Operating systems

- Process synchronization
- Deadlocks

Process Synchronization

- Background
- Critical Section Problem
- Semaphores
- Classical Problems

Background

- A **cooperating** process is one that can affect or be affected by the other processes executing in the system.
- Any process that does **share** data with other processes is a **cooperating** process.
- **Concurrent access** to shared data may result in data **inconsistency**.
- Maintaining data consistency requires **mechanisms** to ensure the **orderly execution** of cooperating processes.

Cooperating processes

- Consider the following pseudo code:

```
global x
x = 0
x = x + 1
print x
```

```
Value of x:
    0
    1
1<== output
```

Cooperating processes

- Two processes executing same code:

P0	P1	Value of x:
global x	global x	
x = 0		0
x = x + 1		1
	x = 0	0
print x		0 ←P0 output
	x = x + 1	1
	print x	1 ←P1 output

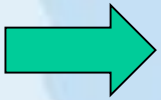
Critical Section Problem

- Each process has a segment of code, called **a critical section**, in which the process may be changing common variables, updating a table, writing a file, etc.
- When one process is executing in its critical section, **no other process** is to be allowed to execute in its critical section.
- Thus, the execution of critical sections by processes should be **mutually exclusive** in time.

Critical Section Problem

- Design a **protocol** which processes can use to cooperate.
- Each process must request **permission** to enter its critical section implemented in the **entry section**.

General structure of a typical process



```
do {
        entry section
            critical section
        exit section
            remainder section
} while (TRUE);
```

Critical Section Problem

- A solution to the critical-section problem must satisfy the following requirements:

1. **Mutual exclusion**: if process P_i is executing in its critical-section, then no other process can be executing in its critical-section.
2. **Progress:** If no process is in its critical-section, one of the processes that wish to enter their critical-sections will be selected to enter its critical-section.
3. **Bounded waiting:** A bound on the number of selections that a process can wait for entering its critical-section.

Two-process solution

Process P_0

```
do {
    while (turn != 0) {}

        critical section

    turn = 1

        remainder section
} while (TRUE)
```

Process P_1

```
do {
    while (turn != 1){}

        critical section

    turn = 0

        remainder section
} while (TRUE)
```

turn is shared variable. Initially **turn** = 0 or 1. This algorithm satisfies mutual exclusion, but not progress.

No progress solution

- The problem is that algorithm doesn't retain information about the **state** of the process.
- Solution: **add flag[2]** shared variable.
- If **flag[i]** == **true** → process **i** is **ready** to enter the critical section.
- Initial state:

  flag[0] = **flag[1]** = **false**, **turn** = **0** or **1**

Two-process solution

Process P_0

```
do {
    flag[0] = TRUE
    turn = 1
    while (flag[1] && turn==1){}

        critical section

    flag[0] = FALSE

        remainder section
} while (TRUE)
```

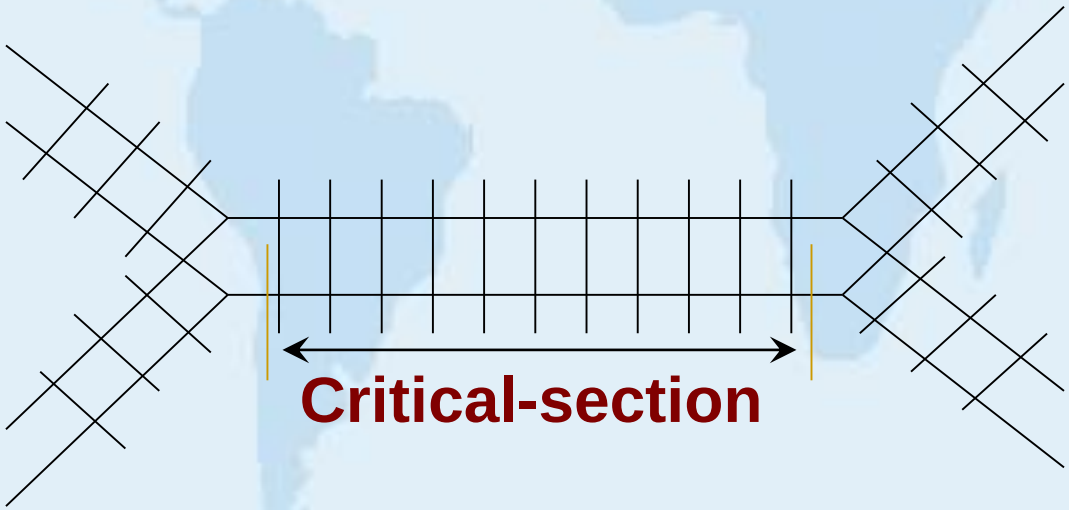
Process P_1

```
do {
    flag[1] = TRUE
    turn = 0
    while (flag[0] && turn==0){}

        critical section

    flag[1] = FALSE

        remainder section
} while (TRUE)
```

Process **P_1** enters **critical section** when **flag[0]** = FALSE or **turn** = **1.**

Semaphores

- Previous solution is difficult to **generalize** to bigger problems, i.e. many processes.
- Another solution to the critical section problem is a **semaphore.**



Semaphore

Semaphore Operations

- A semaphore **S** is an integer variable, that, apart from initialization, is accessed **only** through two standard atomic operations: **wait** and **signal.**
- The classic definition of **wait** and **signal**:

```
wait(S){
    while S ≤ 0
          do no-op
    S--;
}
```

```
signal(S){
   S++;
}
```

Semaphore Implementation

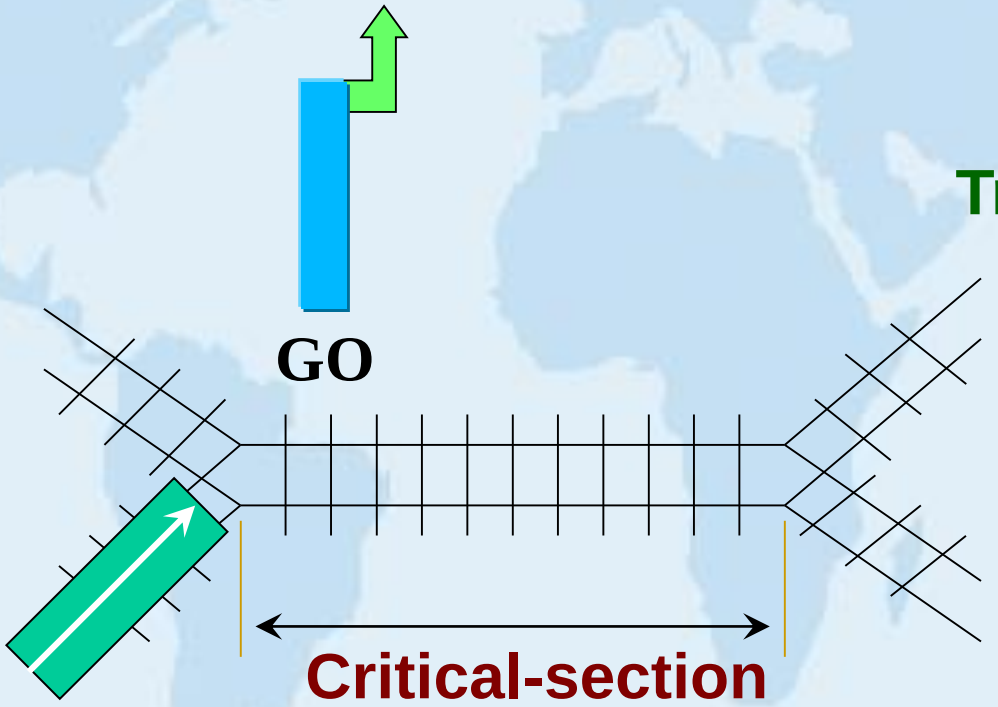
- When one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value.
- In addition, in case of **wait**(S), the testing of integer value S (S ≤ 0), and its possible modification (S--), must also be executed without interruption.

Semaphore: Usage

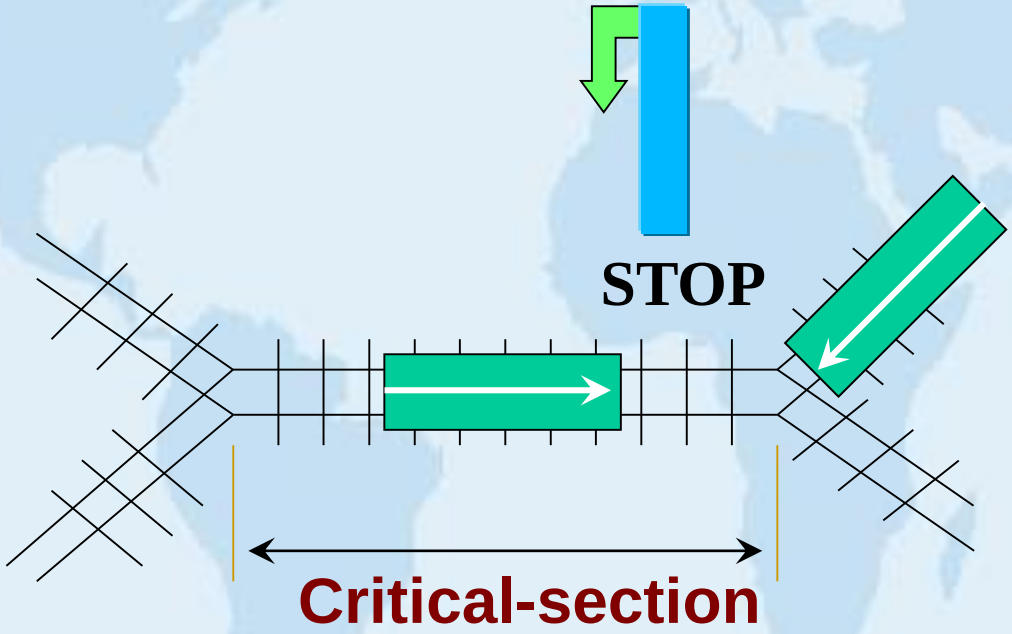
- A solution to the ***n***-process critical-section problem by semaphore. The ***n*** processes share a semaphore, **mutex**, initialized to 1. Each process P_i is organized as follows:

```
do {
        wait (mutex)

        // critical section

        signal (mutex)

        // remainder section
} while (TRUE)
```

Semaphore Representation

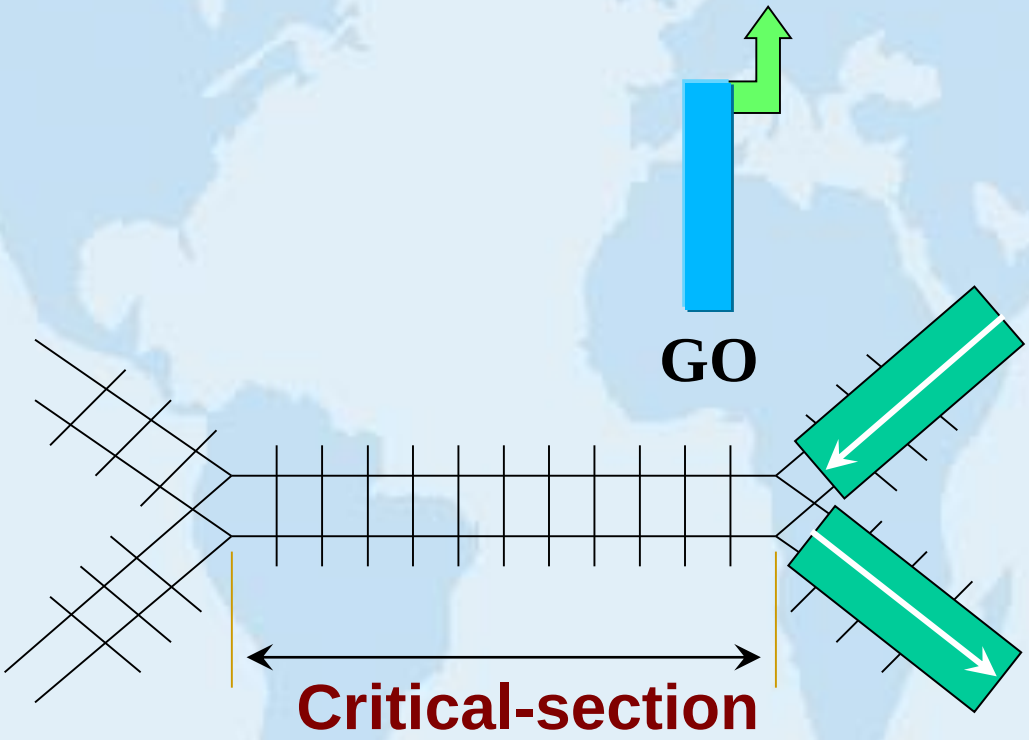


Semaphore Representation



Train 1 is in the critical-section and train 2 arrives

Semaphore Representation



Train 1 lives the critical-section and train 2 can enter

Busy Waiting Problem

- The main disadvantage of the semaphore is the **busy waiting:** While a process is in its critical-section, any other process that tries to enter its critical-section must loop continuously in the wait statement, which wastes CPU time.
- This type of semaphore is called a **spinlock** (because process “spins” while waiting for lock).
- Waiting requires **no** context switch, so spinlocks (short time) are often used in multiprocessor systems.

Solution to busy waiting

- One solution to **busy waiting** is to associate each semaphore a **waiting queue.** When a process is in its critical-section, the other processes trying to enter their critical-sections are put in the waiting queue (process state changes from **running** to **waiting**).
- The control of CPU is transferred to the **CPU scheduler**. The FIFO or other strategies can be used for the queue associated to a semaphore.

Waiting state

- When some process executes a **signal** operation, a blocked process in the queue is restarted by **wakeup** operation (state changes from **waiting** to **ready**)
- Only one process is allowed executing **wait** and **signal** on the same semaphore at the same time. **Busy waiting** is useful for **wait** and **signal** in multiprocessor system.

Semaphore Type

- **Block** and **wakeup** involve context switch.

  If context switch takes more time than spinlock, then **spinlock** is more efficient.

- Let's see a solution without **spinlock**.

  To implement semaphores under this definition, we define a semaphore as:

```
typedef struct{
    int value;
    list Q;
}semaphore;
```

Semaphores operations

```
wait (semaphore S){
     S.value--;
     if S.value < 0 {
          add this process to S.Q;
          block ();}
}
signal (semaphore S):
     S.value++;
     if S.value ≤ 0{
          remove process P from S.Q;
          wakeup(P);}
}
```

Semaphore Requirements

- Semaphores implementation must satisfy the following requirements:

1. **Mutual exclusion**
2. **Progress**
3. **No deadlock**

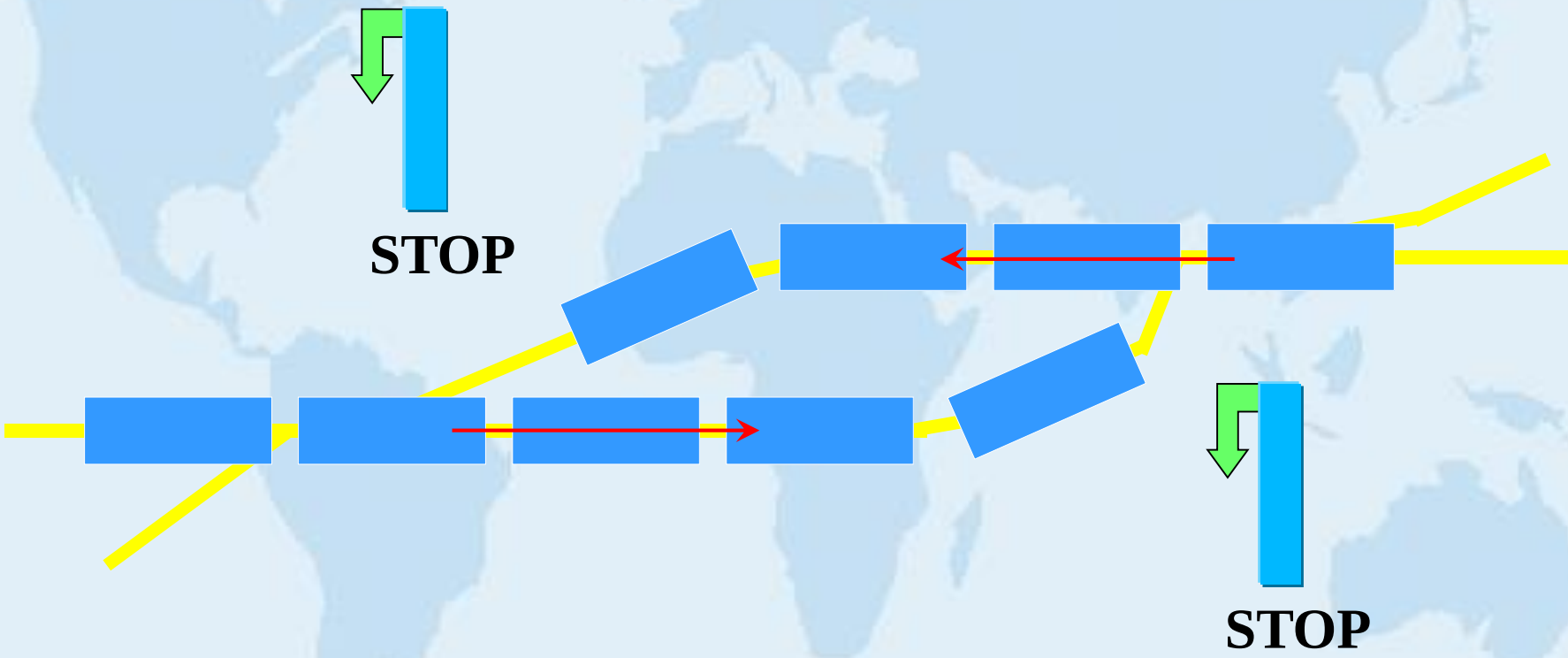
Semaphore - Deadlock

- An example of deadlock with two semaphores S and Q.

```
Process P0

Wait(S);
Wait(Q);
Critical section
Signal(S);
Signal(Q);
```

```
Process P1

Wait(Q);
Wait(S);
Critical section
Signal(Q);
Signal(S);
```

Semaphore - Deadlock



Synchronization Summary

- **Cooperating** process interact by sharing data and resources.
- To protect the shared access to data and resources, process must have **mutual exclusion** to critical-section code and must be **synchronized** (have an orderly execution).

Classical Problems

- **Classical problems** are simple examples that represent very complex, real problems.
- A classical problem shows the important features of the real problem and real solution.
- These problem are found in operating systems and applications.

Classical Problems

- **The Bounded Buffer Problem**
  (Producer-Consumer Problem)
- **The Readers Writers Problem**
- **The Dining Philosophers Problem**

The Bounded Buffer Problem

- Process P_0 is a **producer** of data.
- Process P_1 is a **consumer** of data.
- P_0 puts data into a **buffer** of bounded size **n**.
- P_1 gets the data from the buffer.
- If the buffer is **full**, P_0 waits.
- If the buffer is **empty**, P_1 waits.

The Bounded Buffer Problem Semaphores

- Three semaphores are used:
  - **mutex** (initialized to 0) provides mutual exclusion for access to the buffer.
  - **empty** (initialized to n) counts the empty locations of the buffer.
  - **full** (initialized to 0) counts the full locations of the buffer.
- **empty** and **full** are called **counting semaphores**.

The Bounded Buffer Problem Solution

```
Producer P0
do{
      produce data;
      wait(empty);
      wait(mutex);
      add data to
            the buffer;
      signal(mutex);
      signal(full);
}while (TRUE);
```

```
Consumer P1
do {
      wait(full);
      wait(mutex);
      remove data from
            the buffer;
      signal(mutex);
      signal(empty);
      consume data;
}while (TRUE);
```

The Readers Writers Problem

- Processes $P_0, P_1, \ldots, P_n$ **read** data from a file.
- Processes $Q_0, Q_1, \ldots, Q_m$ **write** data to the file.
- If a process reads data, other process **can** read data (**many readers OK**).
- If a process writes data, **no other** process can read or write data (**one writer OK**, read and write not OK).

The Readers Writers Problem Semaphores

- The **wrt** semaphore is for the mutual exclusion for access to the shared files.
- The **mutex** is for mutual exclusion for the count update.
- The **r_count** keeps the number of processes that are reading the file.
- **Starvation** of writers – when **r_count** does not reach **0**!

The Readers Writers Problem Solution

Writer Q

```
do {
   wait(wrt);
      writing;
   signal(wrt);
}while (TRUE);
```

Reader P

```
do {
   wait(mutex);
   r_count++;
   if r_count=1; wait(wrt);
   signal(mutex);
   reading;
   wait(mutex);
   r_count--;
   if r_count=0; signal(wrt);
   signal(mutex);
}while (TRUE);
```

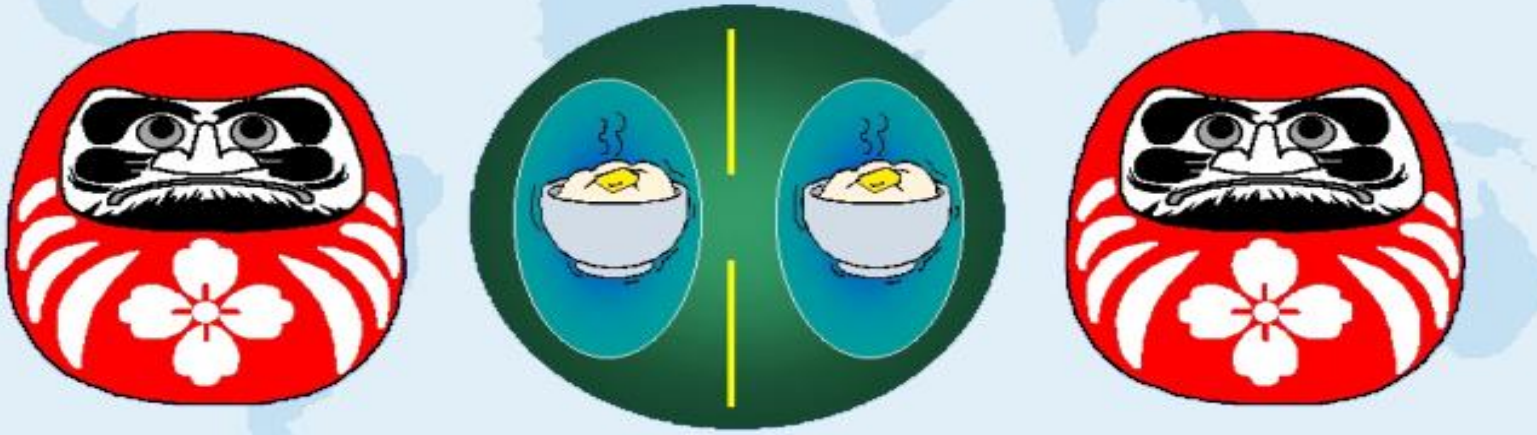
The Dining Philosophers Problem

2 philosophers spend their lives thinking and eating. They share a table and 2 chopsticks (1 pair).
A philosopher gets 2 chopsticks to eat.
After eating, the philosopher puts down the 2 chopsticks.

Question: How can both philosophers eat?

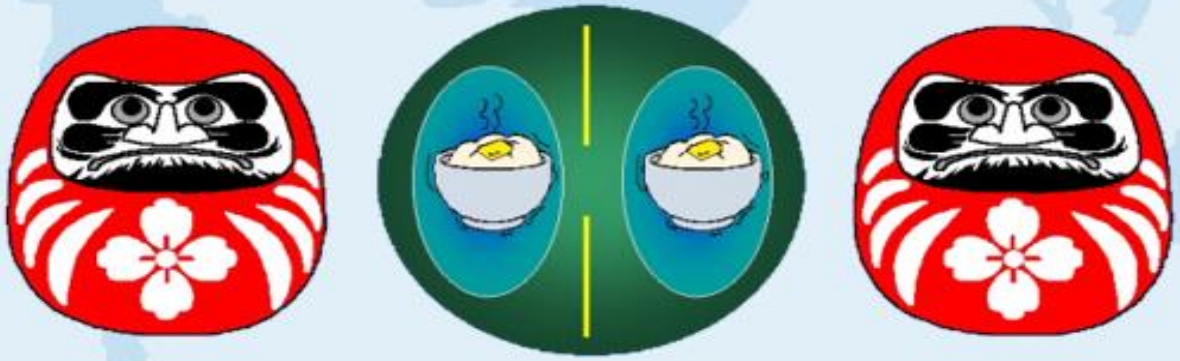
The Dining Philosophers Problem

- The situation of dining philosophers.



The Dining Philosophers Problem Semaphores

- Each chopstick is a semaphore.
- A philosopher **waits** for 2 chopsticks.
- A philosopher **signals** when finishing eating.



The Dining Philosophers Problem Solution

```
do{
    wait(chopstick_1);
    wait(chopsticks_2);
    philosopher eats;
    signal(chopstick_1);
    signal(chopsticks_2);
   philosopher thinks;
}while (TRUE);
```

Summary

- For the cooperating process that share data, **mutual exclusion** must be provided.
- One solution is the ensure that a **critical-section** of code is in use by only one process at a time.
- The main disadvantage of these user coded solutions is **busy waiting.**
- **Semaphores** overcome this difficulty. They can be used to solve various synchronization problems.

That is all for today!