Chapter 7: Synchronization Examples

CS 3423 Operating Systems
Fall 2019
National Tsing Hua University

Classic Synchronization Problems

- Purpose
  - Used for testing newly proposed synchronization schemes
- Problems
  1. Bounded-Buffer problem (producer-consumer)
  2. Readers-Writers problem
  3. Dining-Philosopher problem

1. Bounded-Buffer Problem

- *n* buffers, 3 semaphores
  - semaphore mutex = 1; // mutual exclusive access
  - semaphore full = 0; // "barrier", #items produced
  - semaphore empty = *n*; // #empty buffers
- Producer:
  - [produce] wait(empty), wait(mutex), enqueue-buffer, signal(mutex), signal(full);
- Consumer:
  - wait (full), wait(mutex), dequeue-buffer, signal(mutex), signal(empty); [consume]

1. Bounded-Buffer Problem

- *n* buffers, 3 semaphores
  - mutex = 1, full = 0, empty = *n*;

Producer:

```
produce next item;
wait(empty);
  wait(mutex);
    add next produced to buffer;
  signal(mutex);
signal(full);
```

Consumer:

```
wait(full);
  wait(mutex);
    remove an item from buffer;
  signal(mutex);
signal(empty);
consume the item;
```

Assumption: context switch can happen anywhere!

2. Readers-Writers Problem

- Readers
  - multiple readers at the same time (no writer)
- Writers:
  - at most one writer at a time is allowed to read-and-write shared data.
    => one reader or one writer excludes all other writers
- Possible variations
  1. readers don't wait unless writer is accessing
  2. writer has highest priority, blocks out readers

2.1 Readers-Writers algorithm
Readers don't wait unless writer accessing

```
// mutex for write
semaphore rw_mutex = 1;
semaphore mutex = 1;
int readcount = 0;
```

```
Writer() { // any writer
  while (TRUE) {
    wait(rw_mutex);
    // write code
    signal(rw_mutex);
  }
}
```

```
Reader() {  // any reader
  while (TRUE) {
    wait(mutex);  // protects readcount ++
    readcount++;
    if (readcount==1) {
      wait(rw_mutex);
    }
    // get write lock if readers haven't already
    signal(mutex):
    // read code
    wait(mutex);  // protects readcount --
    if (--readcount == 0) {
      signal(rw_mutex);
    }
    signal(mutex);
  }
}
```

2.1 how it works

- rw_mutex allows at most
  - one of the writers to read/write, or
  - the first reader to read
- mutex
  - allows one of the readers to update readcount at a time, but
  - allows more than one reader to read at the same time when there is no writer

Issues with Readers-Writers Algorithm (v.1)

- Readers share a single write lock
- Writers may have starvation problem
  - Both v1. and v.2 may have starvation leading to even more variations
- Possible solutions
  - on some systems, kernel provides reader-writer locks

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 (data set)
  - Semaphore `chopstick[5] = {1,1,1,1,1};`

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 are problems with this algorithm?
  - (1) Deadlock, (2) Starvation
    ```

Why deadlock?

- Each philosopher i picks up chopstick[i]
- Before picking up chopstick[(i+1)%5], get context switched
- by the time philosopher i gets switched back, tries to pick chopstick[(i+1)%5], but it is already locked by philosopher (i+1)%5
- No philosopher i can pick up chopstick[(i+1)%5] => deadlock!
- Solution: Monitor

Approach with Monitor

- Declare state of each philosopher
  - enum { THINKING, HUNGRY, EATING } state[5];
- Declare condition variable for each philosopher to delay eating if can't obtain chopsticks at the moment
  - condition self[5];
- Declare methods for
  - pickup chopstick i -- possibly block
  - putdown chopstick i -- possibly unblock neighbor
  - "test" -- try to let i eat if it is hungry

Monitor code for Dining Philosophers

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING}
      state[5];
   condition self [5];

   void pickup (int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait;
   }

  void putdown (int i) {
      state[i] = THINKING;
     // test L and R neighbors
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

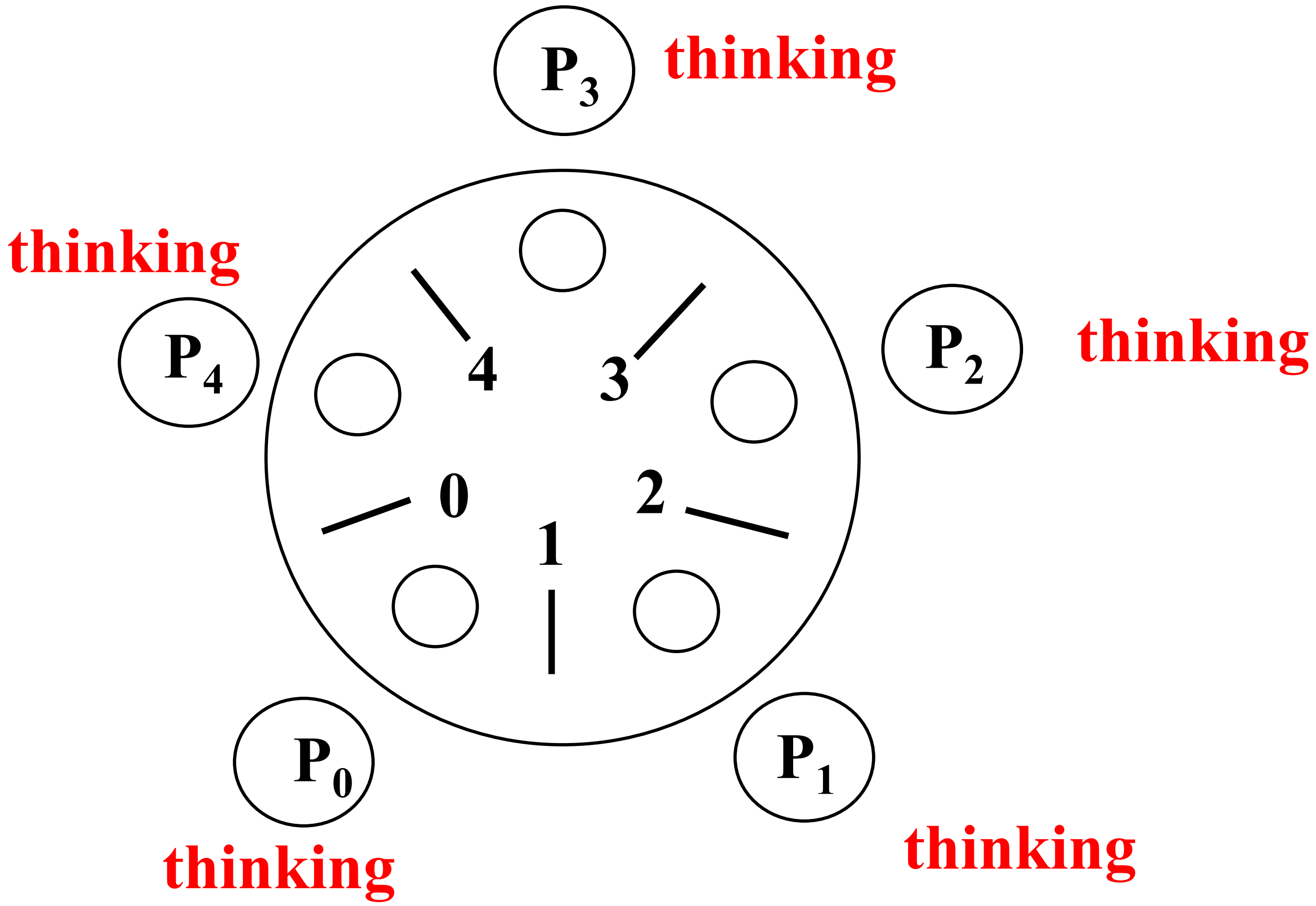
   void test (int i) {
     if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
           self[i].signal () ;
      }
    }

    initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

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

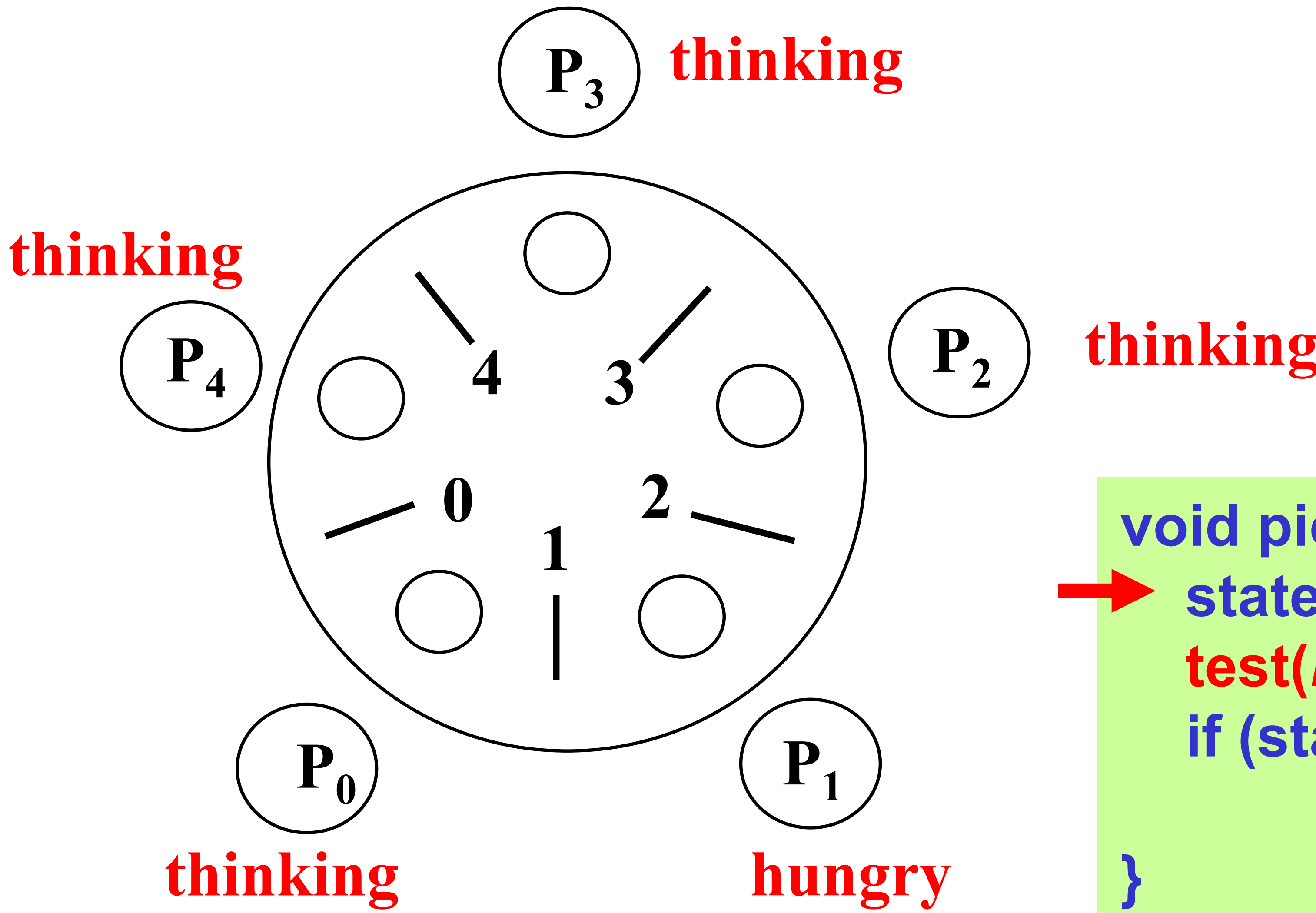
Illustration (1)



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

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

Illustration (2)

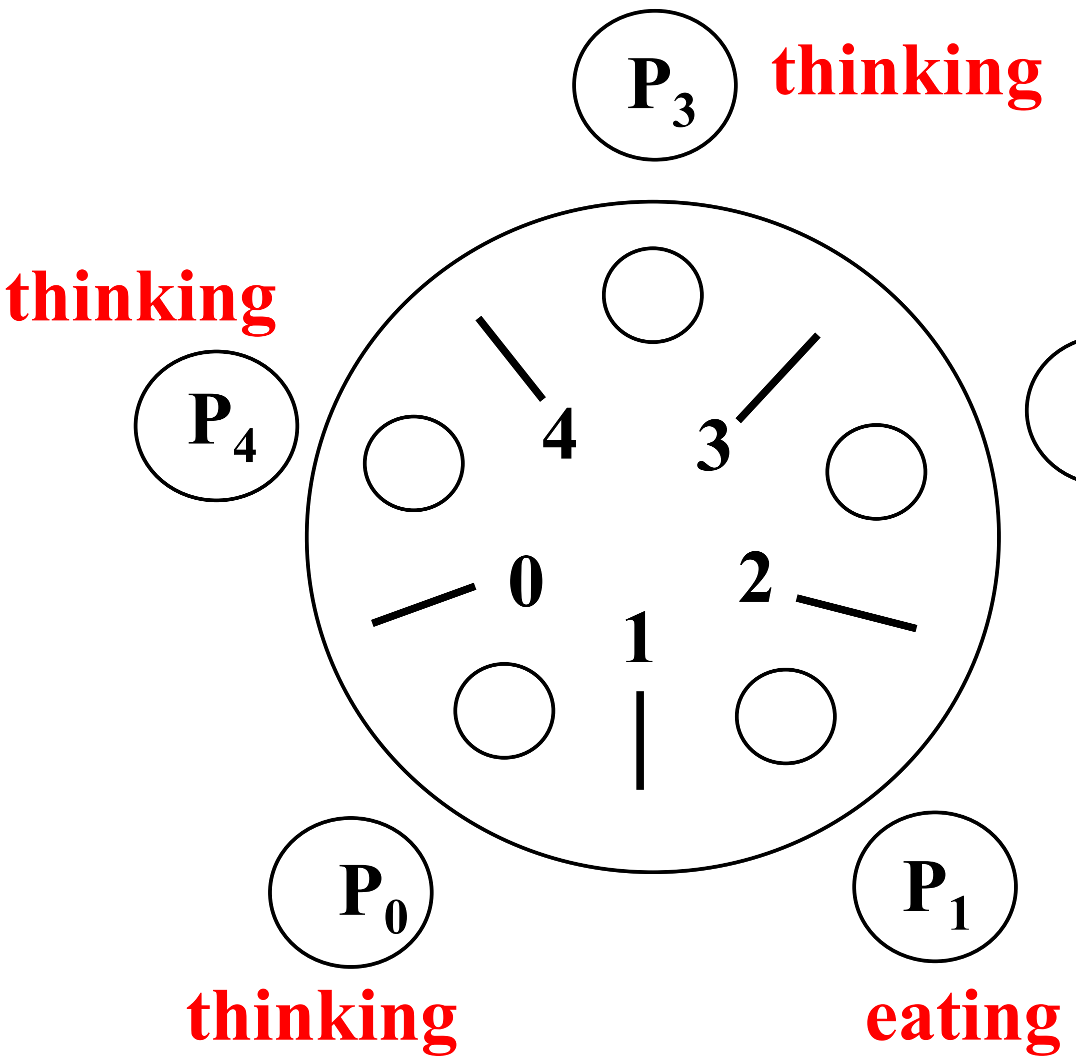



```
void pickup(int i) {
   state[i] = hungry;
   test(i); //try to eat
   if (state[i] != eating)
         self[i].wait();//wait to eat
}
```

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

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

Illustration (3)



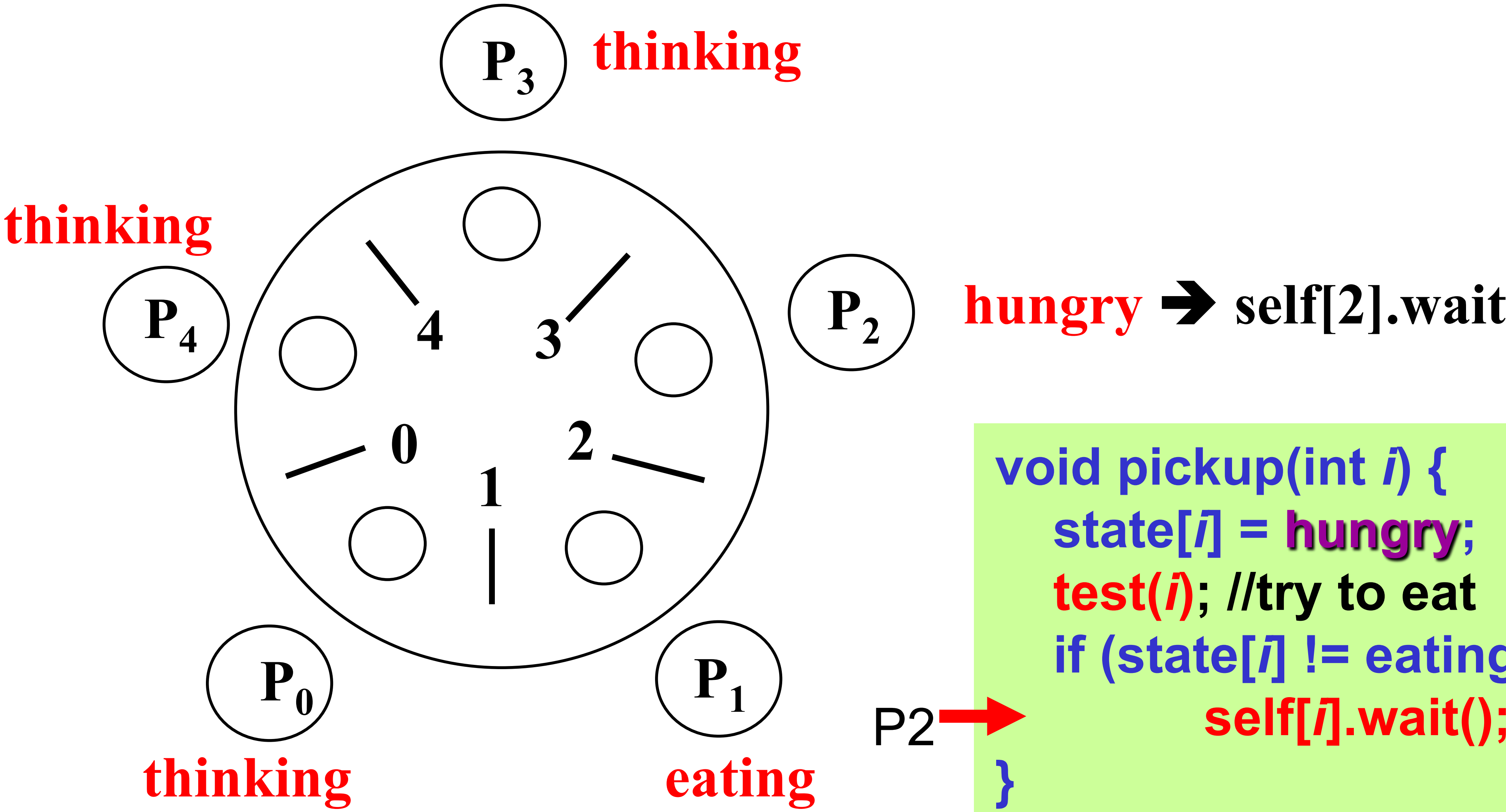
P1 →

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

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

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

Illustration (4)



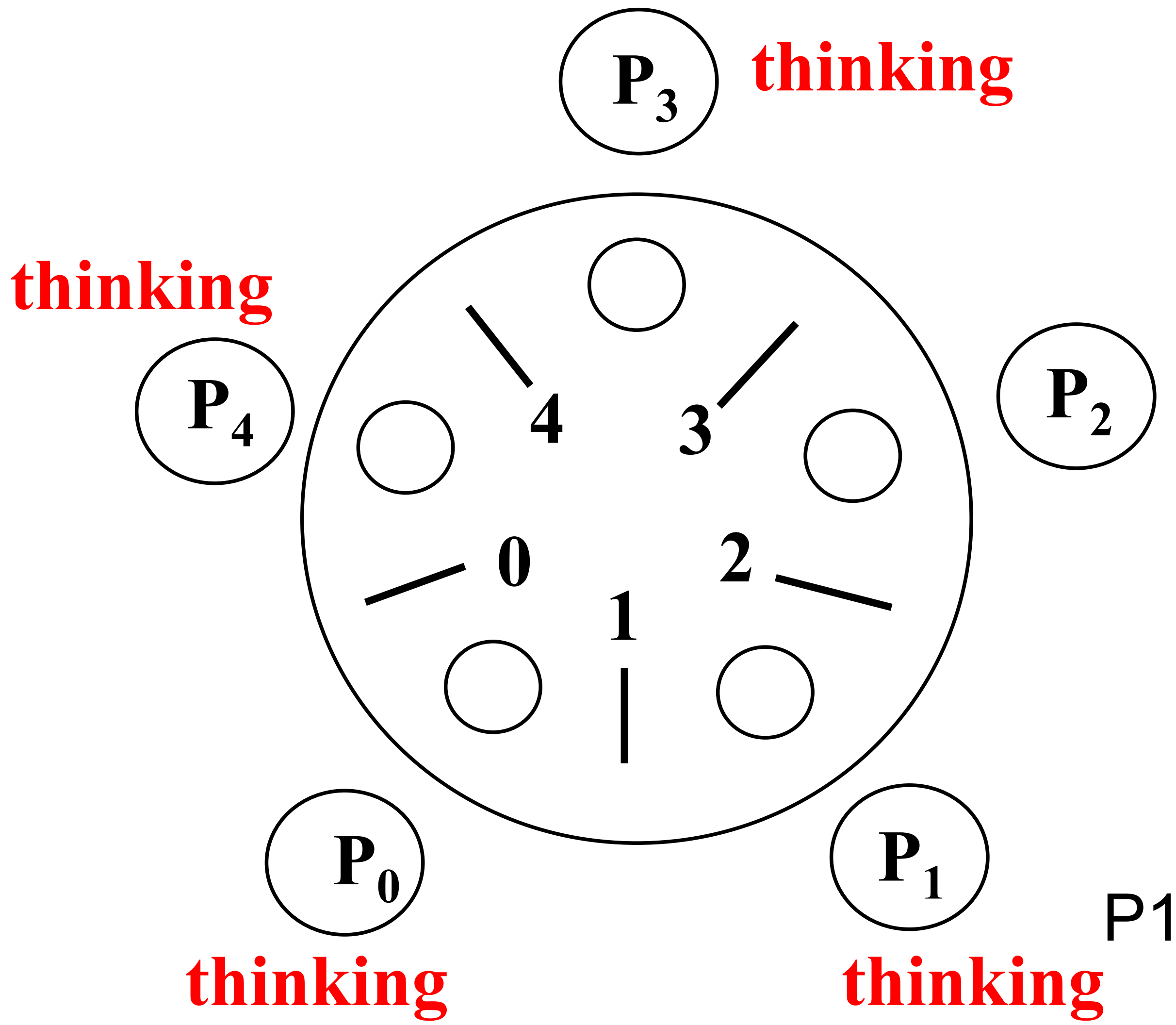

```
void pickup(int i) {
    state[i] = hungry;
    test(i); //try to eat
    if (state[i] != eating)
            self[i].wait();//wait to eat
}
```

P2 →

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

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

Illustration (5)



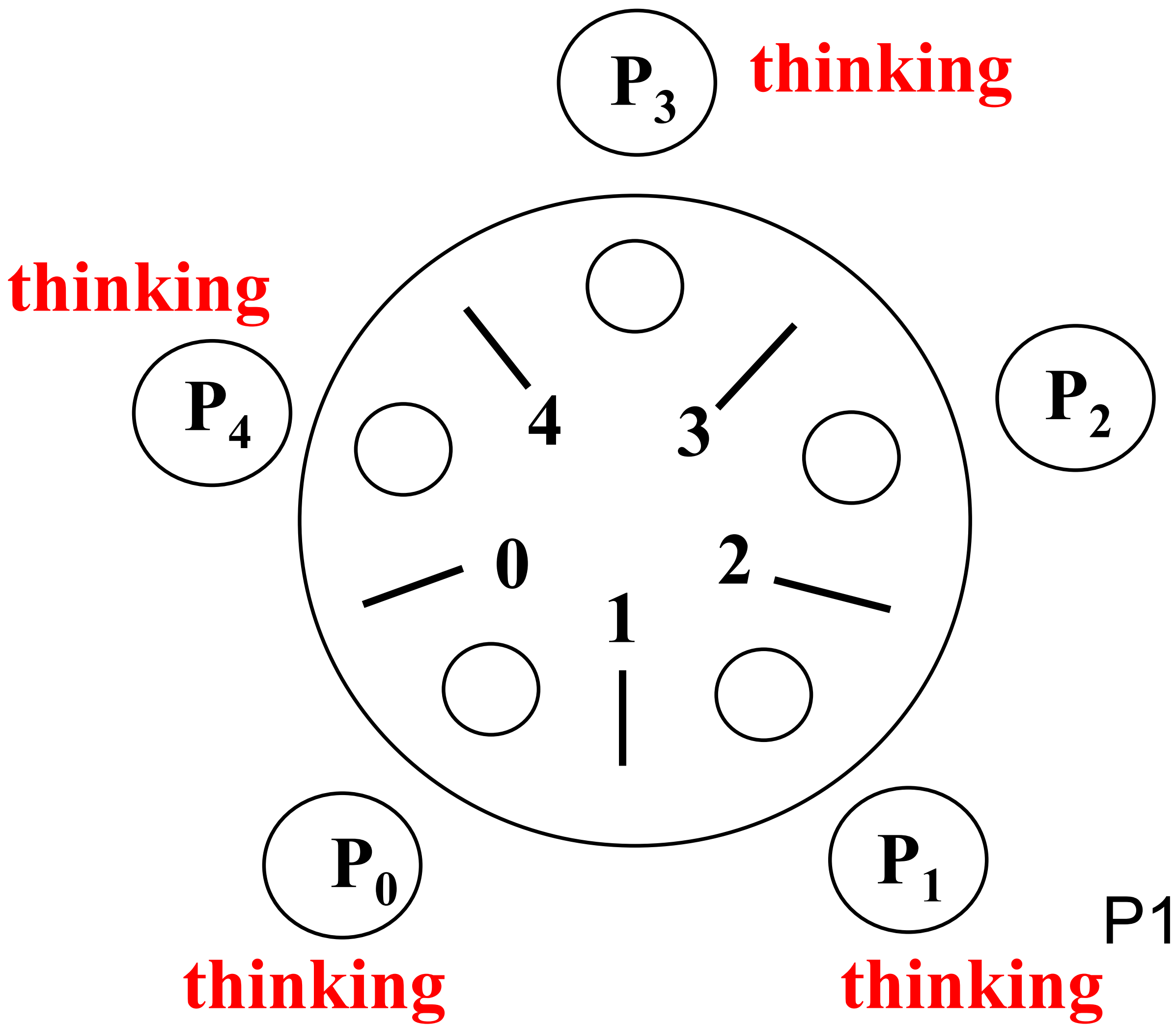

```
void putdown(int i) {
    state[i] = thinking;
    // check if neighbors
    // are waiting to eat
    test((i+4) % 5); test((i+1) % 5);
}
```

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)

Illustration (6)



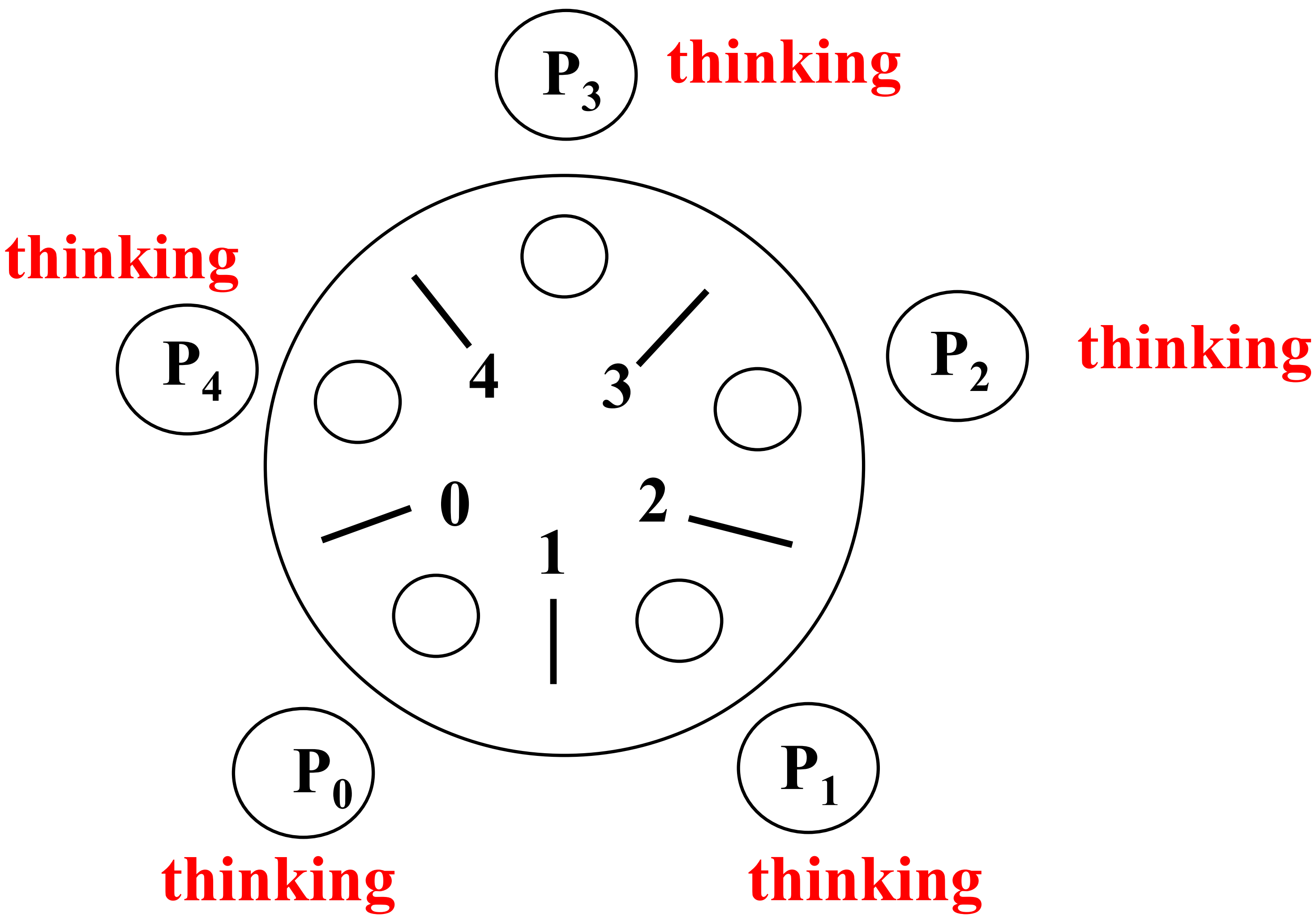
eating ➔ self[2].signal

```
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)

Illustration (7)



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

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