Chapter 07
Synchronization Examples

Classical problems and solutions

Outline

- Classic Problems of Synchronization
- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- Alternative Approaches

Objectives

- Explain the bounded-buffer, readers–writers, and dining–philosophers synchronization problems.
- Describe alternative approaches to solve process synchronization problems.

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- Served as the synchronization primitives:
  - ***n*** buffers, each can hold one item
  - Semaphore **`mutex`** initialized to the value 1
  - Semaphore **`full`** initialized to the value 0
  - Semaphore **`empty`** initialized to the value n

Bounded-Buffer Problem

- **Producer**

```
do {
    ...
    /* produce an item in next_produced */
       ...
    wait(empty);
    wait(mutex);
       ...
    /* add next produced to the buffer */
       ...
    signal(mutex);
    signal(full);
} while (true);
```

Bounded-Buffer Problem

- **Consumer**

```
do {
    wait(full);
    wait(mutex);
        . . .
    /* remove an item from buffer to next_consumed */
        . . .
    signal(mutex);
    signal(empty);
        . . .
    /* consume the item in next_consumed */
        . . .
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at one time
  - Only a single writer can access the shared data at a time

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem

- Writer

```
do {
   wait(rw_mutex);
      ...
   /* perform writing */
      ...
   signal(rw_mutex);
} while (true);
```

- Reader

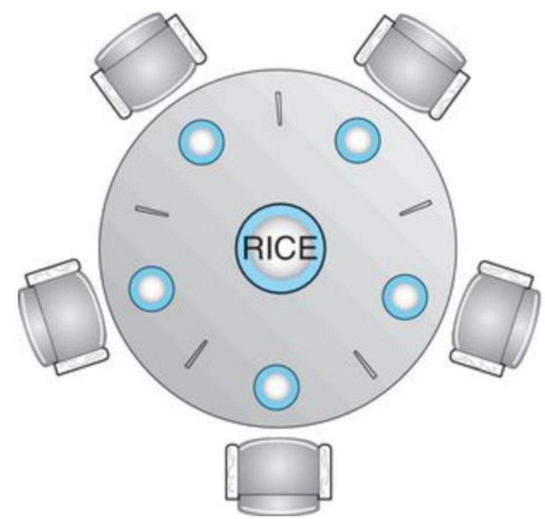
```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      ...
   /* perform reading */
      ...
   wait(mutex);
   read count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

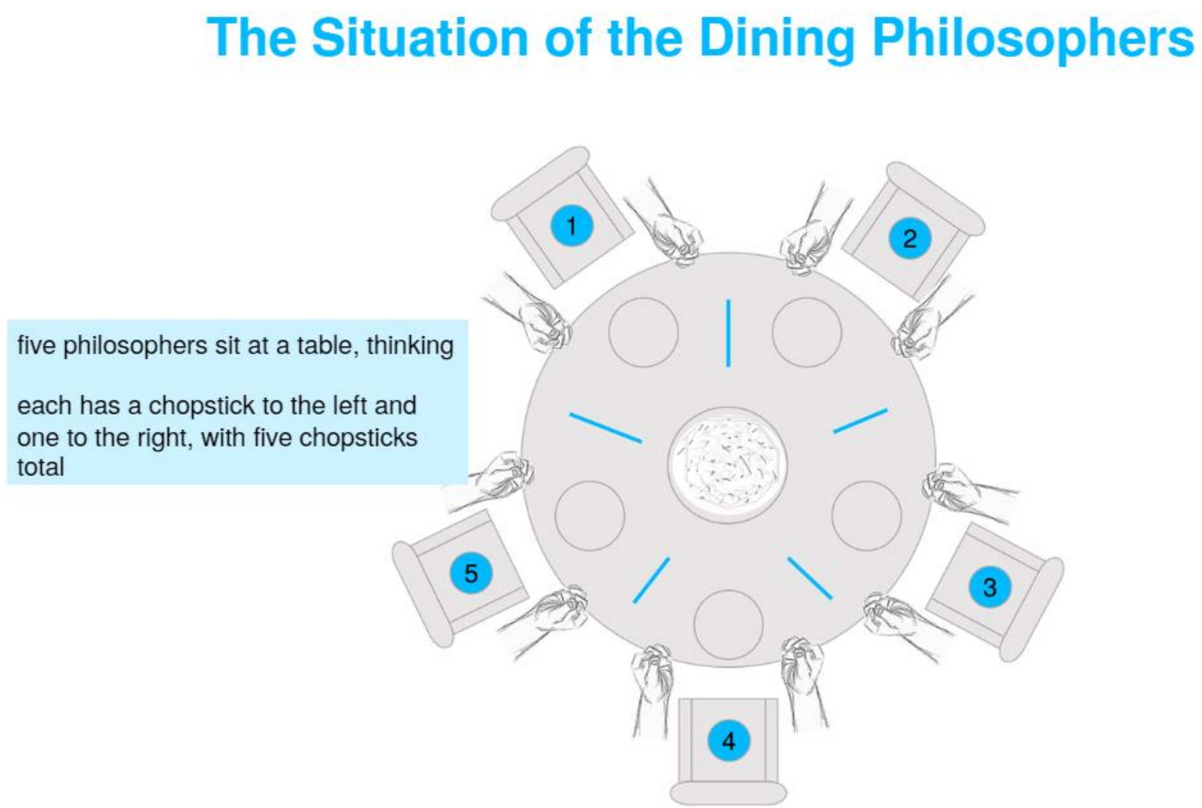
- Several variations of how readers and writers are considered – all involve some form of priorities
  - The ***first*** readers–writers problem
    - No reader be kept waiting unless a writer permitted to access.
    - Writer may starve.
  - The ***second*** readers–writers problem
    - No reader may start reading if a write is waiting to access.
    - Reader may starve.
- Problem is solved on some systems by kernel providing reader-writer locks.
  - Need specifying the mode of lock (read or write)
  - Useful if it is easy to identify read / write operations, or if there are more readers than writers.

Dining-Philosophers Problem

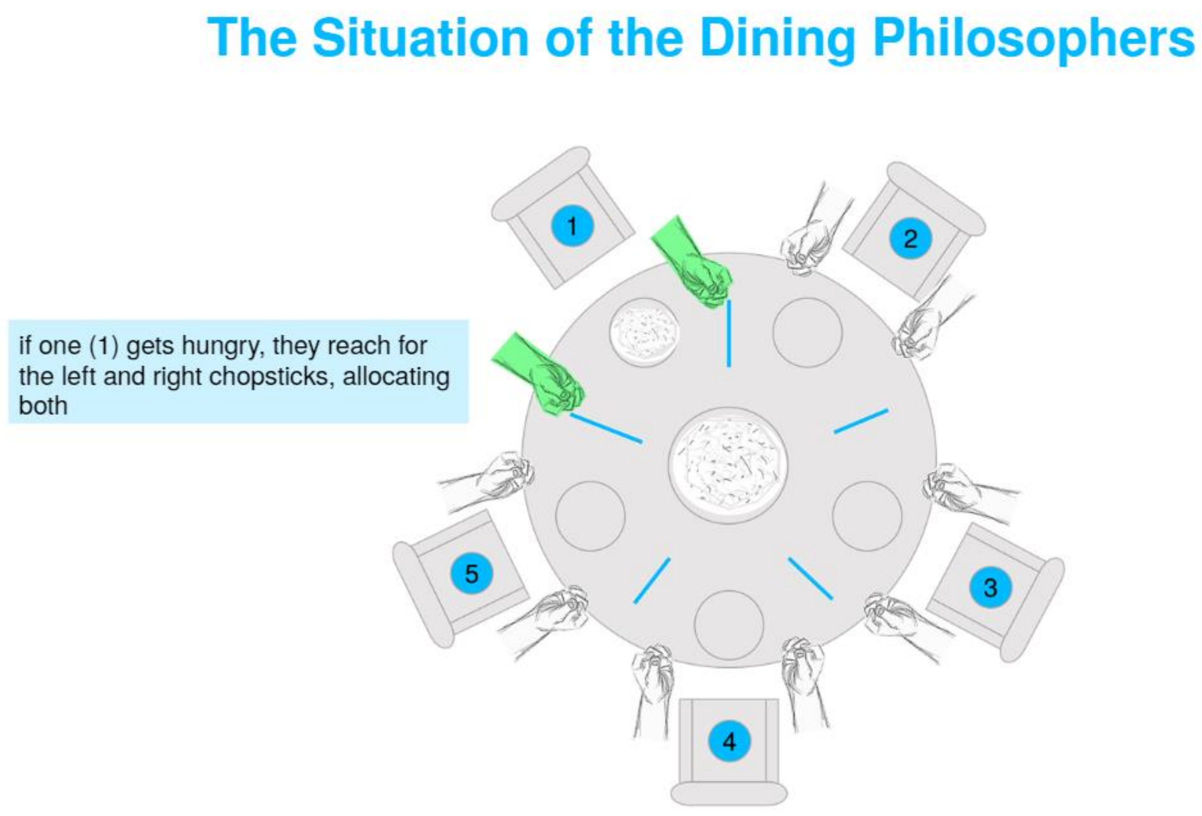
- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data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



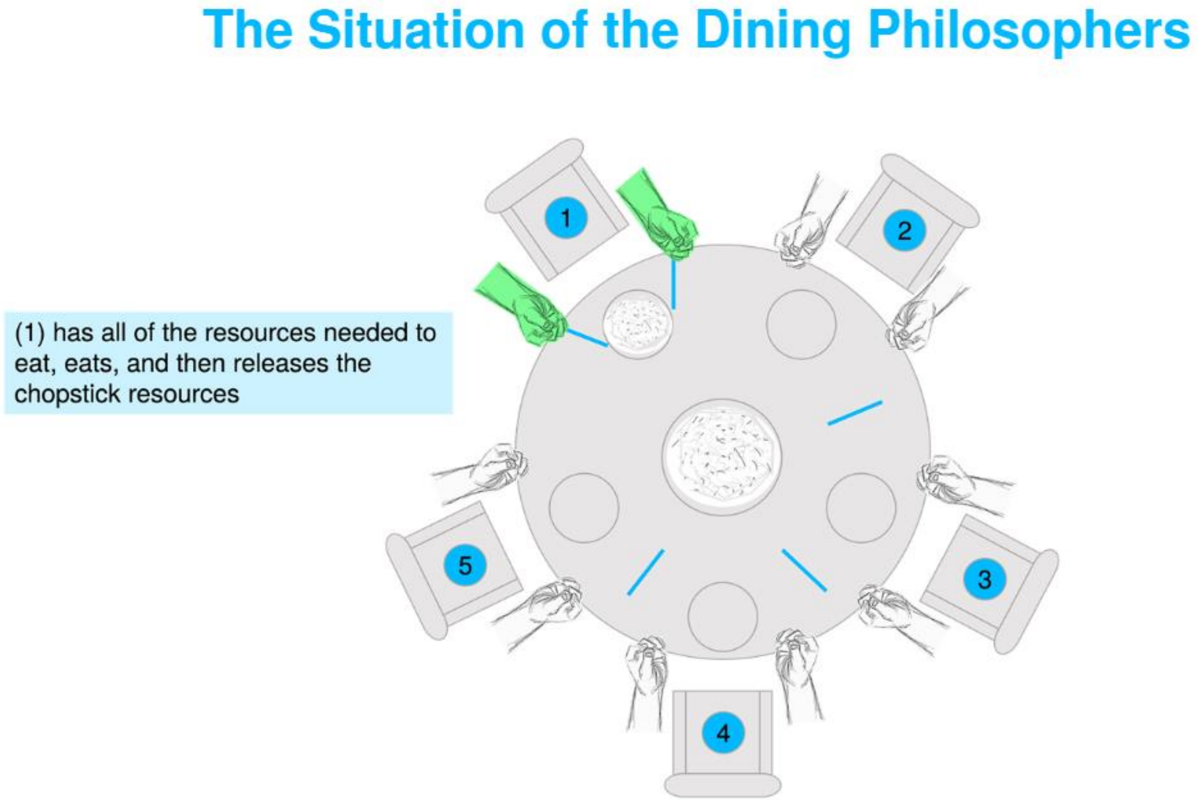
Dining-Philosophers Problem



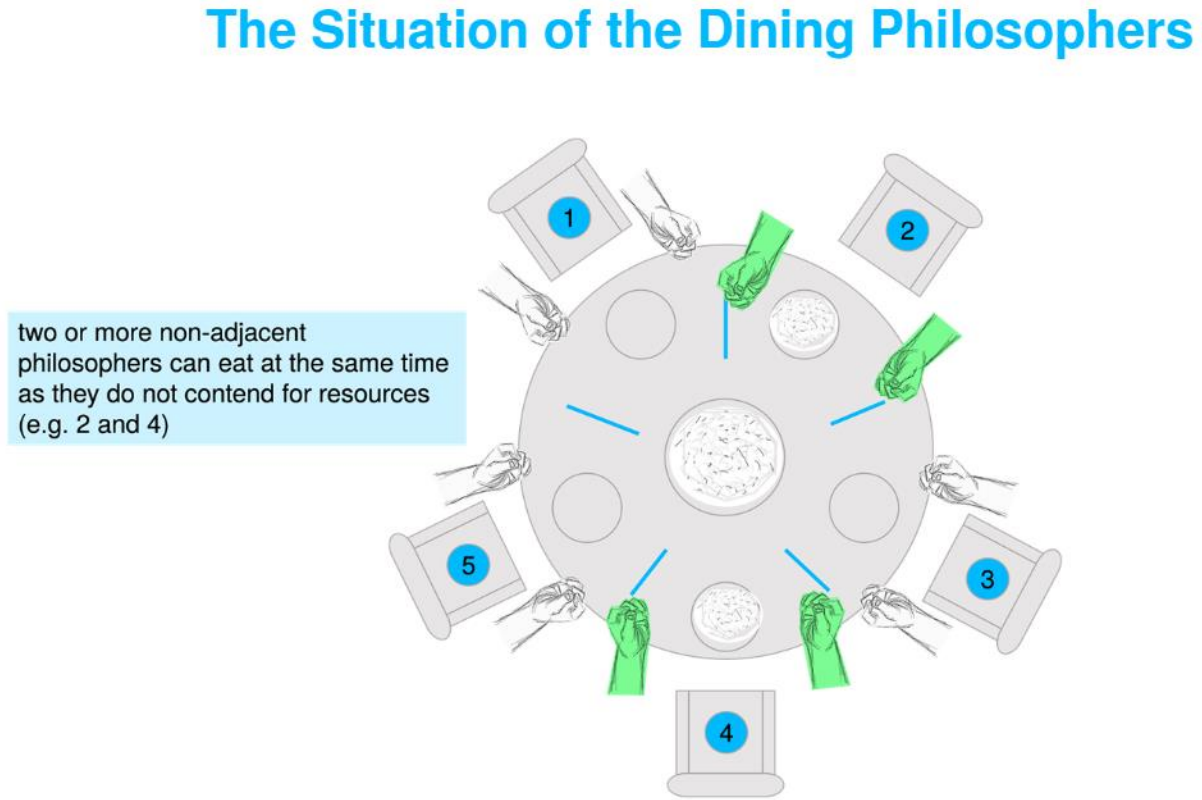
Dining-Philosophers Problem



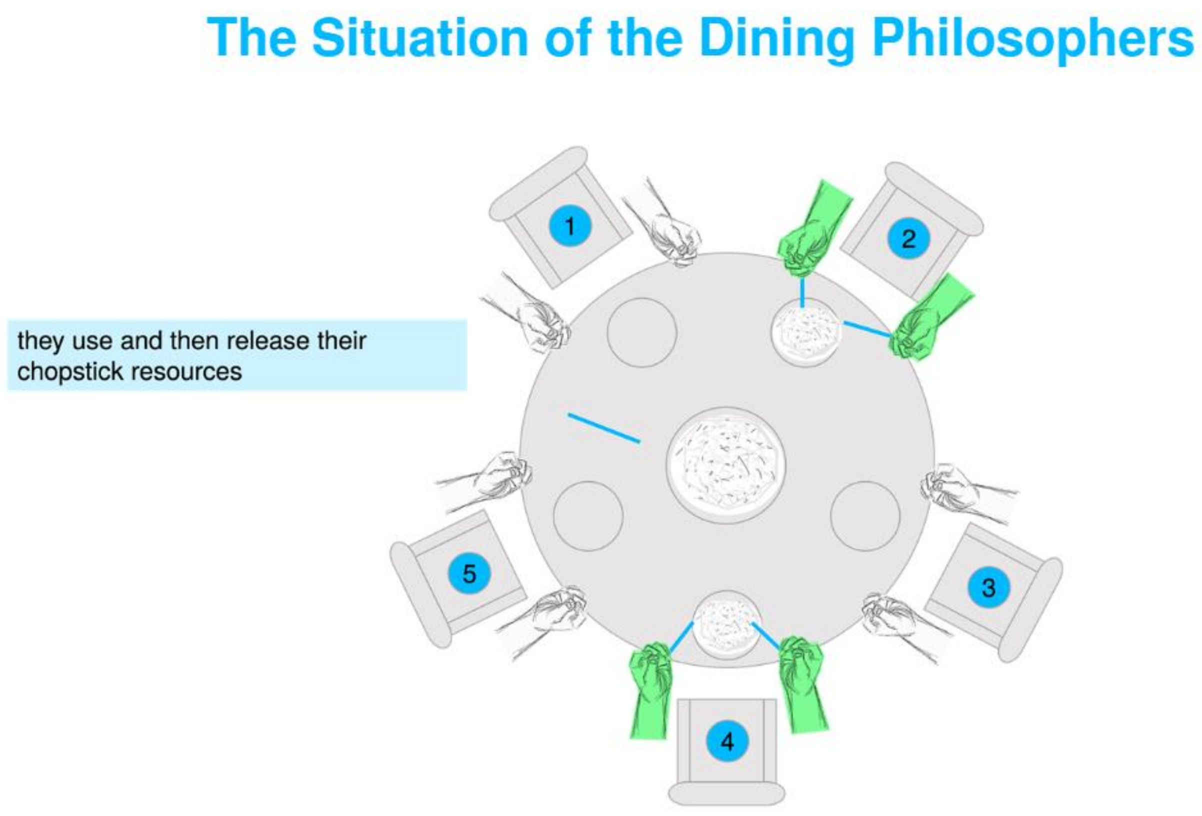
Dining-Philosophers Problem



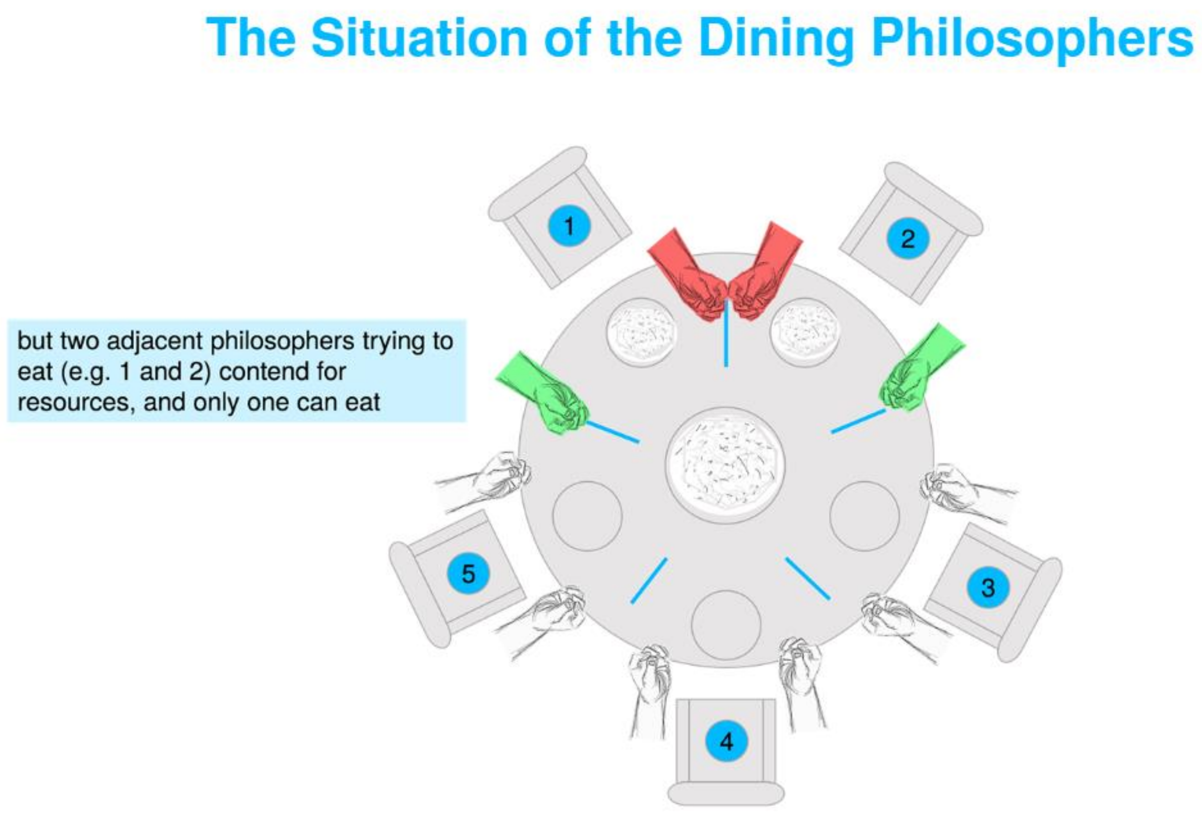
Dining-Philosophers Problem



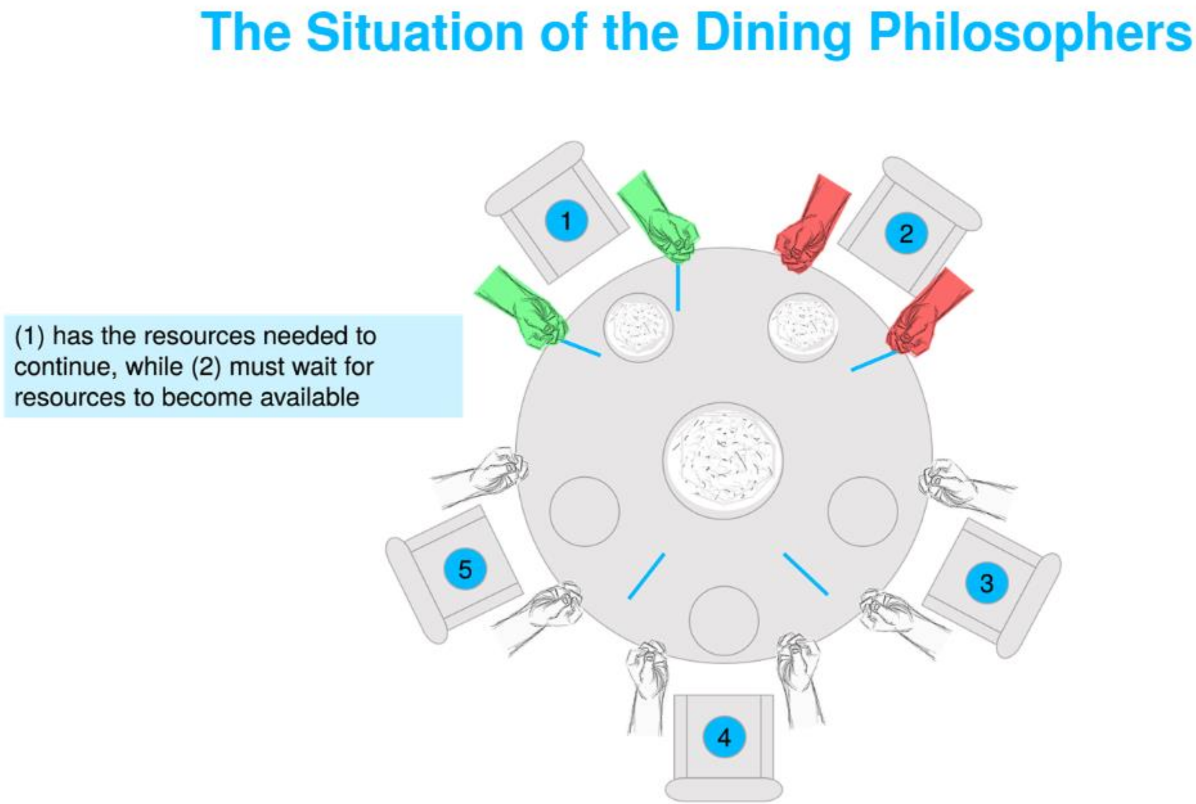
Dining-Philosophers Problem



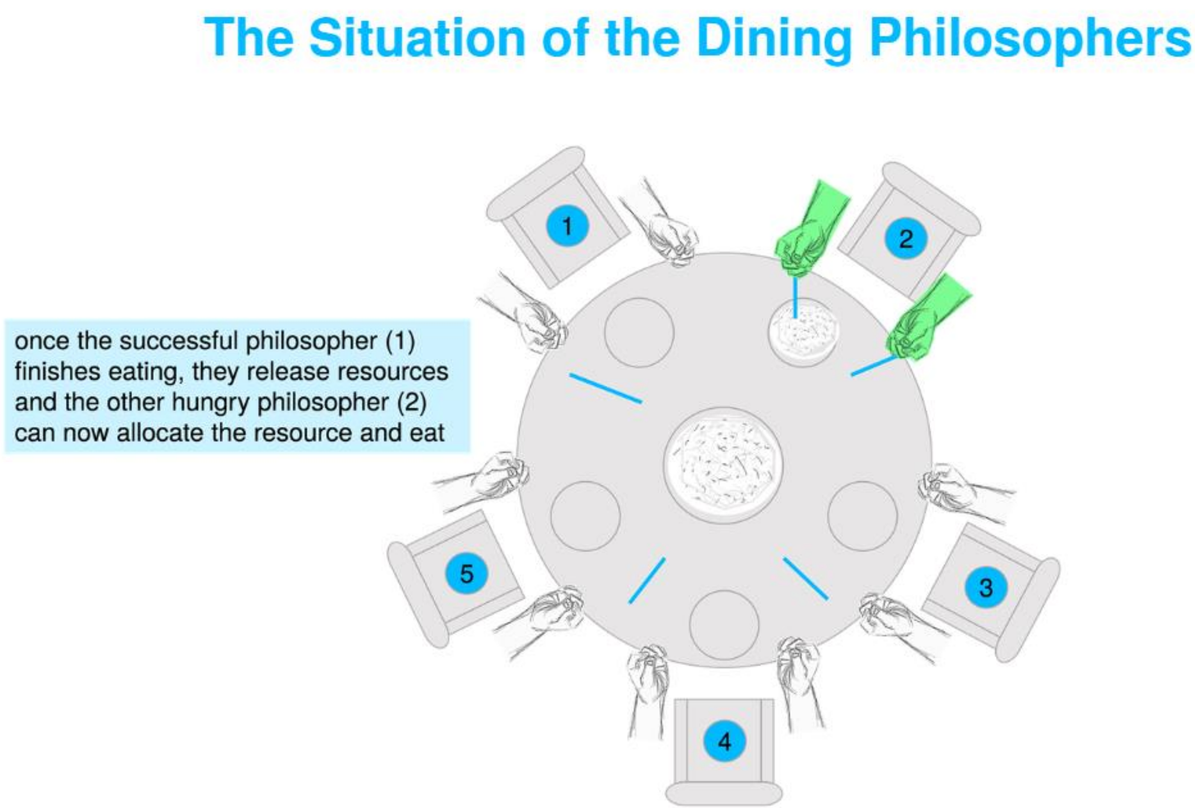
Dining-Philosophers Problem



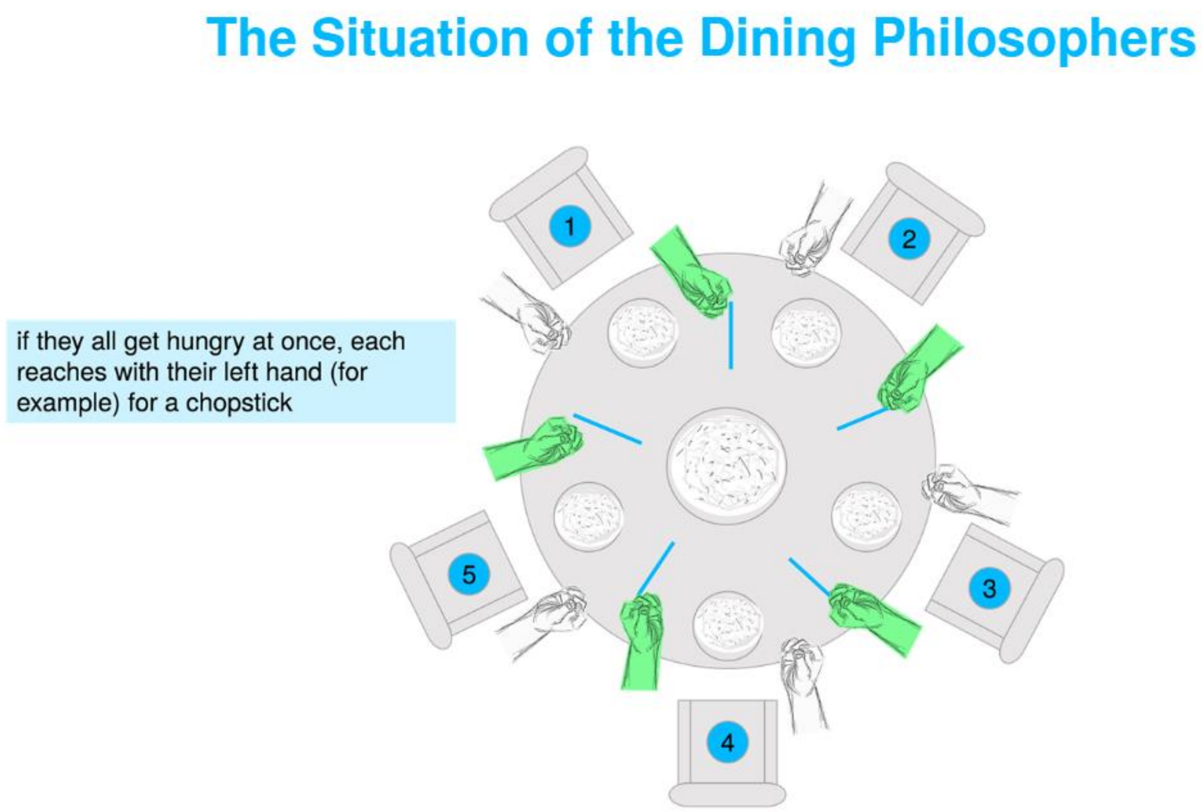
Dining-Philosophers Problem



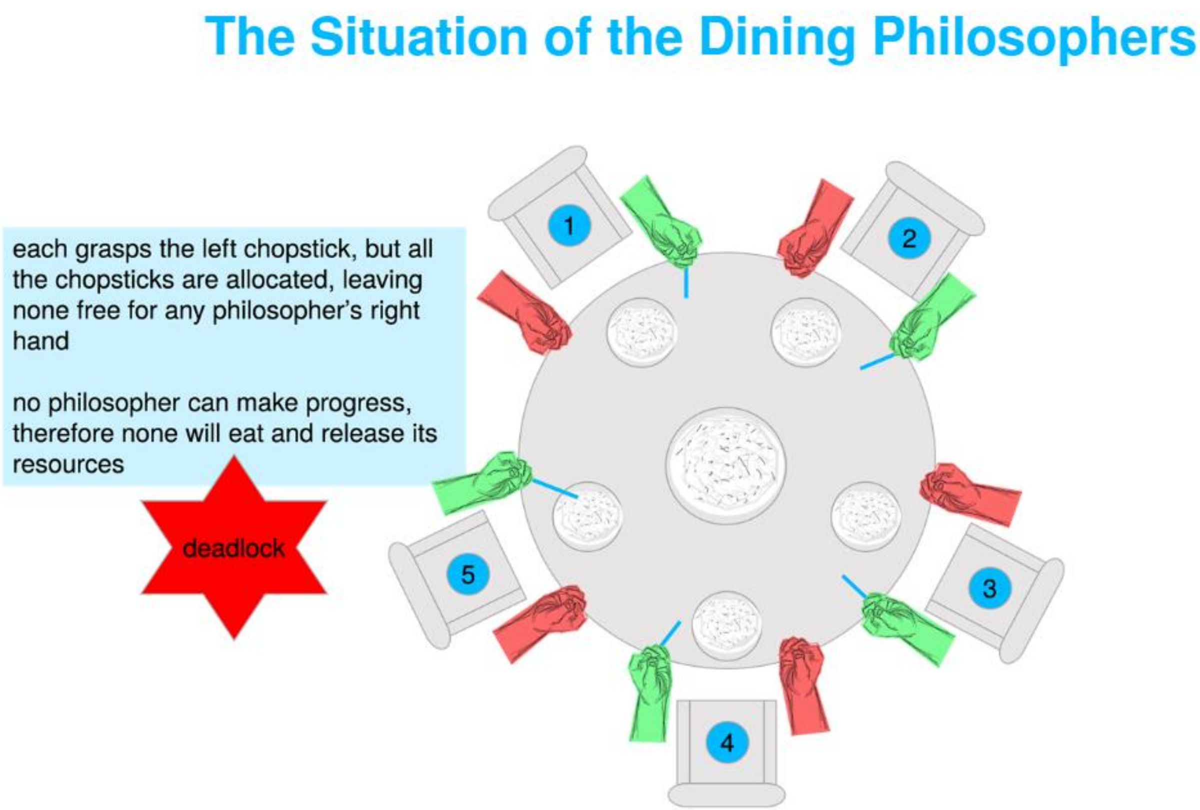
Dining-Philosophers Problem



Dining-Philosophers Problem



Dining-Philosophers Problem



Dining-Philosophers Problem

- Solution using Semaphore
  - Shared data

```
semaphore chopstick[5];
```

  - 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?
    - Deadlock will occur if all five philosophers get their left (or right) chopstick.

Dining-Philosophers Problem

- Several possible remedies to the deadlock problem.
  - Allow at most four philosophers sit at one time
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - Asymmetric solution
    - Odd-number philosopher first gets left chopstick, while even-number philosopher first gets right chopstick.
- Must also prevent from starvation
  - Deadlock-free != non-starvation

Dining-Philosophers Problem

- Solution using Monitor
  - A philosopher may pick up her chopsticks only if both of them are available
  - Distinguish among three states:
    enum {THINKING, HUNGRY, EATING} state[5];
  - Philosopher *i* can eat iff two neighbors are not eating
    - state[(i+4) % 5] != EATING and state[(i+1) % 5] != EATING
  - Condition variable for delay if hungry but either chopsticks are unavailable.
    condition self[5];
- Solution using **`monitor DiningPhilosophers`**:

```
DiningPhilosophers.pickup(i);
                 ...
                 Eat
                 ...
DiningPhilosophers.putdown(i);
```

Dining-Philosophers Problem

- Definition of **`monitor DiningPhilosophers:`**

```
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((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;
   }
}
```

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- A memory transaction is a sequence of read-write operations to memory that are performed atomically.
  - If all operations in a transaction are completed, the memory transaction is committed.
  - Otherwise, the operations must be aborted and rolled back.

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

- Advantage
  - Transactional memory system responsible for guaranteeing atomicity
  - Transactional memory system can identify which statements in atomic blocks can be executed concurrently

OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
   #pragma omp critical
   {
      count += value
   }
}
```

- The code contained within the #pragma omp critical directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.