CS343: Operating System

Monitors and Deadlock

Lect23 : 29th Sept 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

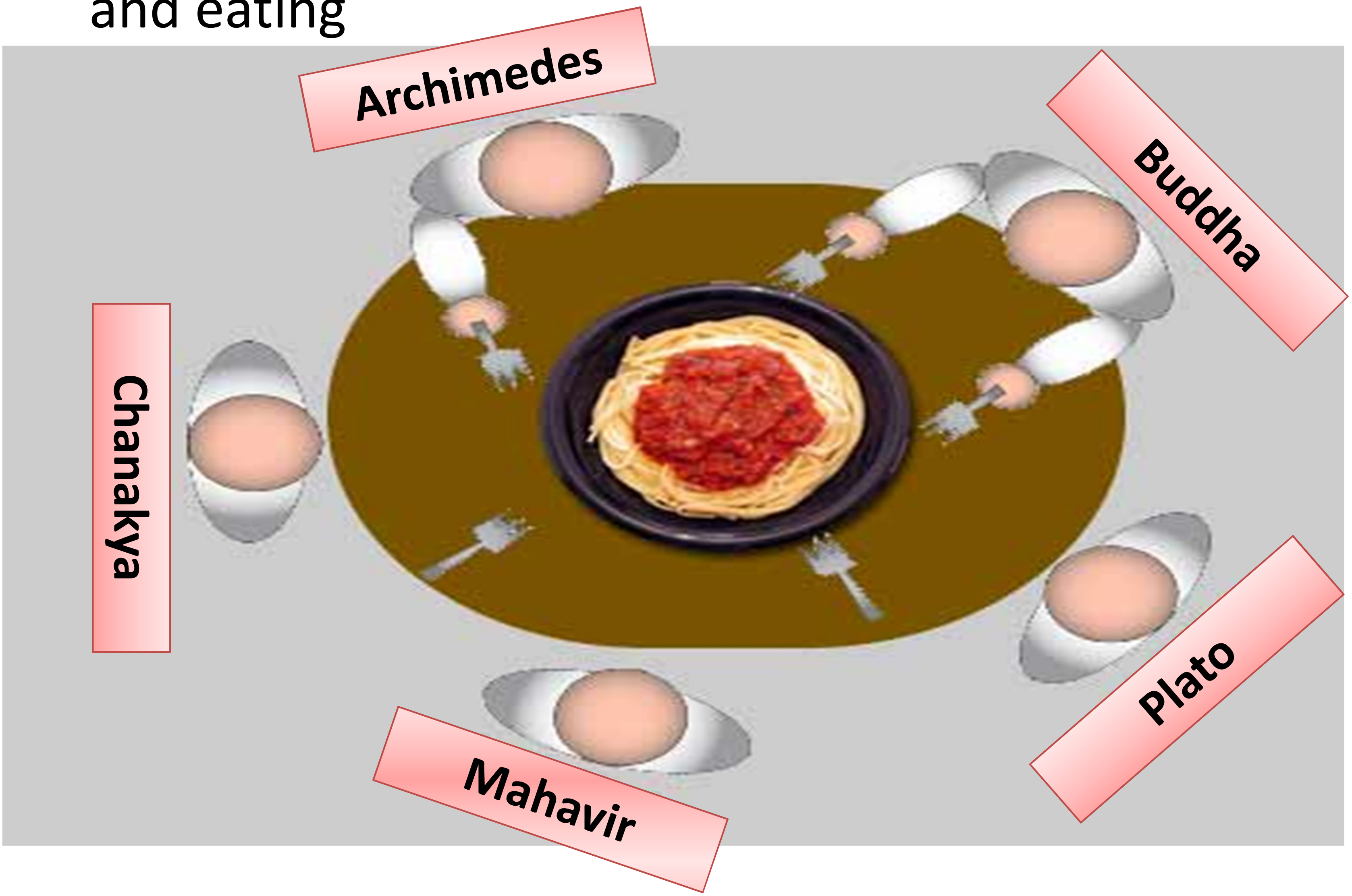
- Classical Sync Problems
  - **Dining-Philosophers Problem**
- Monitor
- Deadlock
  - Conditions (Why deadlock happens)
  - Prevention, Avoidance,
  - Detection, Recovery

Counting/Bin Semaphore

```
S=50; // Initialized to 1 for Binary
// S =0 Locked;  S>=1 Available
void synchronized  wait(S) {
            while (S <= 0) ; // busy wait
            S--;
}
void synchronized signal(S) {
                    S++;
}
```

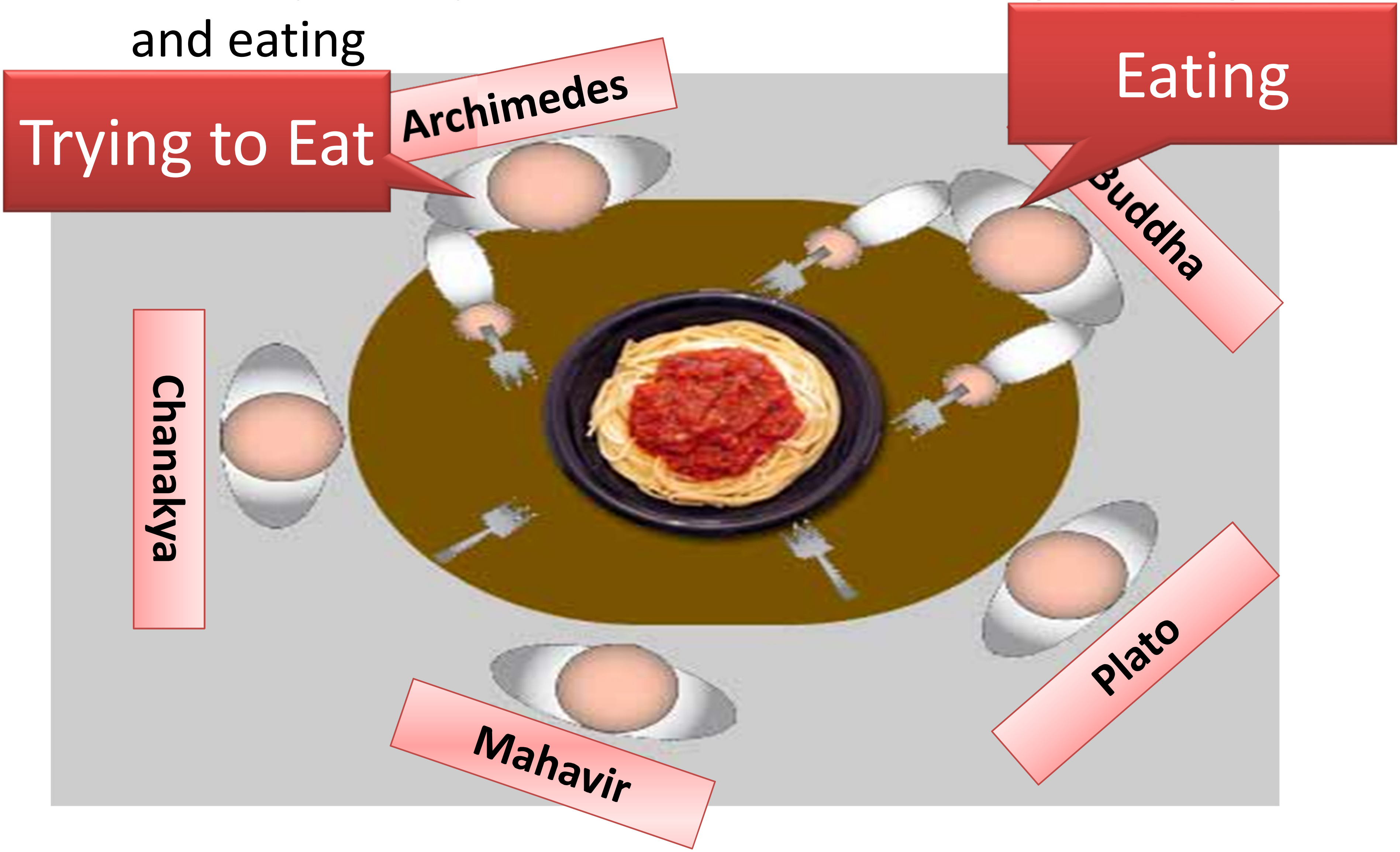
Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating



Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating



Dining-Philosophers Problem

- Philosophers spend their **lives alternating thinking and eating**
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - **Need both to eat, then release both when done**
- In the case of 5 philosophers
  - Shared data : Bowl of rice,
    - Semaphore **chopstick [5]** initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );
            //  eat
        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
            //  think
} while (TRUE);
```

- What is the problem with this algorithm?

Problem in Dining-Philosophers Algorithm

- May be deadlock
  - Every one is Holding one Fork and requesting for other
  - Form a circular wait

Problem in Dining-Philosophers Algorithm

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
    - 5 chop sticks, 4 people: pigeon-hole principle at least one can easily acquire 2 chop sticks , so no deadlock
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
    - Allow both or none, so only two person can get chance and there will not be any deadlock

Problem in Dining-Philosophers Algorithm

- Deadlock handling: Use an asymmetric solution
  - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick. **Left then Right**
  - Even-numbered philosopher picks up first the right chopstick and then the left chopstick. **Right then Left**
  - **As neighbor are different and circular fashion : They will allow you to pickup (if all try at same time)**

General Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

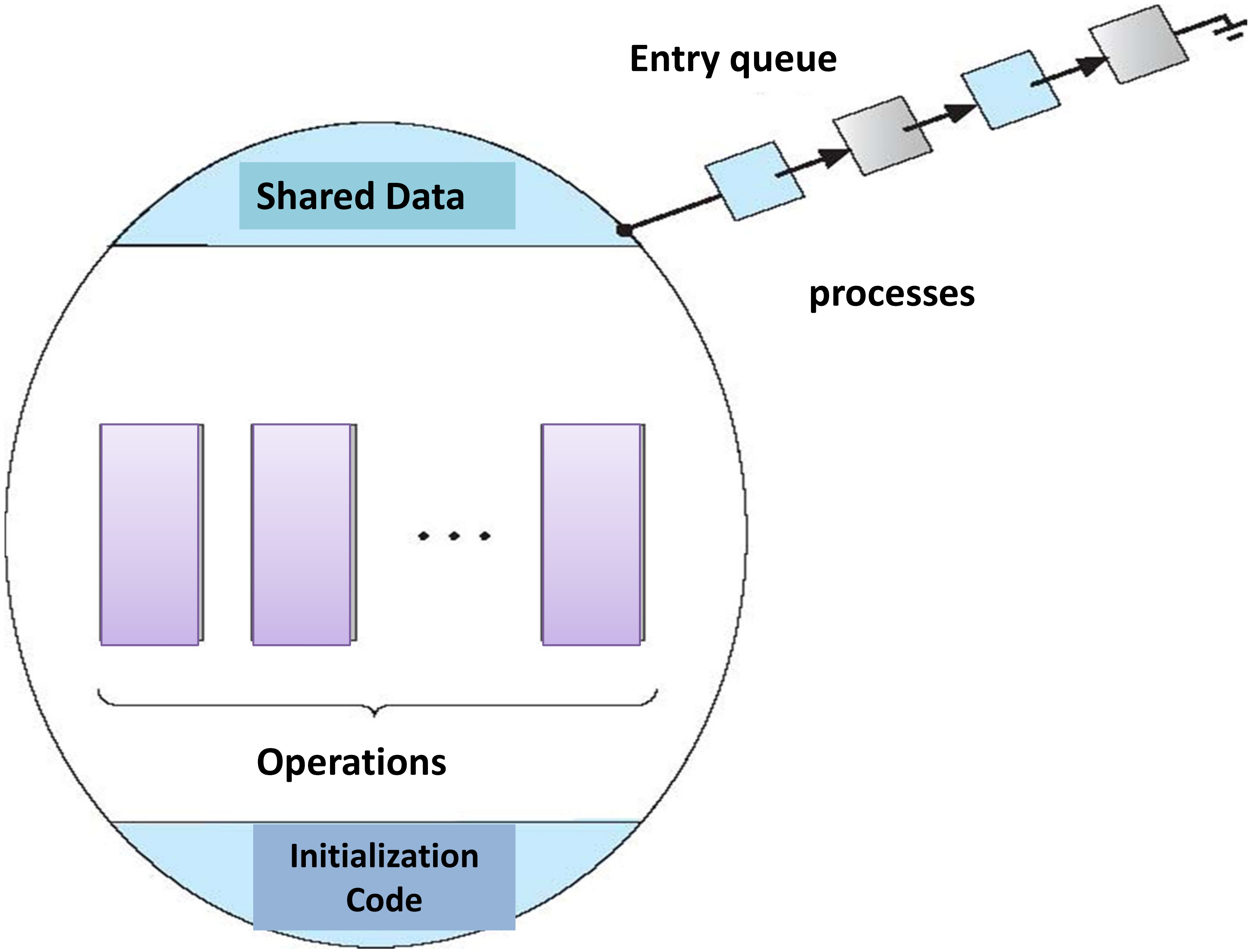
Monitors

- A high-level abstraction that provides
  - A convenient and effective mechanism for process synchronization
- *Monitor : Abstract data type*
  - internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - Remember ***synchronized*** function of java.util

Monitors

```
monitor monitor-name  {
        // shared variable declarations
        procedure P1 (…) { …. }
        procedure Pn (…) {……}

        Initialization code (…) { … }
        }
}
```

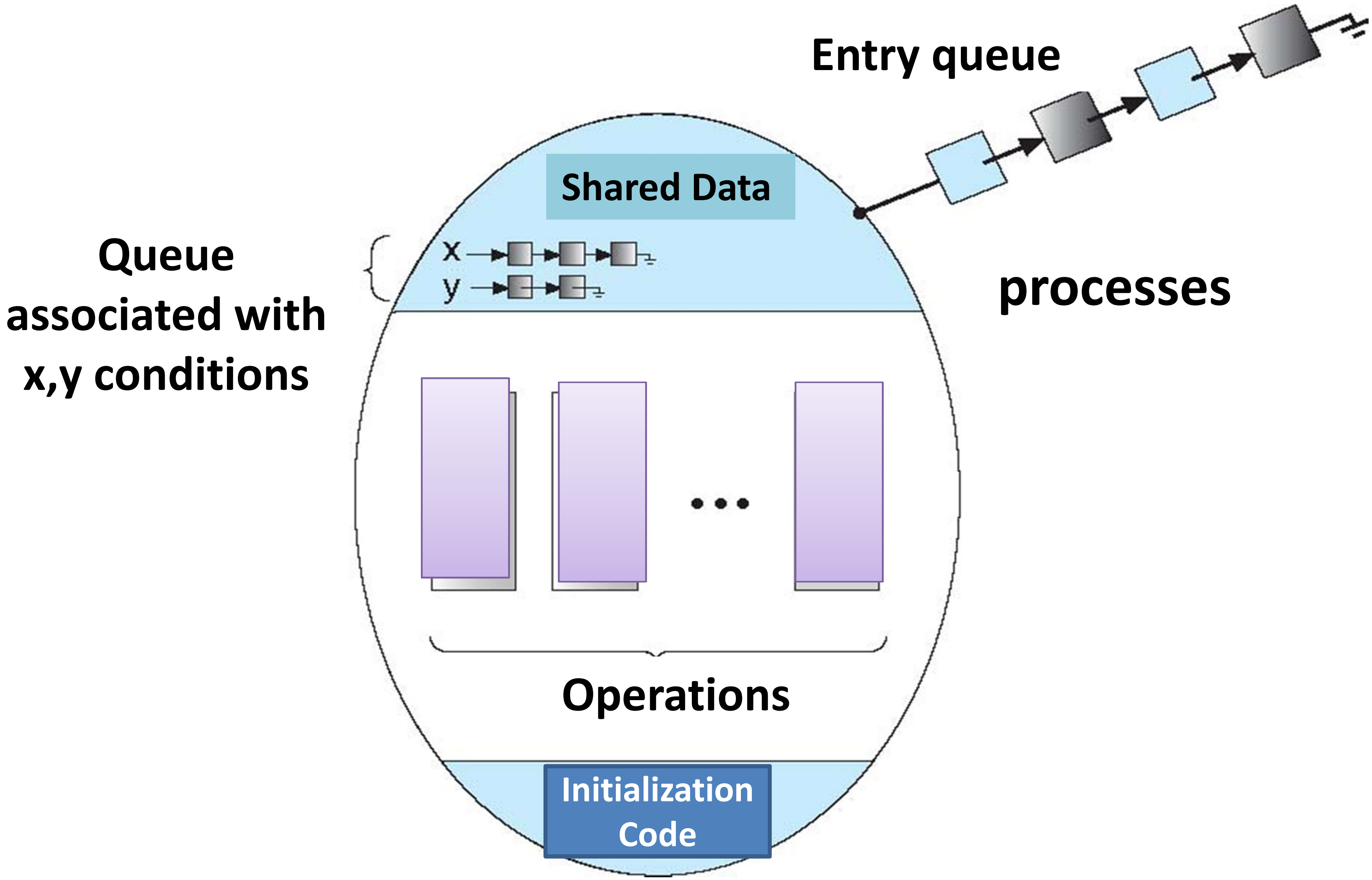
Schematic view of a Monitor



Condition Variables

- **Condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - If no **x.wait()** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
  - Execution Sequence
    - P.signal() Q P or
    - P.signal() P Q

Condition Variables Choices

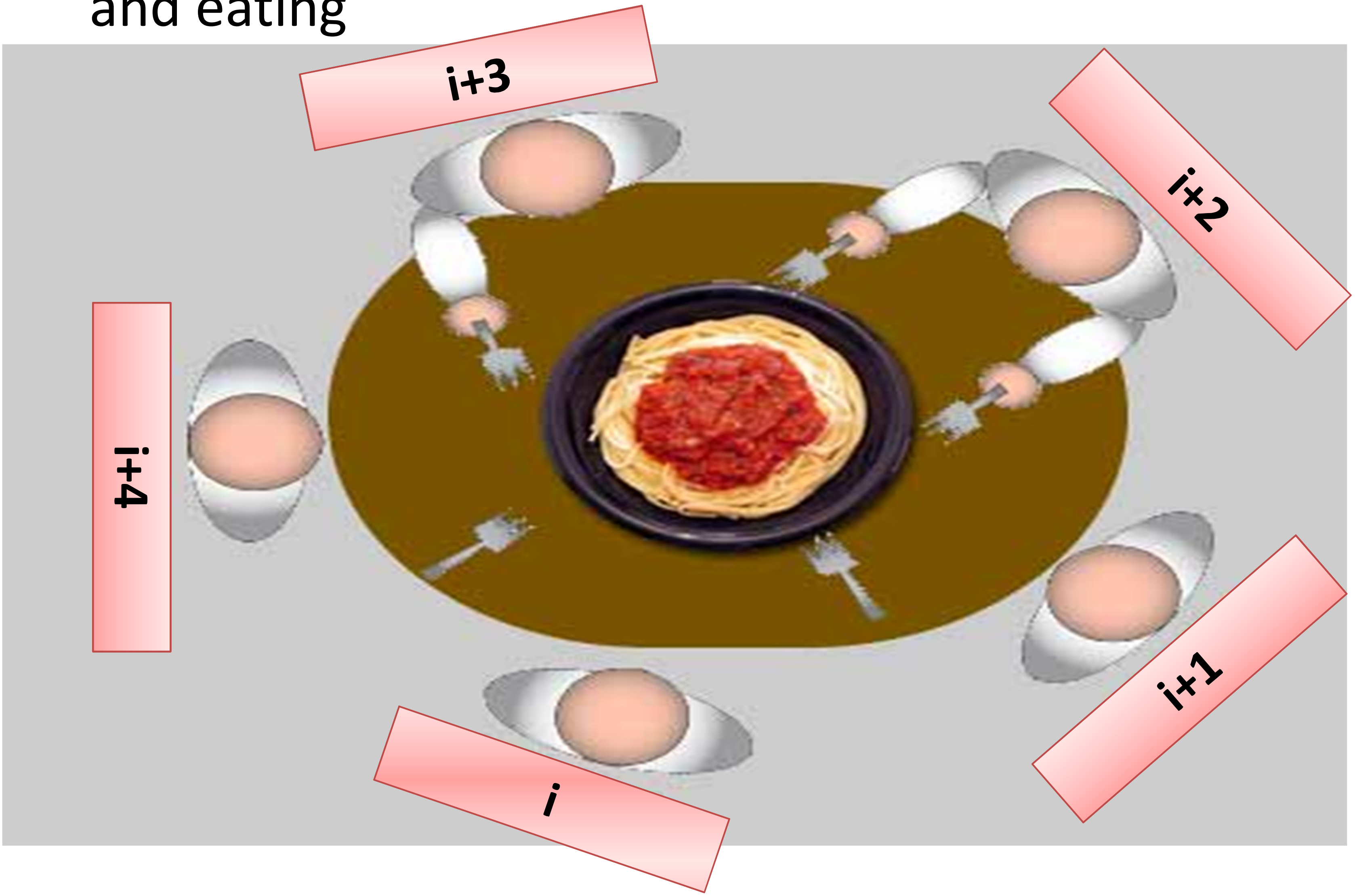
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Implemented including **C#** and **Java**
    - **Java: wait(), notify(), notifyall()**

Monitor Solution to Dining Philosophers

```
class Monitor_DiningPhilosophers  {
   enum { THINKING; HUNGRY, EATING} state [5] ;
   condition   self [5];
   void synchronized pickup (int i) {
        state[i] = HUNGRY;   test(i);
        if (state[i] != EATING) self[i].wait();
   }     //end pickup
   void synchronized  putdown (int i) {
        state[i] = THINKING;
              // test left and right neighbors
         test((i + 4) % 5);    test((i + 1) % 5);
   } //end putdown
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating



Solution to Dining Philosophers (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                    state[i] = EATING ;
                    self[i].signal () ;
                    }
          }  //end test
      void  initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
         } //end init
} // end Monitor
```

Solution to Dining Philosophers (Cont.)

- Each philosopher ***i*** invokes the operations **pickup()** and **putdown()** in the following sequence:

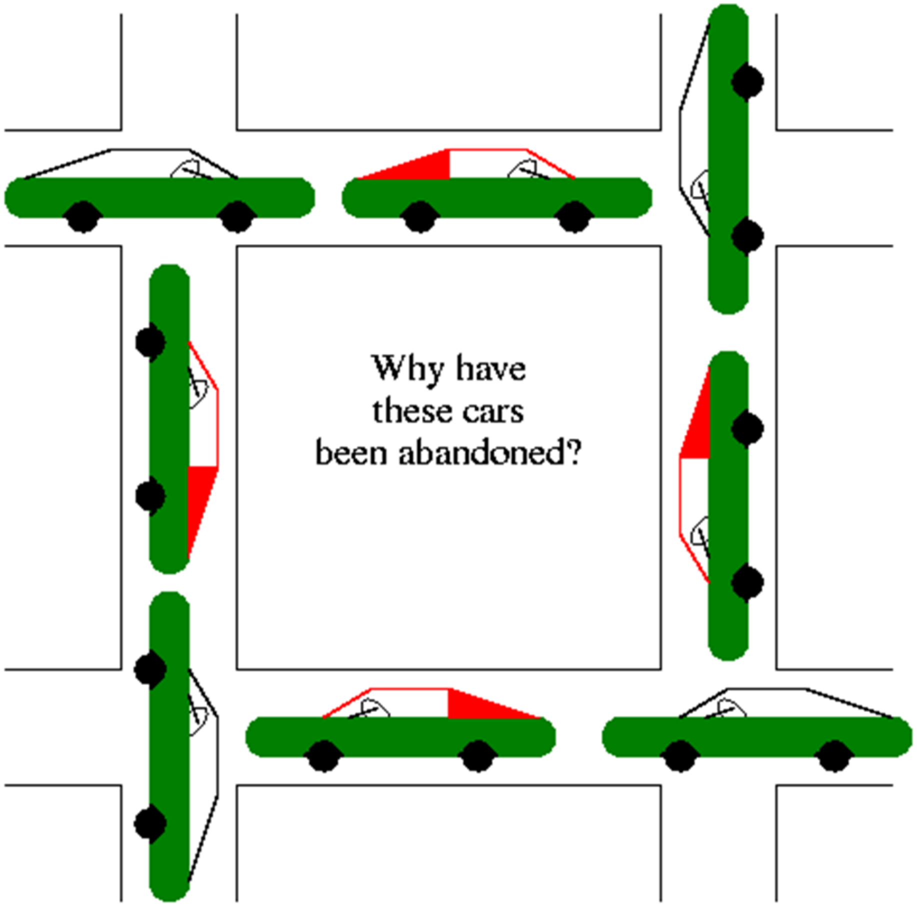
```
DiningPhilosophers.pickup(i);

    EAT();

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Deadlock

Deadlock

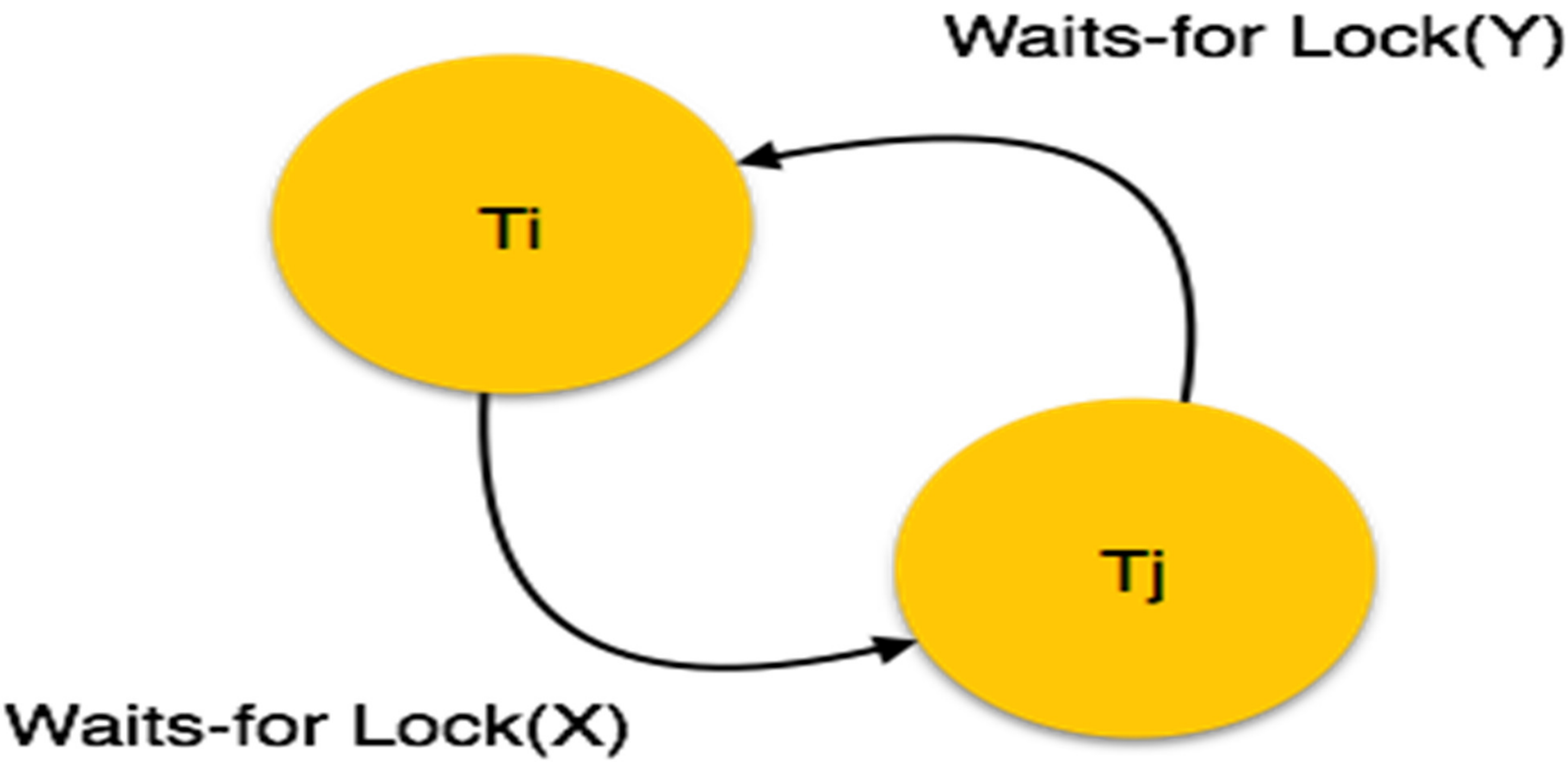
- Mutual exclusion
  - wider road
- Hold and wait
  - One voluntarily go back
- No preemption
  - Police took out one car forcibly
- Circular wait
  - One car is going back direction



Deadlock

Deadlock

Deadlock



System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

    CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
    - **Request** **// Similar to Lock**
    - **Use** **// Similar to CS**
    - **Release** **// Similar to Unlock**

Deadlock Characterization

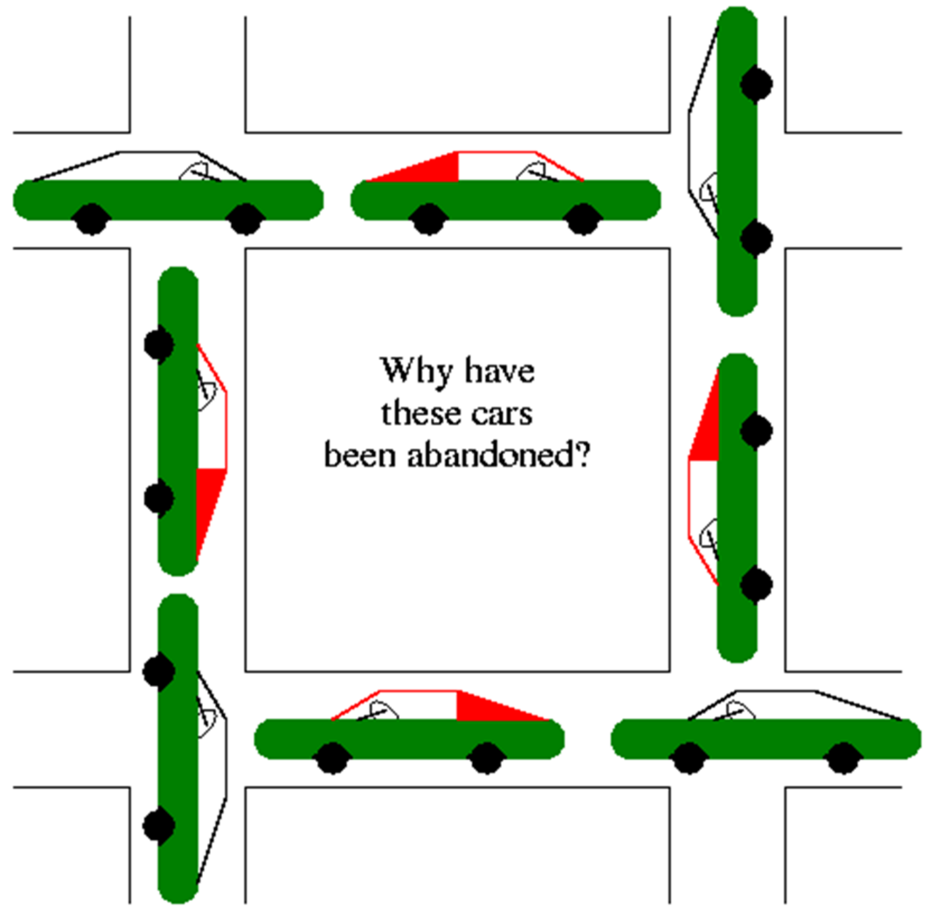
- Deadlock can arise if four conditions hold simultaneously.
- **Mutual exclusion**
  - Only one process at a time can use a resource
- **Hold and wait**
  - A process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**
- **Circular wait**

Deadlock Characterization

- **Mutual exclusion, Hold and wait**
- **No preemption**
  - A resource can be released only voluntarily by the process holding it, after that process has completed its task
- **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.

Deadlock

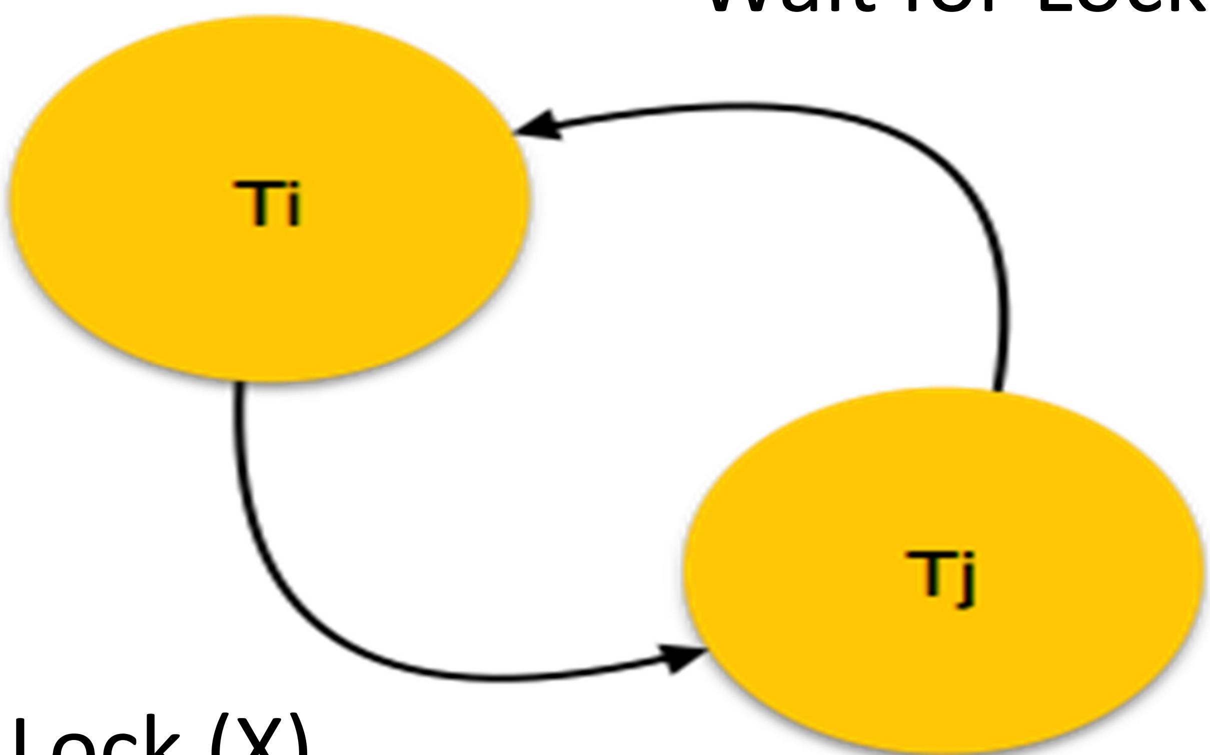
- Mutual exclusion
  - wider road
- Hold and wait
  - One voluntarily go back
- No preemption
  - Police took out one car forcibly
- Circular wait
  - One car is going back direction



Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.

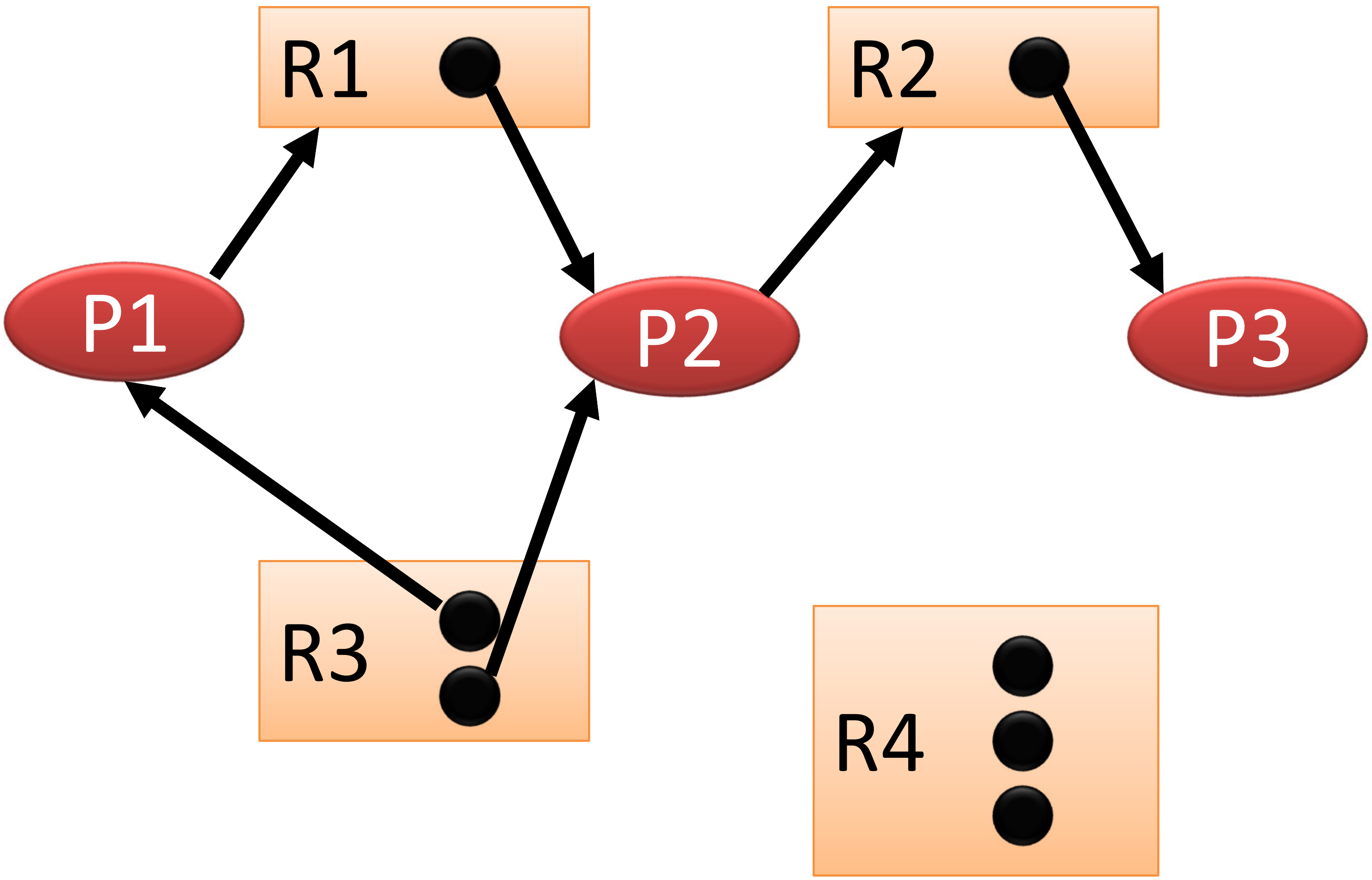
Wait for Lock (Y)



Wait for Lock (X)

Resource Allocation Graph (RAG)

RAG to Characterize Deadlock



Resource-Allocation Graph

A set of vertices V and a set of edges E.

- 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
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge

$$R_j \rightarrow P_i$$

Thanks