CSE2005 - Operating Systems

Module 4 – L1
Process Synchronization

Dr. Rishikeshan C A
VIT Chennai

Outline

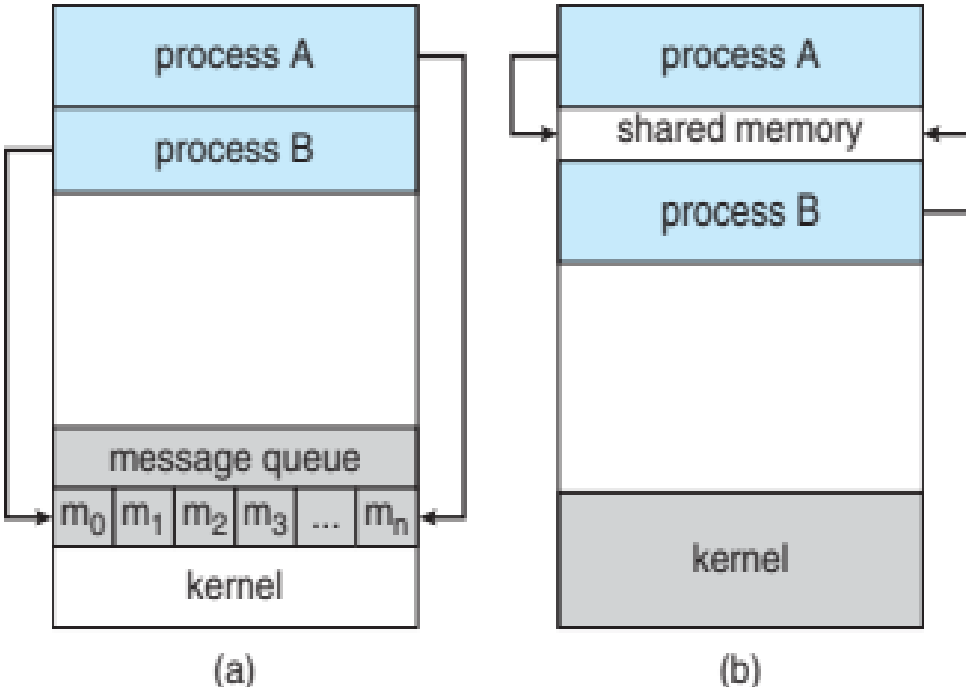
- Interprocess Communication
- Race condition
- Critical Section
- Semaphores

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Critical section

Motivating Scenario

shared variable

```
int counter=5;
```

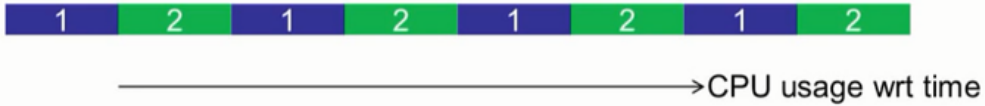
program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core



Critical section

Motivating Scenario

Shared variable

```
int counter=5;
```

program 0

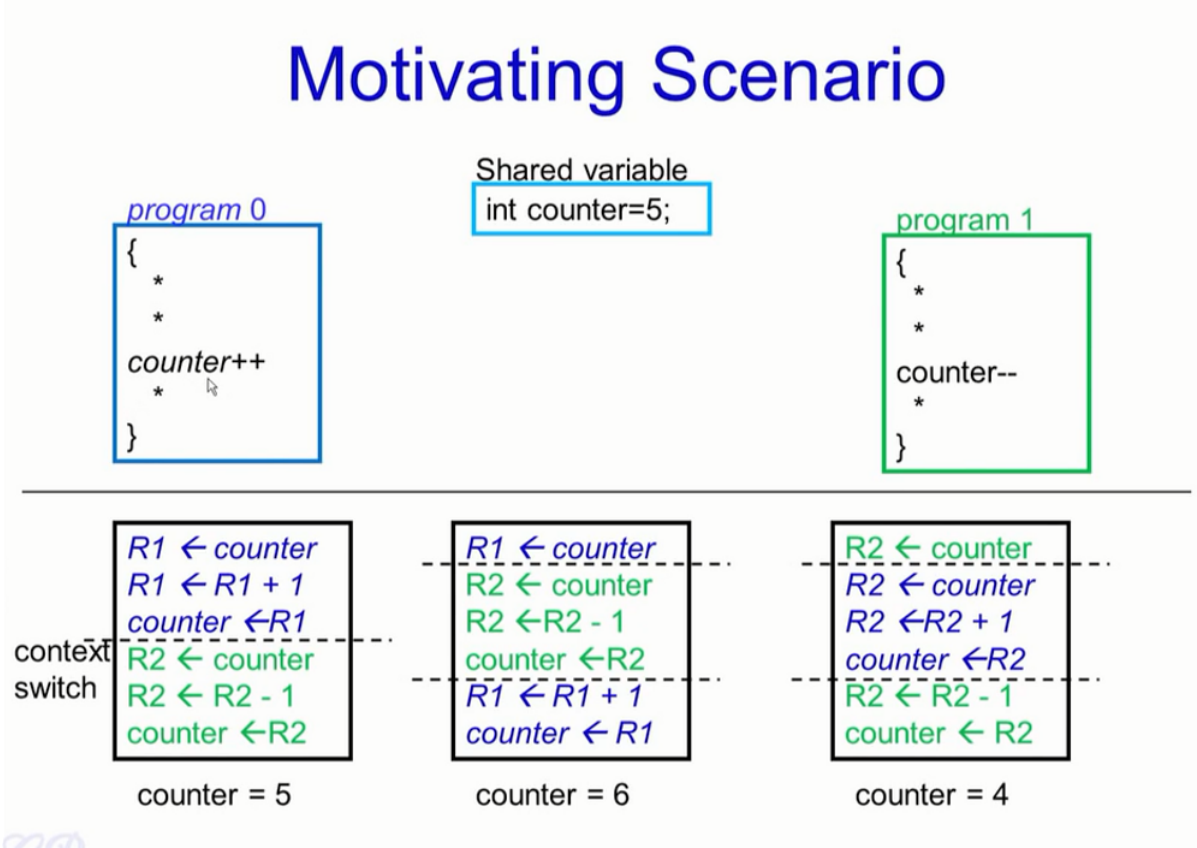
```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

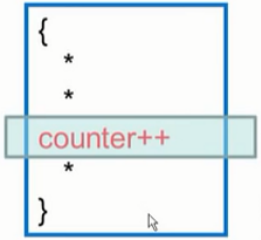
- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6

Critical section



Race Conditions

- Race conditions
  - A situation where several processes access and manipulate the same data (*critical section*)
  - The outcome depends on the order in which the access take place
  - Prevent race conditions by synchronization
    - Ensure only one process at a time manipulates the critical data



critical section

No more than one process should execute in critical section at a time

Race Conditions in Multicore

shared variable

```
int counter=5;
```

program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

Algorithm for Process P_i

```
do {

     while (turn == j);

          critical section

     turn = j;

          remainder section

 } while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
            critical section
    flag[i] = false;
            remainder section
 } while (true);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
     P_i enters CS only if:
     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

Locks and Unlocks

shared variable

```
int counter=5;
lock_t L;
```

program 0

```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

program 1

```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
```

```
release() {
    available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

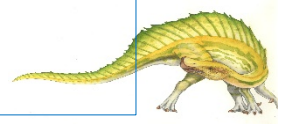
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the wait() operation

```
wait(S)
{
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

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



Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section

Classical synchronization problems

READERS-WRITERS PROBLEM

CRITICAL SECTION PROBLEM

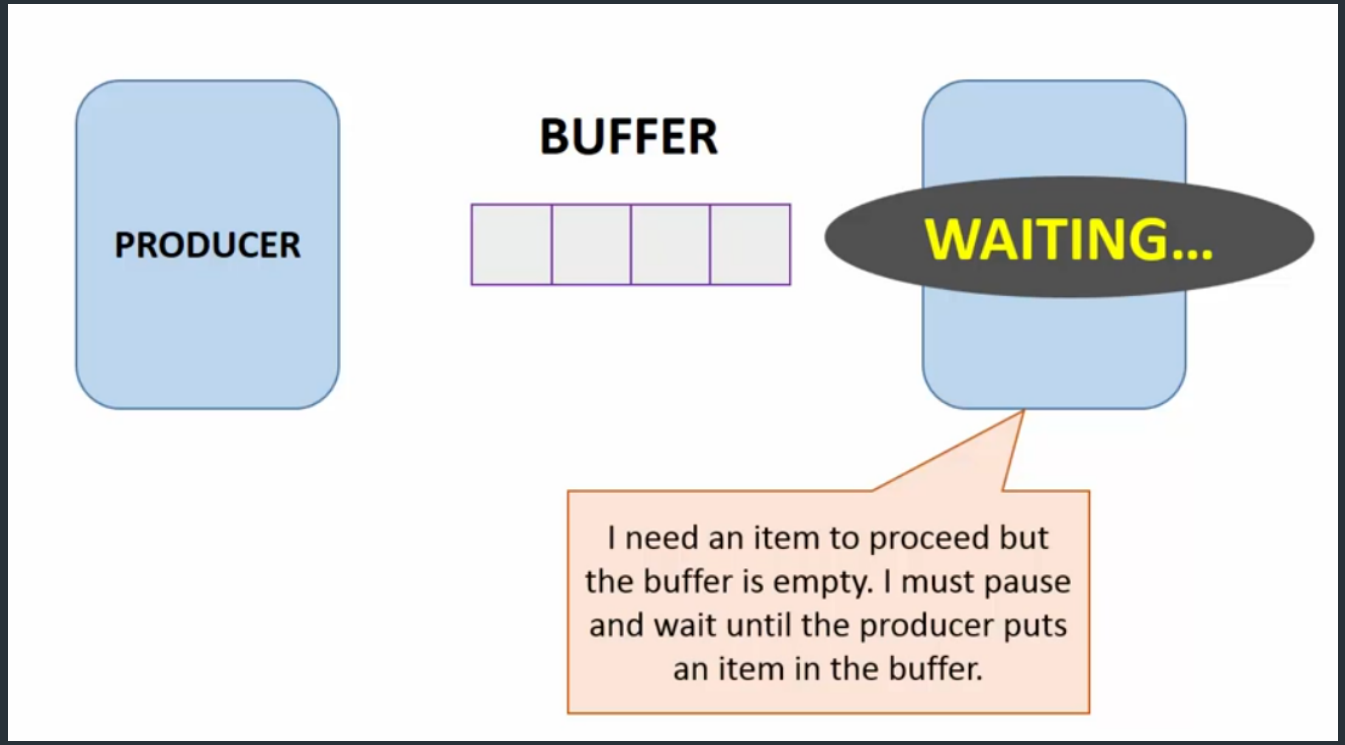
SLEEPING BARBER PROBLEM

CIGARETTE SMOKERS PROBLEM

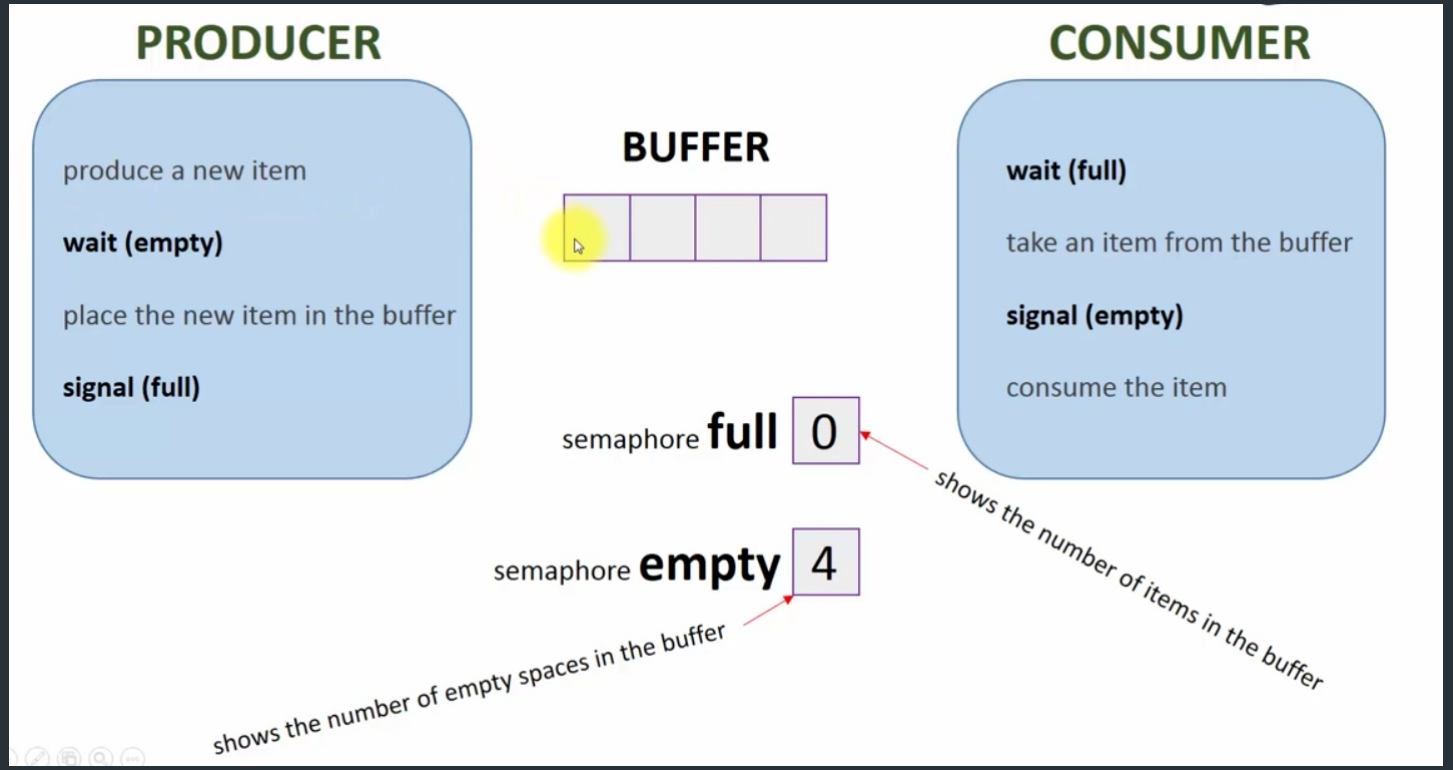
DINING PHILOSOPHERS PROBLEM

PRODUCER-CONSUMER PROBLEM

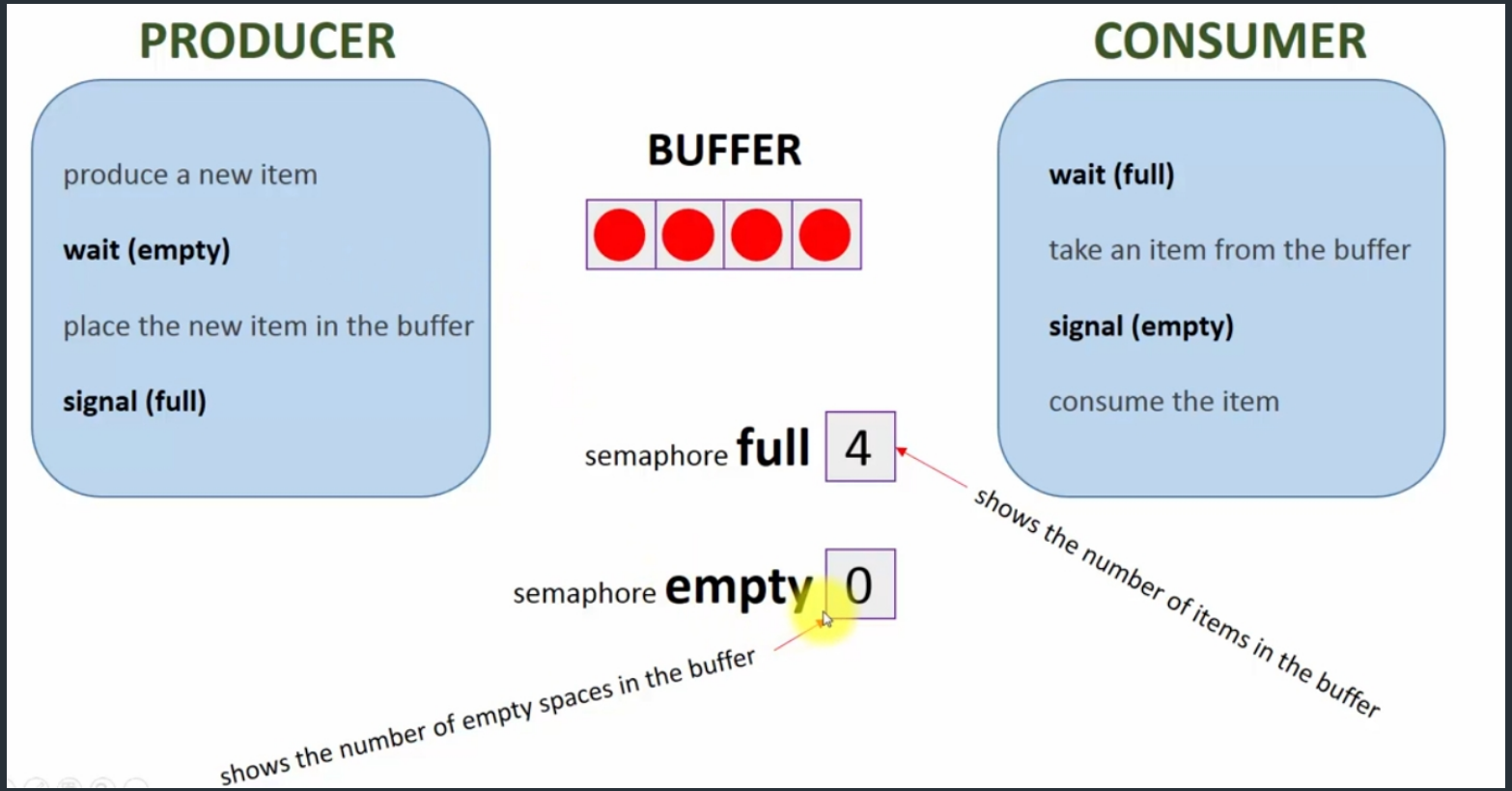
Producer consumer problem



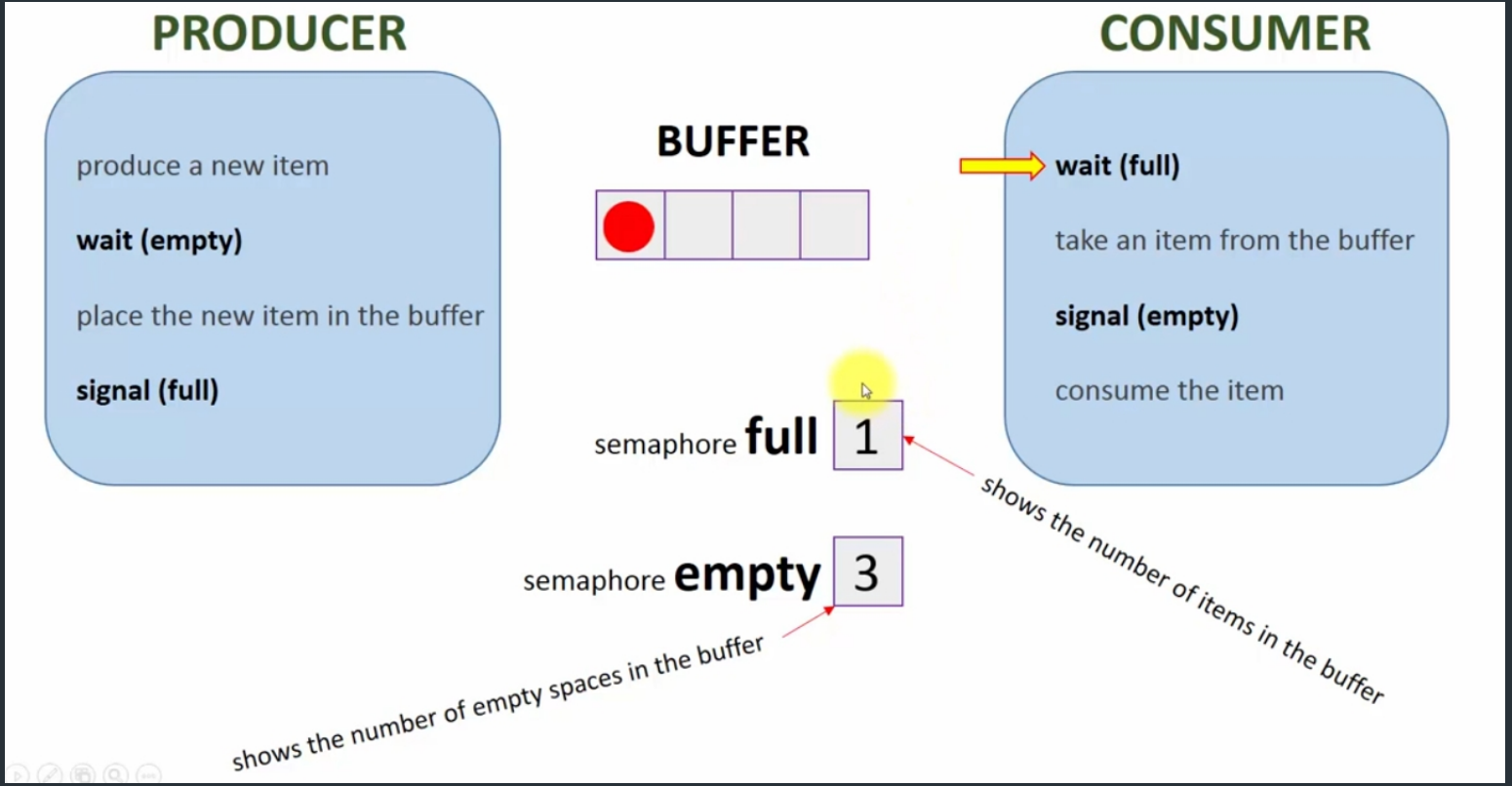
Producer consumer problem



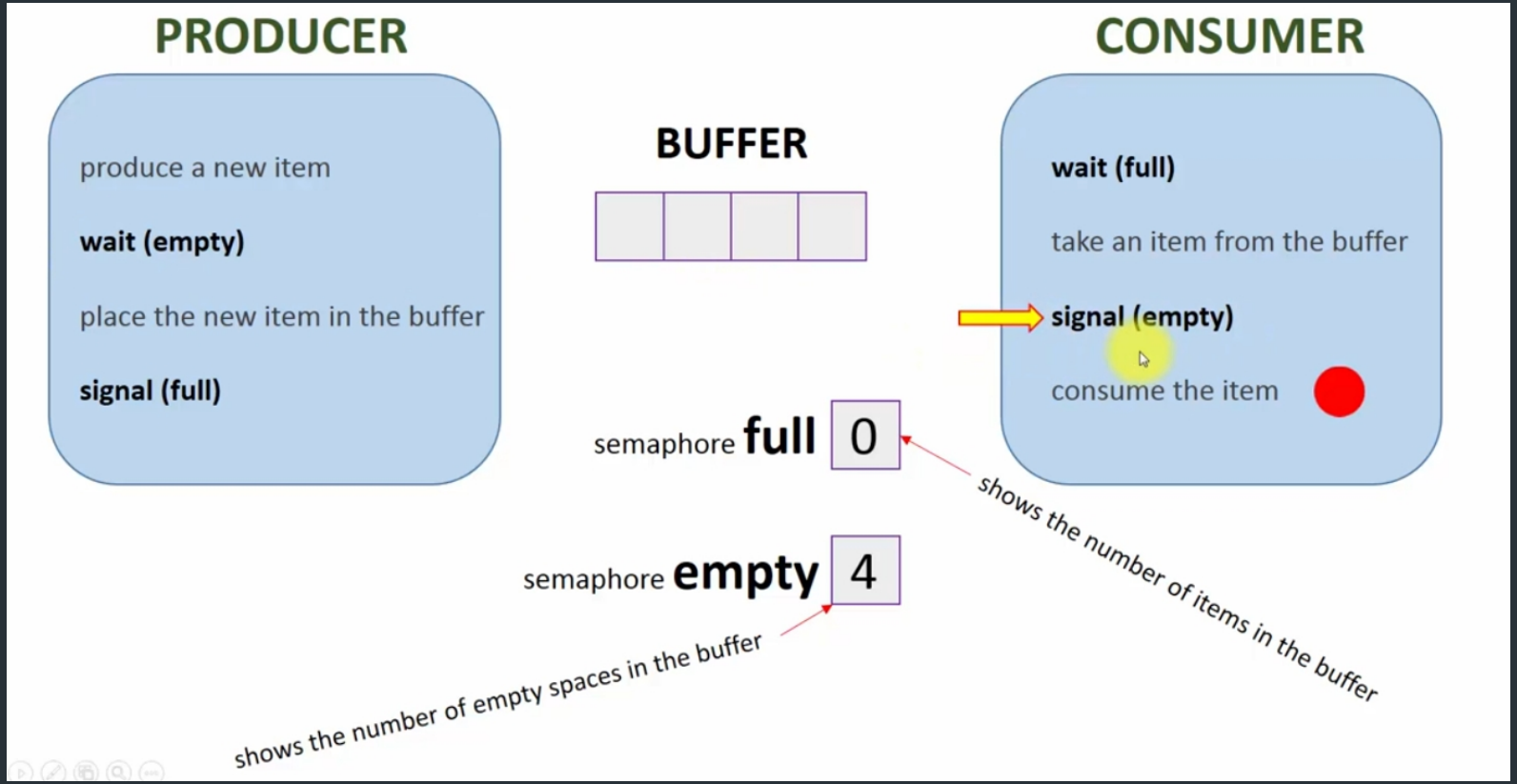
Producer consumer problem



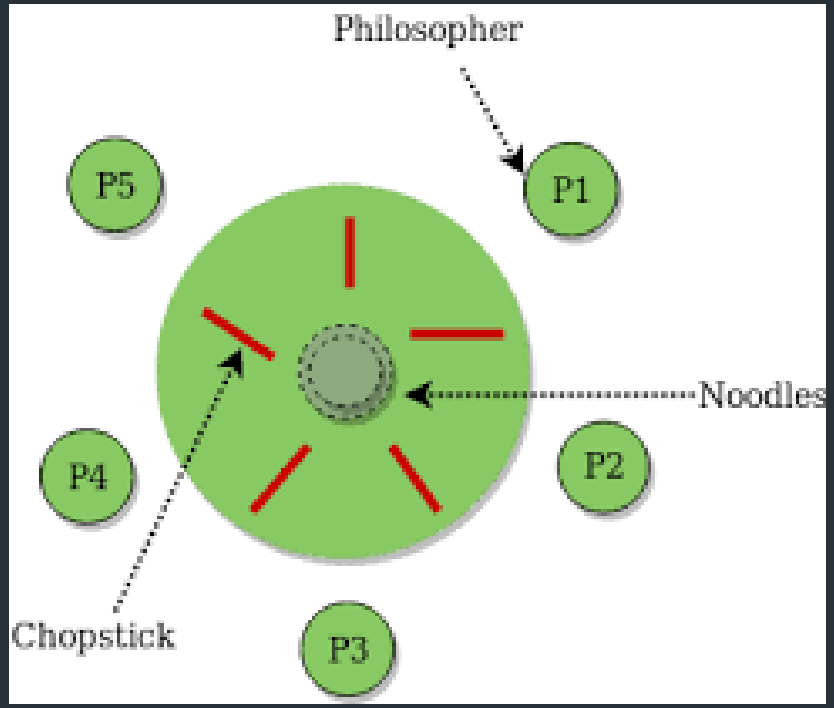
Producer consumer problem



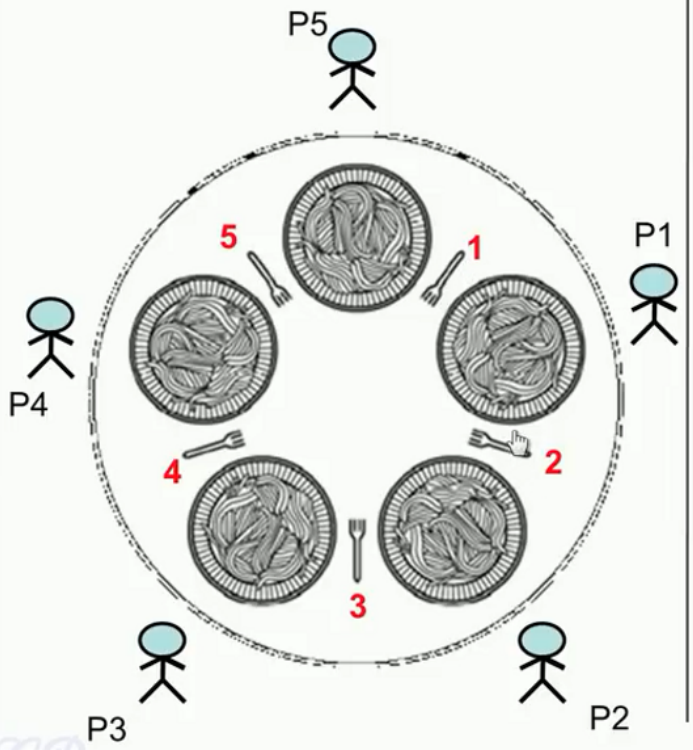
Producer consumer problem



Dining philosophers problem



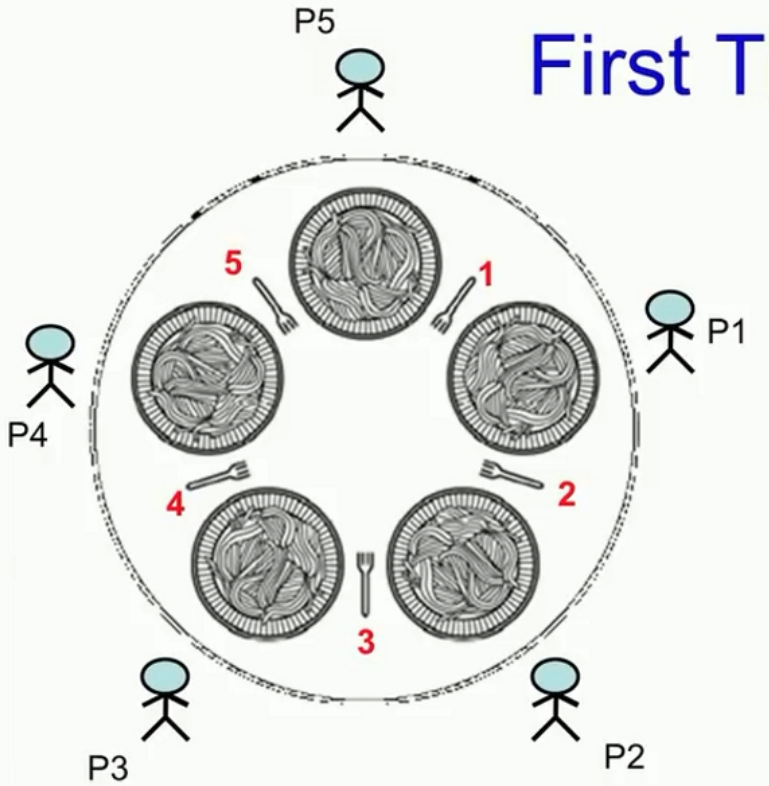
Dining Philosophers Problem



- **Philosophers either think or eat**
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.
- Problem Statement : Develop an algorithm where no philosopher starves.

Dining-Philosophers Problem

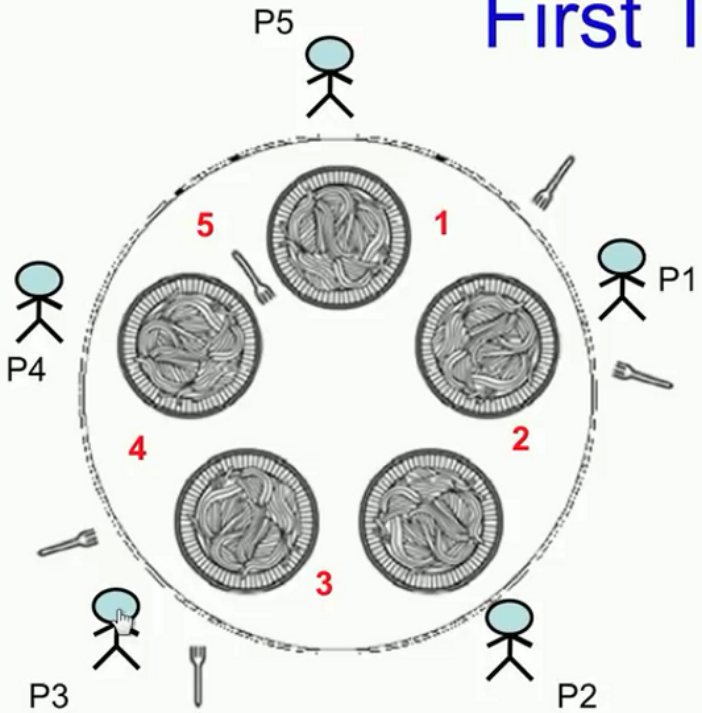
First Try




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Li);
      put_fork(Ri);
  }
}
```

Dining-Philosophers Problem

First Try



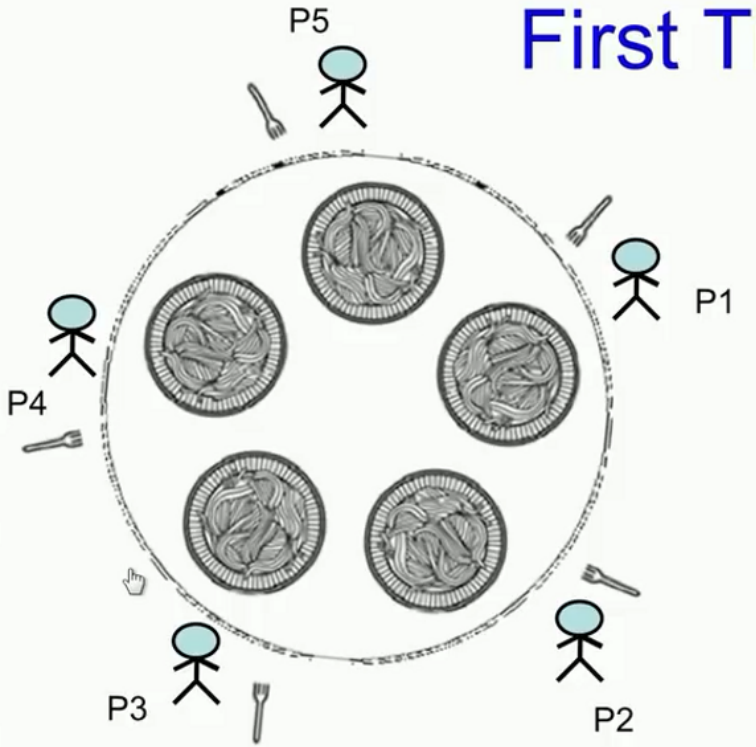

```
#define N 5

void philosopher(int i){
  while(TRUE){
     think(); // for some_time
     take_fork(Ri);
     take_fork(Li);
     eat();
     put_fork(Li);
     put_fork(Ri);
  }
}
```

What happens if only philosophers P1 and P3 are always given the priority?

P4, P5, and P2 starves… so scheme needs to be fair

Dining-Philosophers Problem

First Try



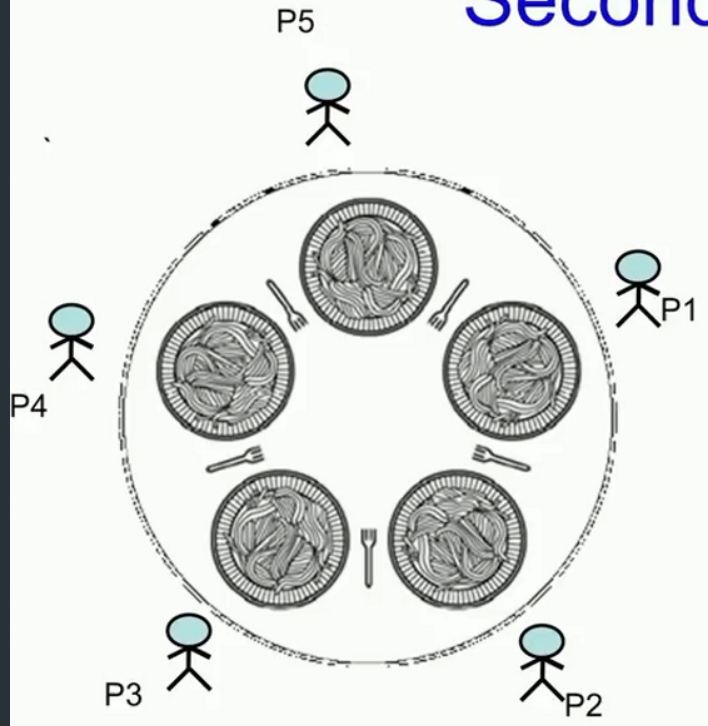

```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Li);
      put_fork(Ri);
  }
}
```

What happens if all philosophers decide to pick up their right forks at the same time?

Possible starvation due to deadlock

Dining-Philosophers Problem

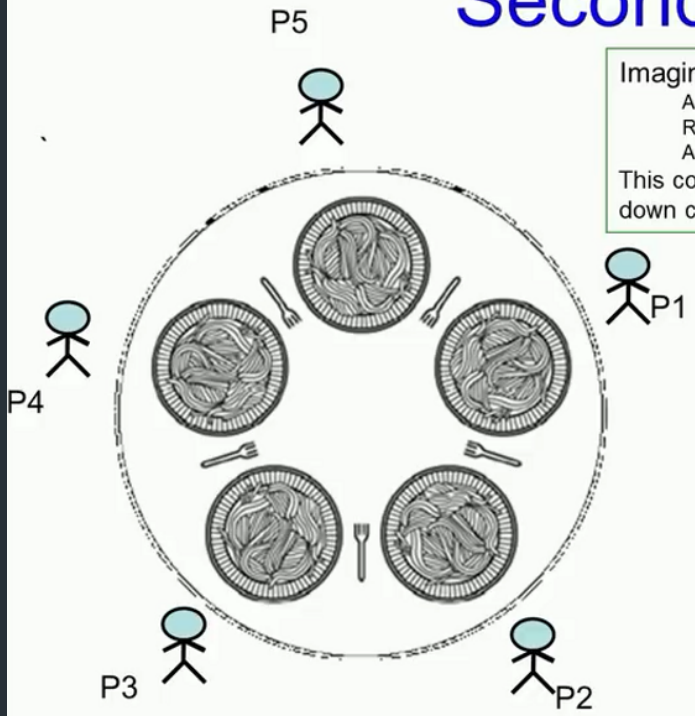
Second try




```
#define N 5

void philosopher(int i){
  while(TRUE){
     think();
     take_fork(Ri);
     if (available(Li){
       take_fork(Li);
       eat();
       put_fork(Ri);
       put_fork(Li);
     }else{
        put_fork(Ri);
        sleep(T)
     }
}
```

Dining-Philosophers Problem

Second try



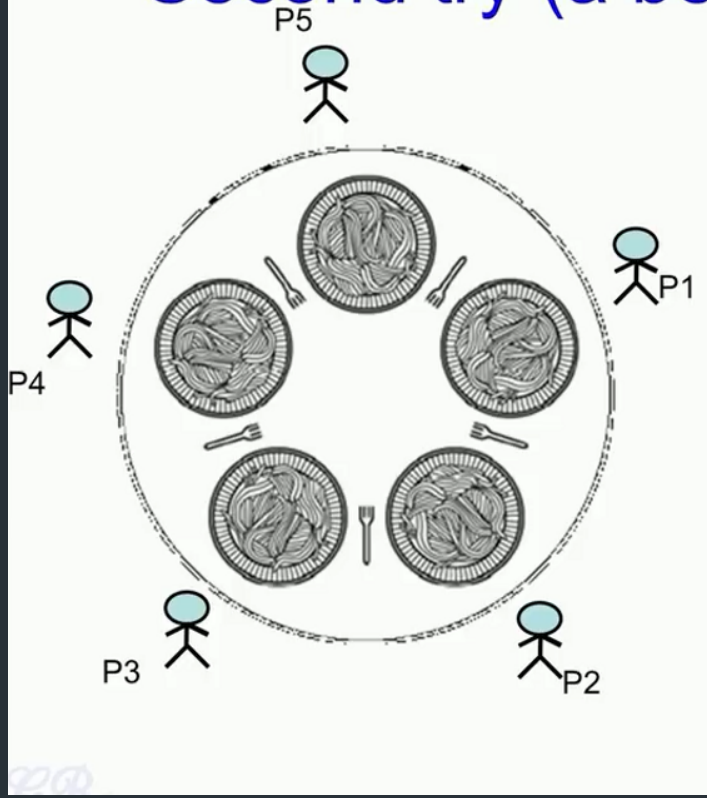
Imagine,
- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){
    think();
    take_fork(Ri);
    if (available(Li){
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
    }else{
       put_fork(Ri);
       sleep(T)
    }
}
```

Dining-Philosophers Problem

Second try (a better solution)




```
#define N 5

void philosopher(int i){
  while(TRUE){
     think();
     take_fork(Ri);
     if (available(Li){
        take_fork(Li);
        eat();
        put_fork(Li);
        put_fork(Ri);
     }else{
         put_fork(Ri);
         sleep(random_time);
     }
}
```

Dining-Philosophers Problem

Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
  - Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
  while(TRUE){
     think(); // for some_time
     lock(mutex);
     take_fork(Ri);
     take_fork(Li);
     eat();
     put_fork(Li);
     put_fork(Ri);
     unlock(mutex);
  }
}
```

Dining-Philosophers Problem

Solution with Semaphores

Uses N semaphores (s[1], s[2], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

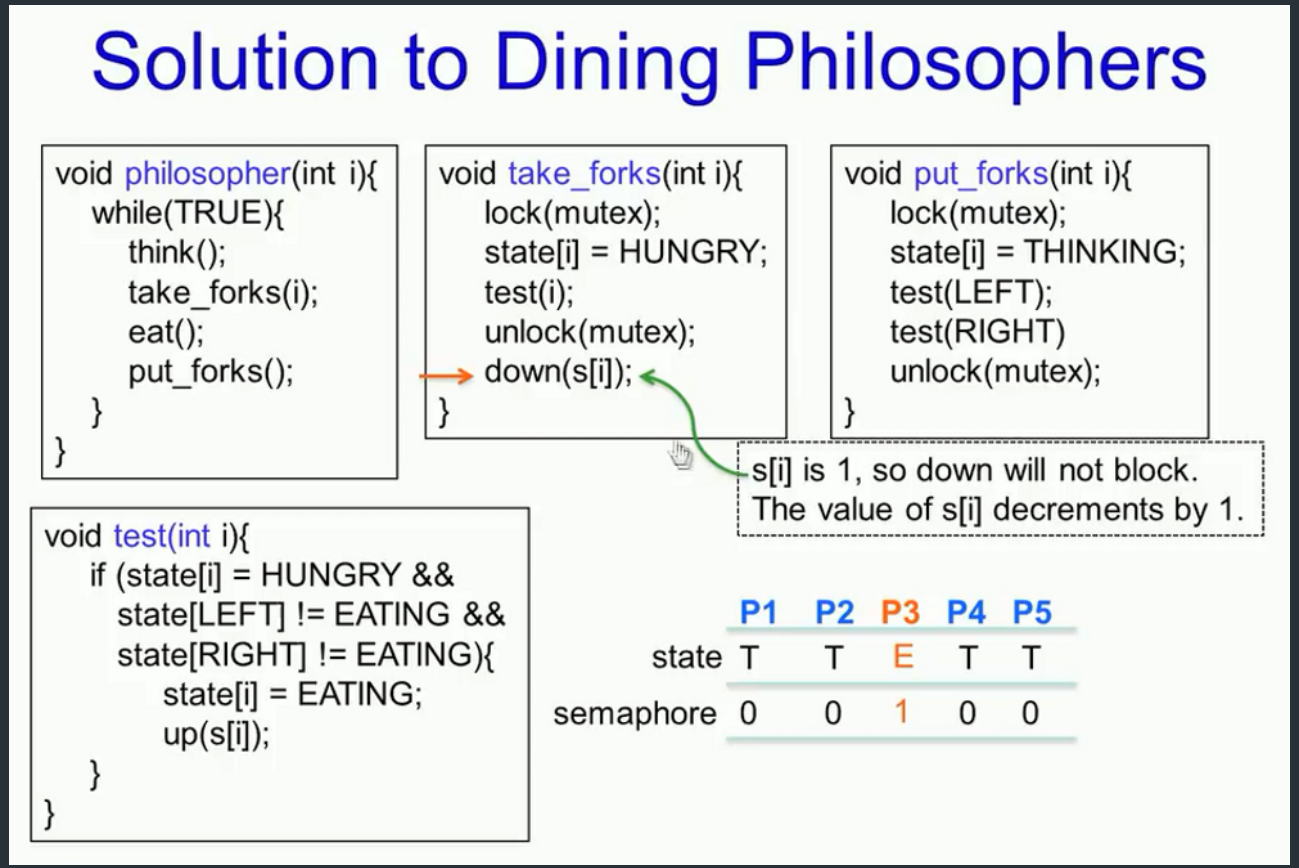
```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

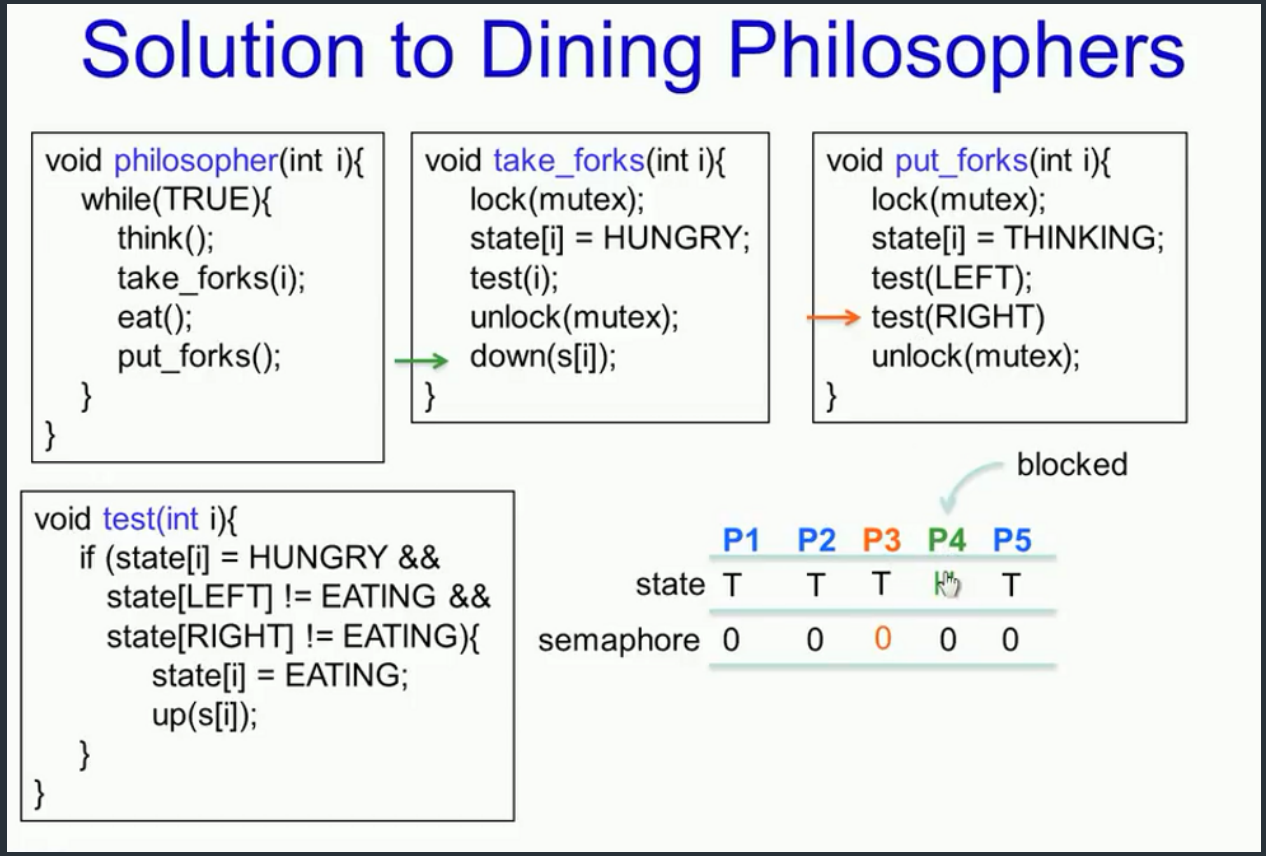
```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	H	T	T
semaphore	0	0	0	0	0

Dining-Philosophers Problem



Dining-Philosophers Problem



Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

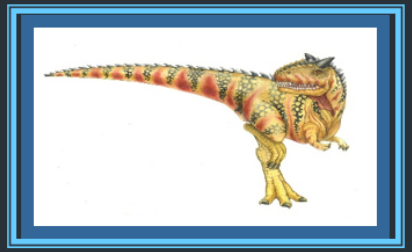
blocked

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

References

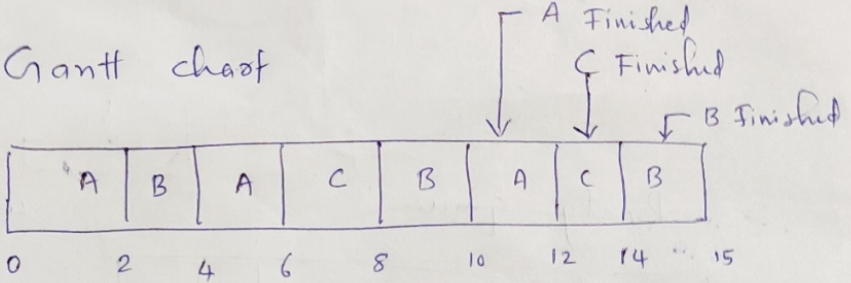
1. Abraham Silberschatz, Peter B. Galvin, Greg Gagne-Operating System Concepts, Wiley (2018).
2. Ramez Elmasri, A.Gil Carrick, David Levine, Operating Systems, A Spiral Approach - McGrawHill Higher Education (2010).
3. https://pdos.csail.mit.edu/6.828/2012/lec/l-lockfree.txt
4. https://en.wikipedia.org/wiki/Non-blocking_algorithm

Thank you



Process	A.T	B.T
A	0	6
B	1	5
C	3	4

Gantt chart



A C B

Ready queue order