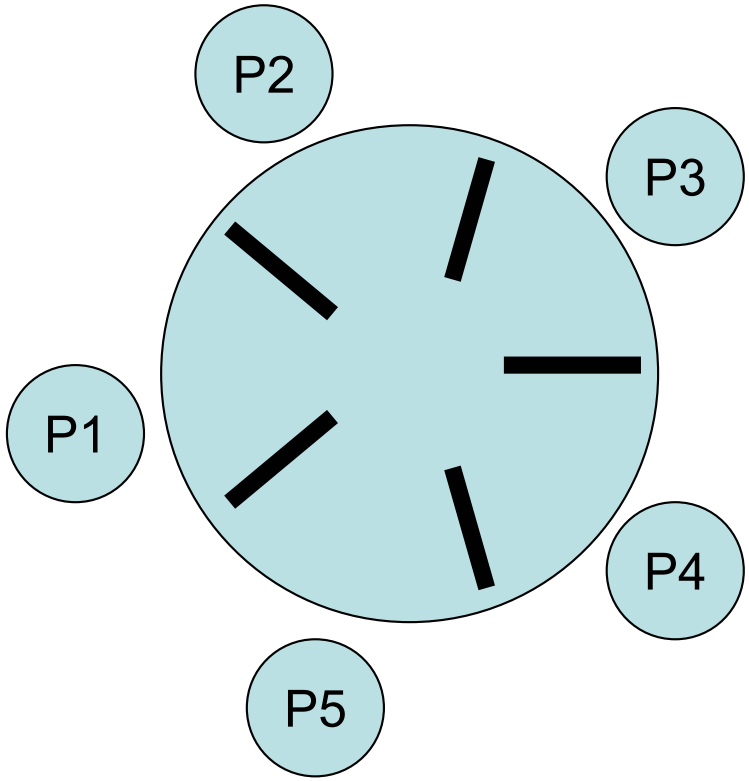
Dining Philosophers Problem

- *N* philosophers seated around a circular table
  - There is one chopstick between each philosopher
  - A philosopher must pick up its two nearest chopsticks in order to eat
  - A philosopher must pick up first one chopstick, then the second one, not both at once
- Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is
  - deadlock-free, and
  - starvation-free



Dining Philosophers Problem

- A simple algorithm for protecting access to chopsticks:
  - each chopstick is governed by a mutual exclusion semaphore that prevents any other philosopher from picking up the chopstick when it is already in use by another philosopher

    semaphore chopstick[5]; // initialized to 1

  - Each philosopher grabs a chopstick *i* by P(chopstick[i])
  - Each philosopher releases a chopstick *i* by V(chopstick[i])

Dining Philosophers Problem

- Pseudo code for Philosopher i:

```
while(1) {
    // obtain the two chopsticks to my immediate right and left
    P(chopstick[i]);
    P(chopstick[(i+1)%N];

    // eat

    // release both chopsticks
    V(chopstick[(i+1)%N];
    V(chopstick[i]);
}
```

- Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

Dining Philosophers Problem

- Unfortunately, the previous “solution” can result in deadlock
  - each philosopher grabs its right chopstick first
    - causes each semaphore’s value to decrement to 0
  - each philosopher then tries to grab its left chopstick
    - each semaphore’s value is already 0, so each process will block on the left chopstick’s semaphore
  - These processes will never be able to resume by themselves - we have deadlock!

Dining Philosophers Problem

- Some deadlock-free solutions:
  - allow at most 4 philosophers at the same table when there are 5 resources
  - odd philosophers pick first left then right, while even philosophers pick first right then left
  - allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.
    - we'll see this solution next using monitors
- A deadlock-free solution is not necessarily starvation-free
  - for now, we'll focus on breaking deadlock

Solution 1 *(give a number to forks and always try with the smaller)*

```
Array [1..5] of semaphores: fork = [1, 1, 1, 1, 1];
Philosopher i:
    Repeat
        think;
        if (i < 4)
        then    fork[i].down();
                fork[i+1].down();
        else    fork[0].down();
                fork[4].down();
        eat;
        fork[(i+1) mod 5].up();
        fork[i].up();
```

Solution 2 *(odd and even philosophers behave differently)*

```
Array [1..5] of semaphores: fork = [1, 1, 1, 1, 1];
Philosopher i:
    Repeat
        think;
        if (i mod 2 = 0)
        then    fork[i].down();
                fork[(i+1) mod 5].down();
        else    fork[(i+1) mod 5].down();
                fork[i].down();
        eat;
        fork[(i+1) mod 5].up();
        fork[i].up();
```

Solution 3 *(allow at most 4 philosophers at a time sitting at the table)*

Array [1..5] of semaphores: fork = [1, 1, 1, 1, 1];

semaphore: table = 4

Philosopher i:

```
    Repeat
        think;
        table.down();
        fork[i].down();
        fork[(i+1) mod 5].down();
        eat;
        fork[(i+1) mod 5].up();
        fork[i].up();
        table.up()
```

Monitor-based Solution to Dining Philosophers

- Key insight: pick up 2 chopsticks only if both are free
  - this avoids deadlock
  - reword insight: a philosopher moves to his/her eating state only if both neighbors are not in their eating states
    - thus, need to define a state for each philosopher
  - 2nd insight: if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
    - thus, states of each philosopher are: thinking, hungry, eating
    - thus, need condition variables to signal() waiting hungry philosopher(s)
  - Also, need to Pickup() and Putdown() chopsticks

Monitor-based Solution to Dining Philosophers

- Some basic pseudo-code for monitor (we'll abbreviate DP for Dining Philosophers):

```
monitor DP {
   status state[5];
   condition self[5];
   Pickup(int i);
   Putdown(int i);
}
```

- Each philosopher *i* runs pseudo-code:

```
DP.Pickup(i);
...
DP.Putdown(i);
```

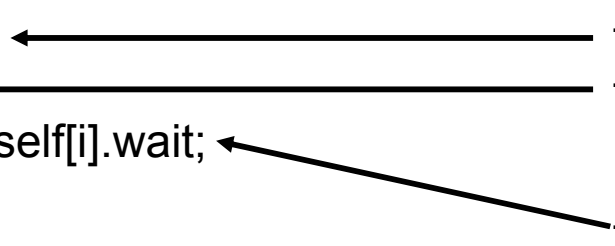
Monitor-based Solution to Dining Philosophers

- Full code for monitor solution (continued on next slide):

```
monitor DP {
  status state[5];
  condition self[5];

  Pickup(int i) {
    state[i] = hungry;
    test(i);
    if(state[i]!=eating) self[i].wait;
  }

  Putdown(int i) {
    state[i] = thinking;
    test((i+1)%5);
    test((i-1)%5);
  }
  ... monitor code continued next slide ...
```



- Pickup chopsticks
  - indicate that I'm hungry
  - set state to eating in test() only if my left and right neighbors are not eating
  - if unable to eat, wait to be signalled
- Put down chopsticks
  - if right neighbor R=(i+1)%5 is hungry and both of R's neighbors are not eating, set R's state to eating and wake it up by signalling R's CV

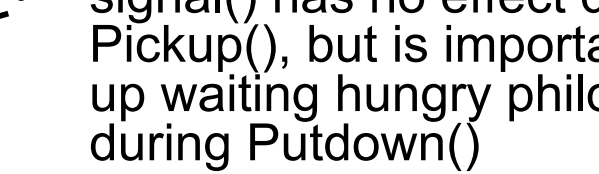
Monitor-based Solution to Dining Philosophers

```
... monitor code continued from previous
      slide...
   ...
   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry) {

         state[i] = eating;
         self[i].signal();
      }
   }

   init() {
      for i = 0 to 4
         state[i] = thinking;
   }

}  // end of monitor
```



- signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

- Execution of Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
  - deadlock-free
  - mutually exclusive in that no 2 neighbors can eat simultaneously