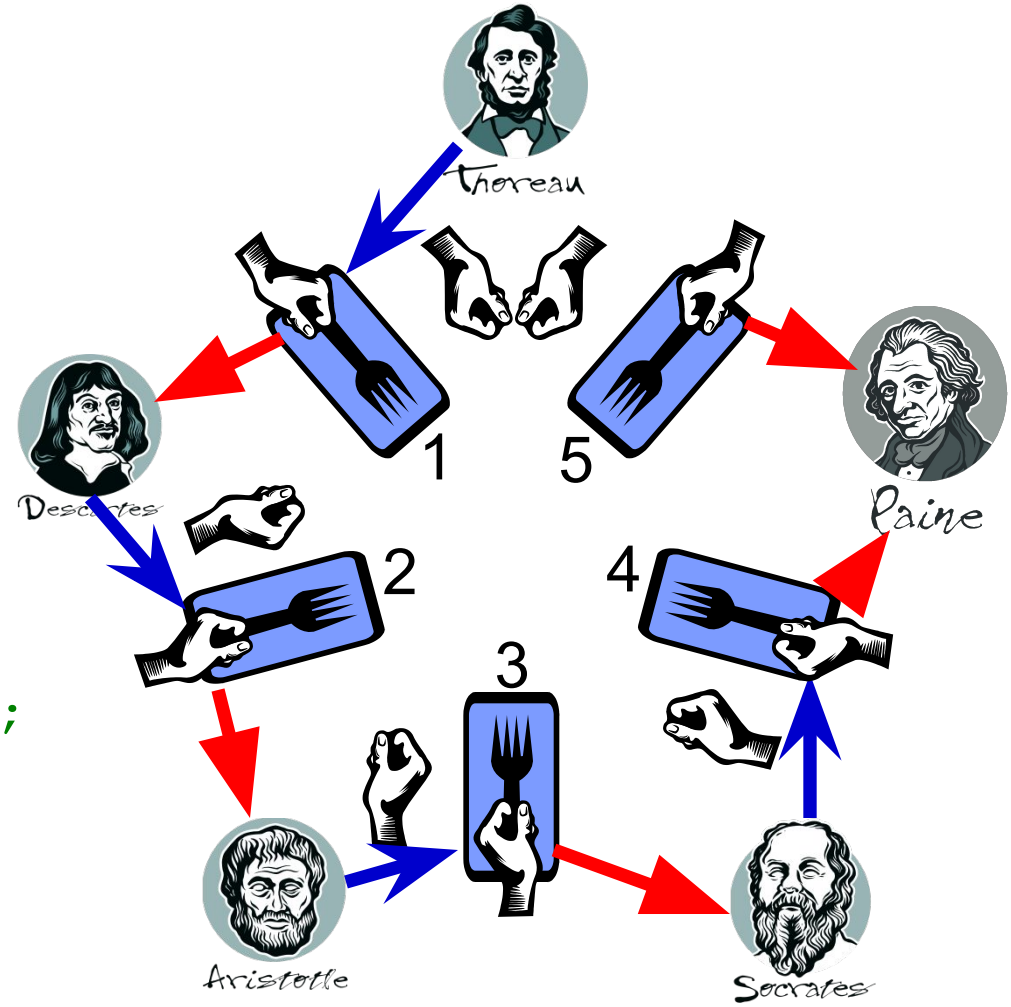


Dining Philosophers solution with numbered resources

Instead, number resources...

One philosopher can eat!

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```



Summary

- Uses of semaphores
- Deadlock
- Dealing with deadlock:
  - Prevention
  - Avoidance
  - Detection