Synchronization (II)

Operating Systems

Yu-Ting Wu

(with slides borrowed from Prof. Jerry Chou)

Outline

- Classical synchronization problems
- Monitor
- Atomic transactions

Classical Synchronization Problems

Classical Synchronization Problems

- **Purpose**
  - Used for testing newly proposed synchronization scheme

- **Bounded-Buffer (Producer-Consumer) Problem**
- **Reader-Writers Problem**
- **Dining-Philosopher Problem**

Bounded-Buffer Problem

- A pool of *n* buffers, each capable of holding one item

- **Producer**
  - Grab an empty buffer
  - Place an item into the buffer
  - Waits if no empty buffer is available

- **Consumer**
  - Grab a buffer and retracts the item
  - Place the buffer back to the free pool
  - Waits if all buffers are empty

Readers-Writers Problem

- A set of shared data objects
- A group of processes
  - Reader processes (read shared objects)
  - Writer processes (update shared objects)
  - A writer process has **exclusive access** to a shared object
- Different variations involving priority
  - **First RW problem:** **no** reader will be kept waiting **unless a writer is updating** a shared object
  - **Second RW problem:** once a writer is ready, it performs the updates **as soon as** the shared objects is released
    - Writer has higher priority than reader
    - Once a writer is ready, no new reader can start reading

First Reader-Writer Algorithm

```
// mutual exclusion for write
semaphore wrt = 1
// mutual exclusion for readcount
semaphore mutex = 1
int readcount = 0;

Writer () {
    while (true) {
        wait (wrt);

        // Writer code.

        signal (wrt);
    }
}
```

```
Reader () {
    while (true) {
        wait (mutex);
            readcount++;
            if (readcount == 1)
                wait(wrt);
        signal(mutex);

            // Reader code.

        wait (mutex);
            readcount--;
            if (readcount == 0)
                signal(wrt);
        signal (mutex);
    }
}
```

acquire write lock → `if (readcount == 1) wait(wrt);`

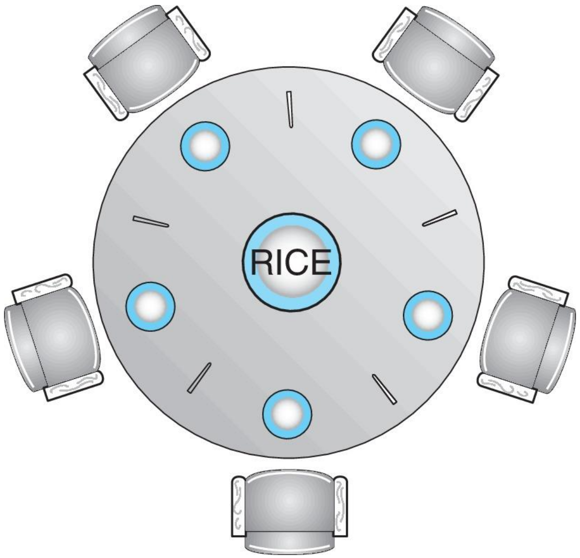
release write lock **if no more reads** → `if (readcount == 0) signal(wrt);`

Readers share a single ***wrt*** lock

Writer may have starvation problem

Dining-Philosophers Problem

- **5 persons** sitting on 5 chairs with **5 chopsticks**
- A person is either thinking or eating
  - **thinking**: no interaction with the rest 4 persons
  - **eating**: need 2 chopsticks at hand
  - a person **picks up 1 chopstick at a time**
  - done eating: **put down both chopsticks**



Monitors

Motivation

- Although semaphores provide a convenient and effective synchronization mechanism, **its correctness is depending on the programmer**
    - All processes access a shared data object must execute **wait()** and **signal()** **in the right order and right place**
    - This may not be true because honest programming error or uncooperative programmer

Monitor

- A **high-level language** construct
- The representation of a monitor type consists of
  - Declaration of **variables** whose values define the state of an instance of the type
  - **Procedures/functions** that implement operations on the type
- The monitor type is similar to a **class in O.O language**
  - A procedure within a monitor can access only **local variable** and the **formal parameters**
  - The local variables of a monitor can be used only by the local procedures
- But, the monitor ensures that **only one process at a time can be active within the monitor**

Monitor (cont.)

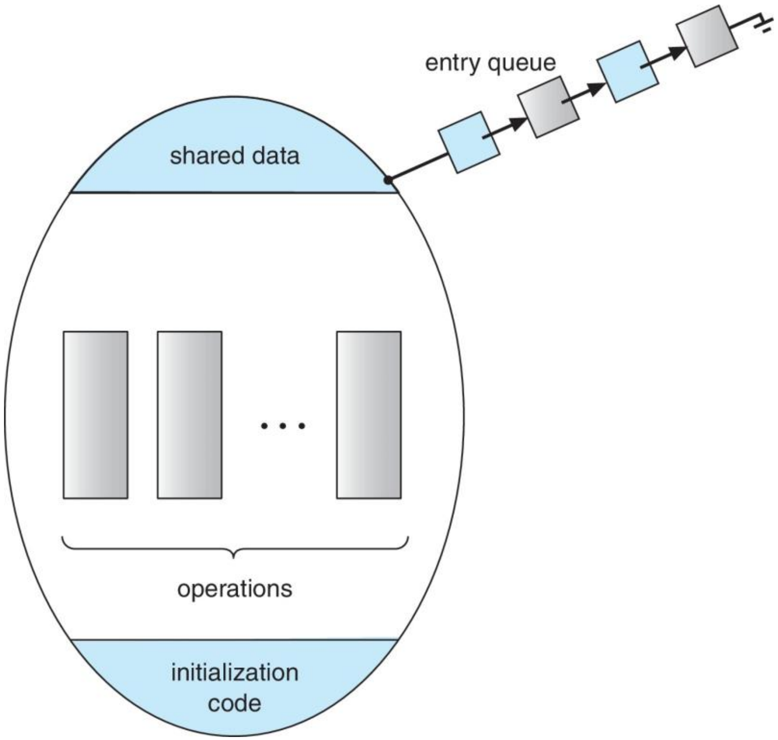
- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent process

Syntax

monitor monitor-name
{
/ shared variable declarations */*
…

function P_1 (…) { … }
function P_2 (…) { … }
…
function P_n (…) { … }

initialization code { … }
}



Monitor Condition Variables

- To allow a process to **wait within** the monitor, a condition variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait()** and **signal()**

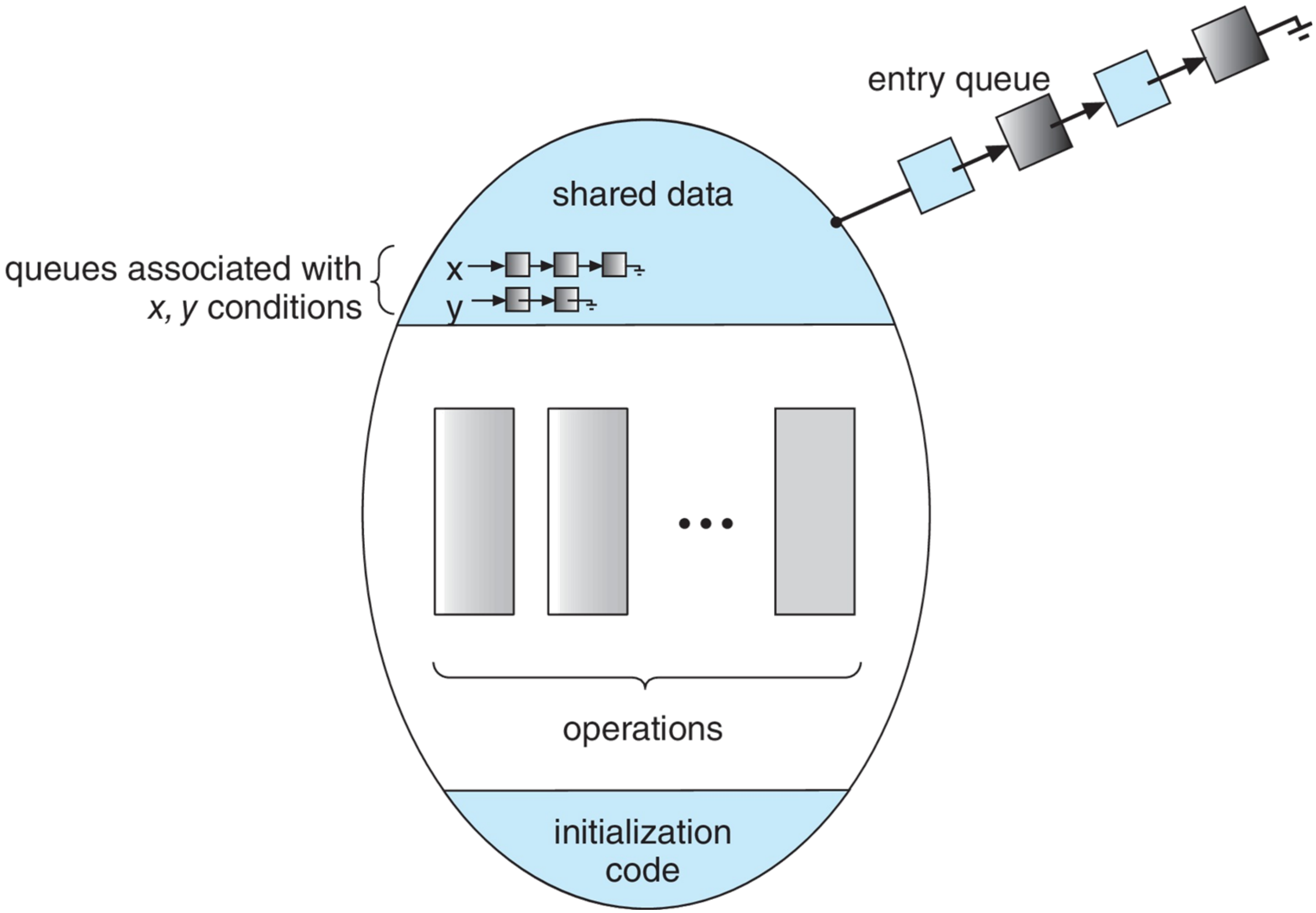
  x.wait();

  means that the process invoking this operation is suspended until another process invokes it

  x.signal();

  resumes exactly one suspended process. If no suspended, then the signal operation **has no effects (in contrast, signal always change the state of a semaphore)**

Monitor Condition Variables



Dining Philosophers Example

```
monitor dp {
    enum { thinking, hungry, eating } state[5];  // current state
    condition self[5];        // delay eating if can't obtain chopsticks

    void pickup(int i);       // pickup chopsticks
    void putdown(int i);      // putdown chopsticks
    void test(int i);         // try to eat

    void init() {
        for (int i = 0 ; i < 5 ;  i++)
            state[i] = thinking;
    }
}
```

Dining Philosophers Example (cont.)

```
void pickup (int i) {
    state[i] = hungry;
    test(i);        // try to eat
    if (state[i] != eating)
        self[i].wait();    // wait to eat
}
```

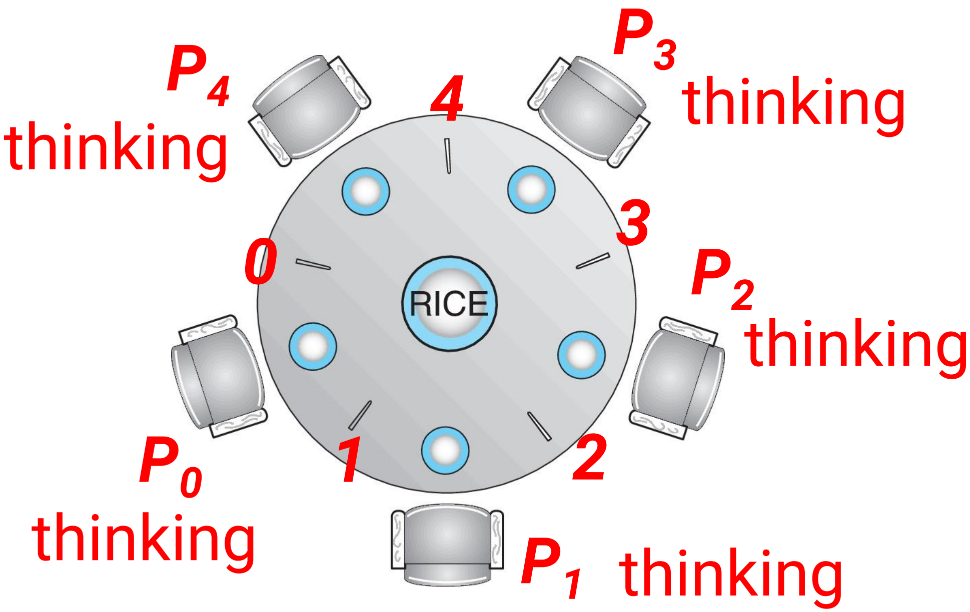
```
void putdown (int i) {
    state[i] = thinking;
    // check if neighbors are
    // waiting to eat
    test ((i+4) % 5);
    test ((i+1) % 5);
}
```

```
// try to let Pi eat (if it is hungry)
void test (int i) {
    if ( (state[ (i+4) % 5 != eating) && (state[ (i+1) % 5] != eating &&
        (state[i] == hungry) ) {
            // no neighbors are eating and Pi is hungry
            state[i] = eating;
            self[i].signal();
    }
}
```

If P_i is suspended, resume it

Dining Philosophers Example (cont.)

- An illustration

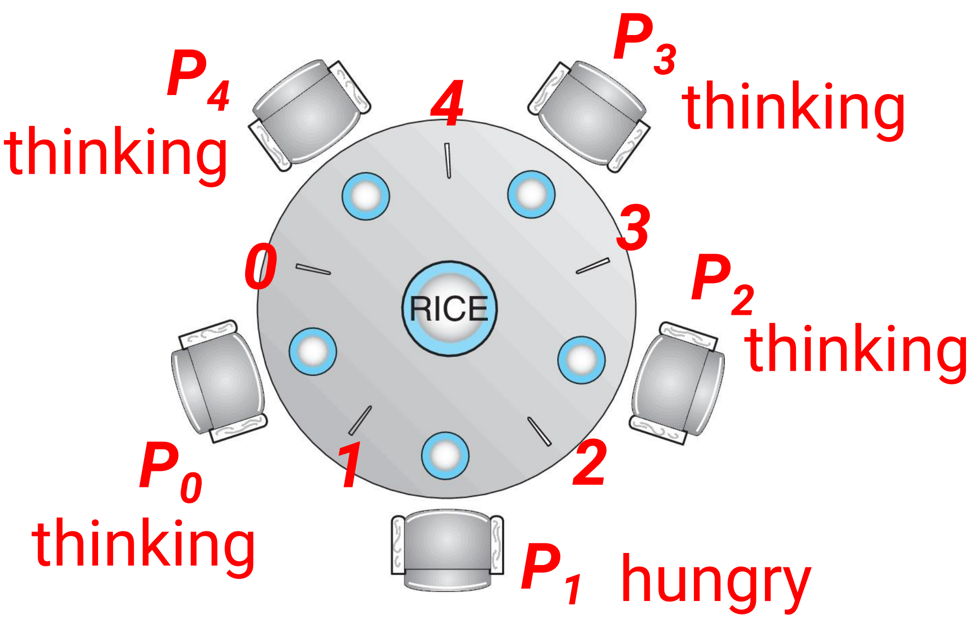


P1:
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

Dining Philosophers Example (cont.)

- An illustration




```
void pickup (int i) {
      state[i] = hungry;
      test(i);      // try to eat
      if (state[i] != eating)
            // wait to eat
            self[i].wait();
}
```

P1:

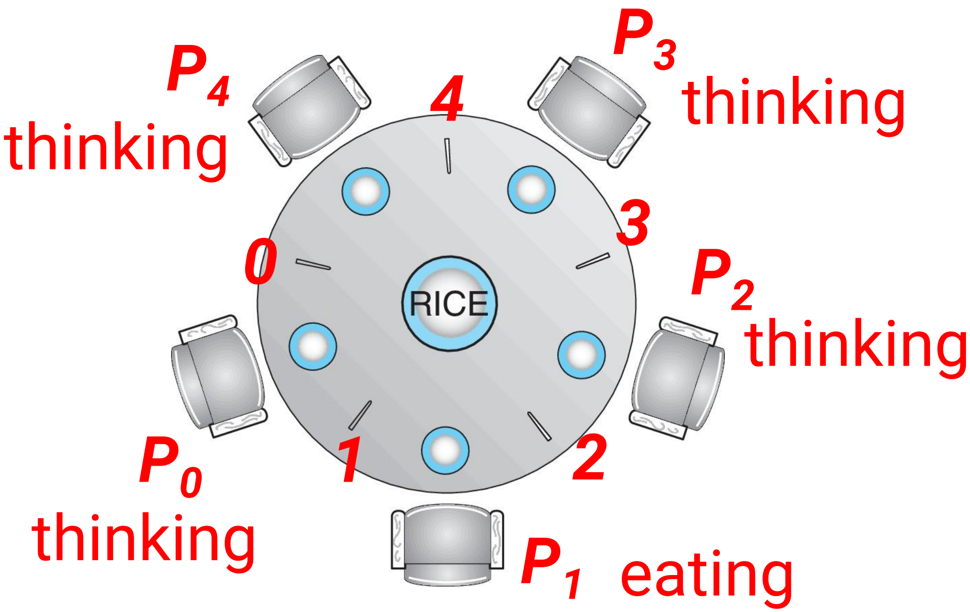
DiningPhilosophers.pickup(1)
    eat
DiningPhilosophers.putdown(1)

P2:

DiningPhilosophers.pickup(2)
    eat
DiningPhilosophers.putdown(2)

Dining Philosophers Example (cont.)

- An illustration



P_1

```
void pickup (int i) {
        state[i] = hungry;
        test(i);    // try to eat
        if (state[i] != eating)
                // wait to eat
                self[i].wait();
}
```

P1:

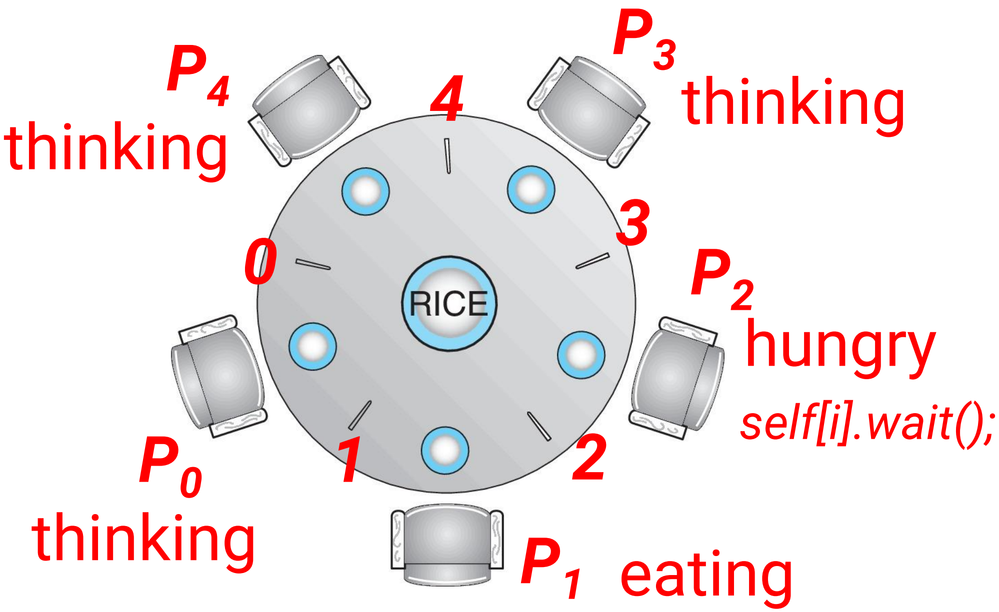
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:

DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

Dining Philosophers Example (cont.)

- An illustration




```
void pickup (int i) {
        state[i] = hungry;
        test(i);    // try to eat
        if (state[i] != eating)
              // wait to eat
              self[i].wait();
}
```

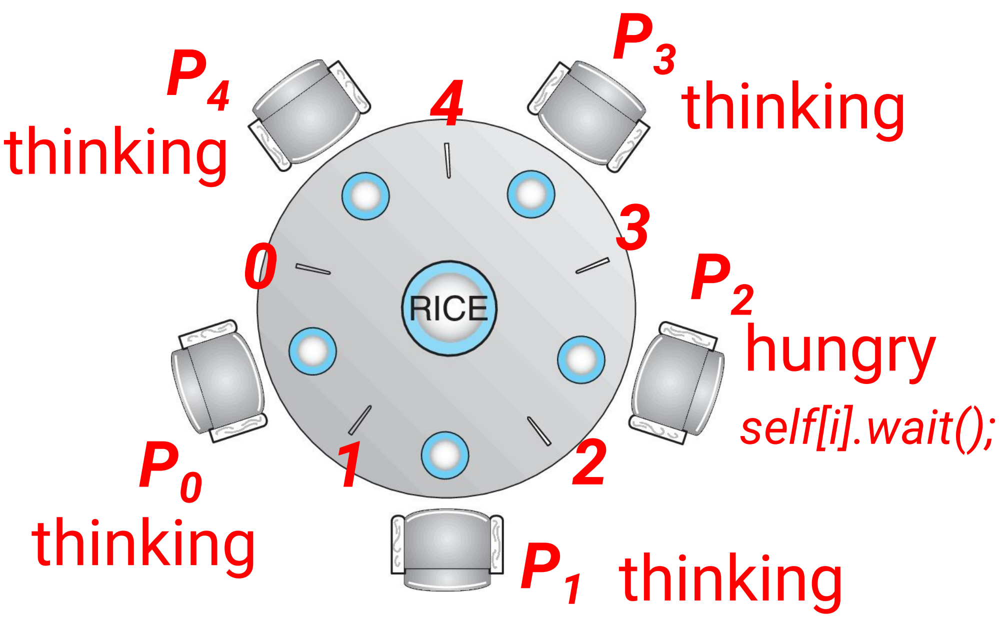
P_2 →

P1:
DiningPhilosophers.pickup(1)
→ *eat*
DiningPhilosophers.putdown(1)

P2:
→ *DiningPhilosophers.pickup(2)*
eat
DiningPhilosophers.putdown(2)

Dining Philosophers Example (cont.)

- An illustration



P1

```
void putdown (int i) {
        state[i] = thinking;
        // check if neighbors are
        // waiting to eat
        test ((i+4) % 5);
        test ((i+1) % 5);
}
```

P1:

DiningPhilosophers.pickup(1)

eat

DiningPhilosophers.putdown(1)

P2:

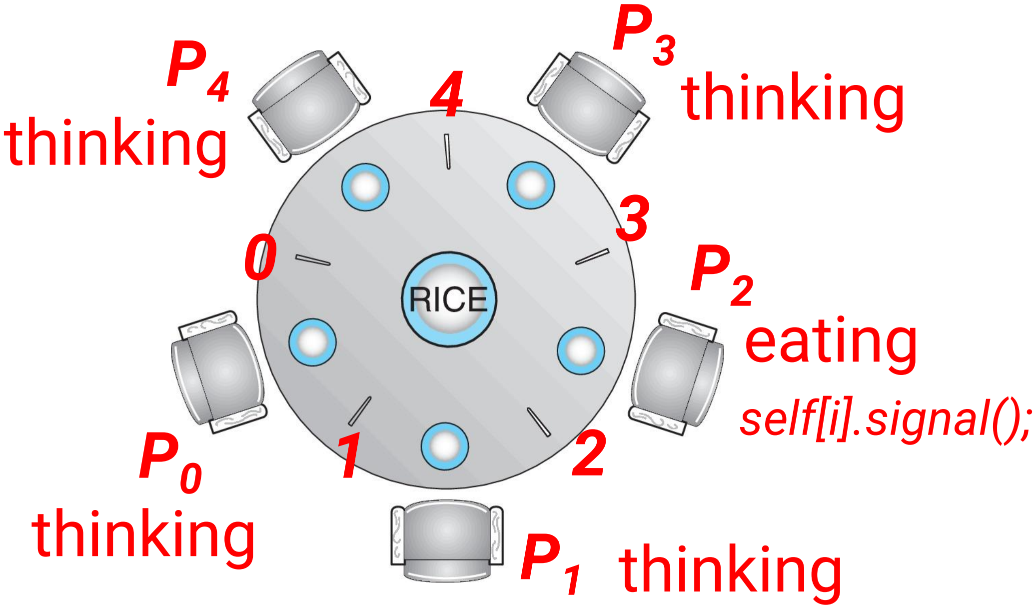
DiningPhilosophers.pickup(2)

eat

DiningPhilosophers.putdown(2)

Dining Philosophers Example (cont.)

- An illustration




```
void putdown (int i) {
        state[i] = thinking;
        // check if neighbors are
        // waiting to eat
P1 ➡    test ((i+4) % 5);
        test ((i+1) % 5);
}
```

P1:

DiningPhilosophers.pickup(1)

eat

➡ *DiningPhilosophers.putdown(1)*

P2:

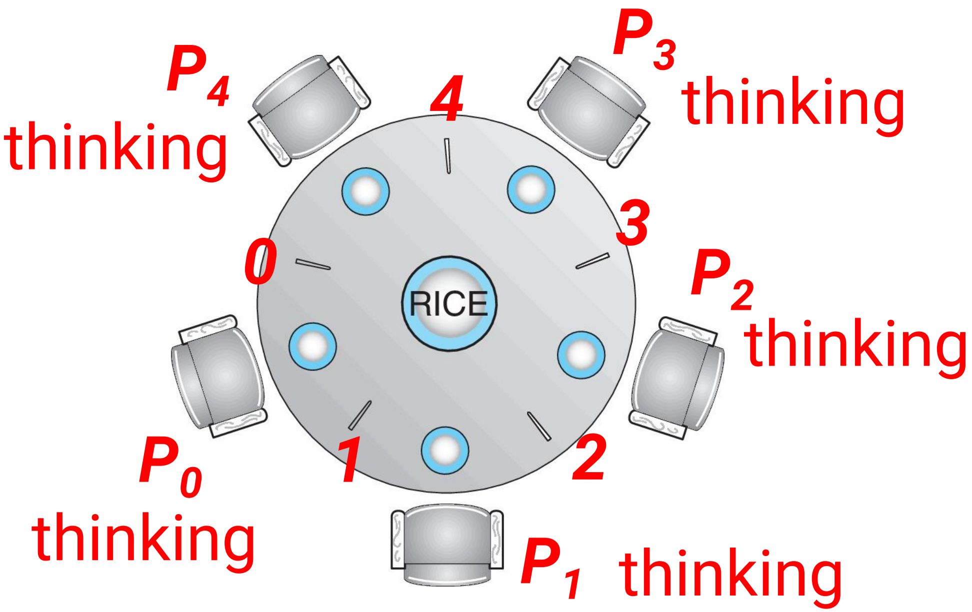
➡ *DiningPhilosophers.pickup(2)*

eat

DiningPhilosophers.putdown(2)

Dining Philosophers Example (cont.)

- An illustration



P1:
DiningPhilosophers.pickup(1)
eat
➡ *DiningPhilosophers.putdown(1)*

P2:
DiningPhilosophers.pickup(2)
eat
➡ *DiningPhilosophers.putdown(2)*

Synchronized Tools in JAVA

- **Synchronized methods (Monitor)**
  - Synchronized method uses the **method receiver** as a lock
  - Two invocations of synchronized methods cannot interleave on the same object
  - When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread exist the object

```
public class SynchronizedCounter
{
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```

Synchronized Tools in JAVA (cont.)

- **Synchronized methods (Mutex Lock)**
  - Synchronized blocks uses the **expression** as a lock
  - A synchronized statement can only be executed once the thread has obtained a lock for the object opr the class that has been referred to in the statement
  - Useful for improving concurrency with fine-grained

```
public void run()
{
    synchronized (p1)
    {
        int i = 10;    // statement without locking requirement
        p1.display (s1);
    }
}
```

Atomic Transactions

System Model

- **Transaction**
  - **A collection of instructions** that performs a **single logic** function
- **Atomic transaction**
  - Operations happen as a single logical unit of work **entirely**, or **not at all**

- Atomic transaction is particular a concern for database system

File I/O Example

- Transaction is a series of **read** and **write** operations
- Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation
- Aborted transaction must be **rolled back** to undo any changes it performed
  - It is part of the responsibility of the system to ensure this property

Log-based Recovery

- **Record** to **stable storage** information about all modifications by a transaction
  - **Stable storage** means never lost its stored data
- **Write-ahead logging**: each log record describes single transaction write operation
  - Transaction name
  - Data item name
  - Old & new values
  - Special events: <T_i starts>, <T_i commits>
- **Log** is used to reconstruct the state of the data items modified by the transactions
  - Use undo (T_i), redo (T_i) to recover data

Checkpoints

- When failure occurs, must consult the log to determine **which transactions must be re-done**
  - Searching process is time consuming
  - Redone may not be necessary for all transaction

- Use checkpoints to reduce the above overhead
  - Output all **log records** to stable storage
  - Output all **modified data** to stable storage
  - Output a log record **<checkpoint>** to stable storage

Objective Review

- Describe the critical-section problem and illustrate a race condition

- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables

- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem