0301304 FUNDAMENTAL OF OPERATIONG SYSTEM

UNIT	MODULES	WEIGHTAGE
1	INTRODUCATION TO OPERATING SYSTEM	20 %
2	PROCESS MANAGEMENT	20 %
3	PROCESS COMMUNICATION AND SYNCHRONIZATION	20 %
4	MEMORY MANAGEMENT	20 %
5	FILE MANAGEMENT , DISK MANAGEMENT , SECURITY AND PROTECTION	20 %

UNIT -3 Process Communication & Synchronization

- Introduction to Process
- Concurrent Processes
  - Process Communication
- Semaphores
- Solution of Classic Synchronization Problem using Semaphores
  - Solution of Dining Philosophers Problem

UNIT -3 Process Communication & Synchronization

- Deadlocks
  - Introduction
  - Defining Deadlocks
  - Conditions for Deadlocks
  - Dealing with deadlock
- Thread
  - Process and Thread
  - Multi-Tasking vs. Multi-Threading
  - Thread Control Block
  - Usage of Multi Thread
  - Types of Thread

Introduction to Process

- A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
- To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

Concurrent Processes

- Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance.
- Concurrent means something that happens at the same time as something else. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead of sequentially as they would have to be carried out by a single processor.
- Concurrent processing is sometimes said to be synonymous with parallel processing.

Processes Communication

- The processes are interacting or communicating by :
  - **Shared Variable**
  - **Message Passing**
- **Shared Variable**
- There is a s**hared variable through which they communicate,** that is, they are not aware of the existence of each other but coordinate with each other in the execution.
- In this way, there is **an indirect communication through shared memory.**

Process Communication

- **Message Passing**
  - There may be the case that the processes need to share data **not required for data access synchronization or control synchronization but for reading purpose**.
  - In this case **there is no need to maintain a shared data** as it incurs the cost of accessing.
  - The **processes can also communicate through messge and be explicitly aware of the existence of each other.**
  - This type of communication known as **message passing.**

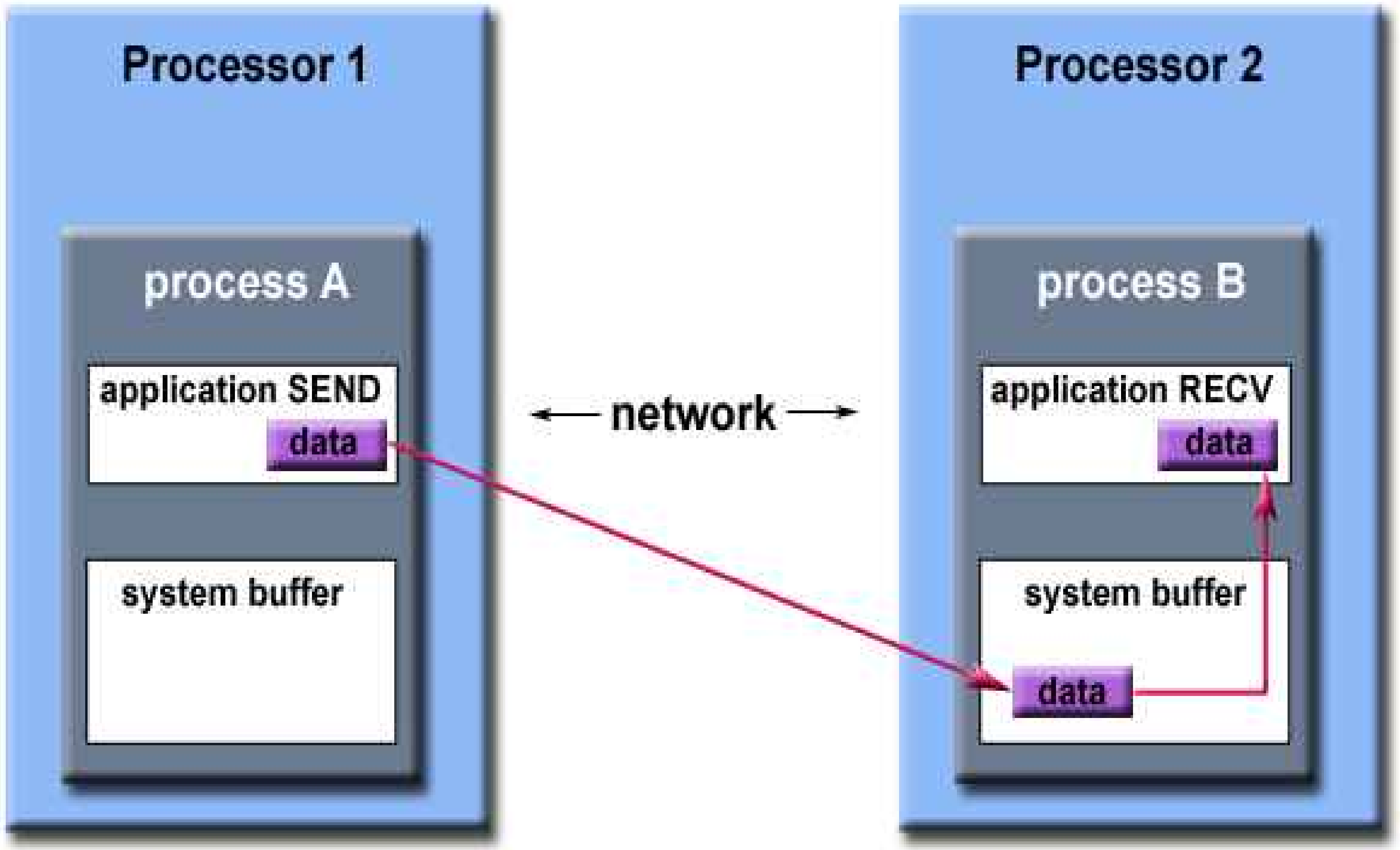
Processes Communication

- **Message Passing**
  - It is used when **processes explicity know each other and exchange message through system calls.**
  - One **system call is used for sending** the message and another for receiving it.
  - The **message has a fixed format** consisting of a message and the name of its sender or receiver process.
  - The process wishing to communicate a message with another **process copies the messge in its message structure with the specific name of the receiver.**

Processes Communication

- **Message Passing**
  - Similarly, **when the receiver receives the message, it copies the messge into its local variable and starts executing.**
  - Therefore, there is no requirement to update the message concurrenlty, there is no need to maintain a shared variable.
  - So **Message passing system is more appropriate.**

Message Passing Communication



Path of a message buffered at the receiving process

Processes Communication

- **Message Passing - Synchronization**
  - The **synchronization is also needed in a message passing system.**
  - When a **sender sends the message, it is not necessary that the receiver is ready to recieve it.**
  - In this case, the **sender will be blocked and the messge will be copied to a buffer**.
  - It is activated only when the intended r**eceiver will execute its system call for receiving the message.**

Processes Communication

- **Message Passing - Synchronization**
  - Similarly, when a process is ready to receive a message, it is not necessary that the sender be ready to send it.
  - In this case, the receiver will be blocked and activated only when the intended sender will send the message to it.
  - Thus there should be synchronization between the sender and the receiver process.

Processes Communication

- **Shared Variable**
  - It can be realized in both the cases of **communication and synchronization that a shared variable is necessary to have a proper synchronized execution of the process.**

Semaphores

- The **operation that can not be overlapped or interleaved with the execution of any other operations are known as individual or atomic operation**
- The semaphore is use to protect any resource such as global shared memory that needs to be accessed and updated by many processes simultaneously.
- **Semaphore acts as a guard or lock on the resource.**
- Whenever a process needs to access the resource, it first needs to take permission from the semaphore.
- If the resource is free, that is, if no other process is accessing or updating it, the process will be allowed, otherwise permission is denied.
- In case of denial, the requesting process needs to wait until semaphore permits it, that is, when the resource becomes free.

Semaphores

- The **semaphore is implememented as an integer variable**, say as **S**, and can be initialized with an positive integer value.
- The semaphore is accessed by only two operations known as **wait** and **signal** operations, denoted by
  - **Wait ---> P**
  - **Signal ---> V**
- When ever a **process tries to enter the critical** section, it needs to perform **wait operation.**
- The wait is an entry criterion according to the designed protocol.

Semaphores

```
Operation Wait

P(S)
{
     while (S <= 0);
     S = S - 1;
}

Operation Signal

V(S)
{
      S = S + 1;
}
```

```
do {

wait (Semaphore)
{

   Critical
   Section

}

signal (semaphore)
...
...
}
```

Semaphores

- Whenever a process tries to enter the critical section, it needs to perform wait operation. The wait is an entry criterion according to the designed protocol.
- If the CS is free or no other process is using it, then it is allowed, otherwise denied. The count of semaphore is decremented when a process accesses the available critical section; hence, the count of semaphore tells us the availabilityof the critical section.
- Initially, the count of semaphore is 1. If it is accessed by a process, then the count is decremented and becomes zero. Now, if another process tries to access the critical section, then it is not allowed to enter unless the semaphore value becomes greater than zero.
- When a process exits the critical section, it performs the signal operation, which is an exit criterion.

Semaphores

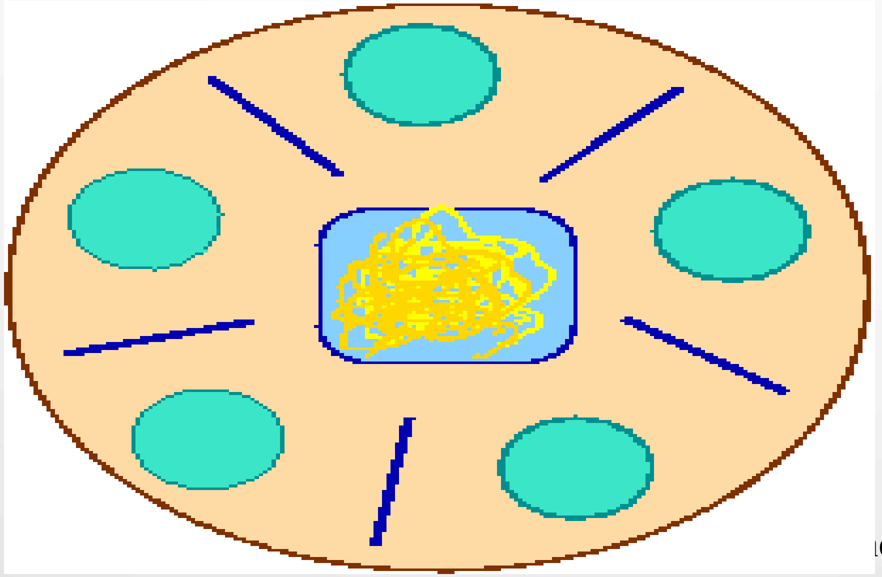
- The semaphore whose value P(S) is either zero or one is known as binary semaphore.
- There is one problem in the implementation of such a semaphore. When a process does not get access to the critical section, it loops continually waiting for it.
- This does not produce any result but consumes CPU cycles, thereby wasting the processor time. This busy waiting is a problem in a multi-programming system where processor time is shared among the processes.
- This type of semaphore is known as a spinlock, since the process spins while waiting for the lock.

Semaphores

- The semaphore whose value is either zero or one is known as **binary semaphore.**
- The type of semaphore that takes a value greater than one is known as **counting semaphore.**
- In a binary semaphore, the CS locked by process may be unlocked by any other process is called **mutex semaphore.**

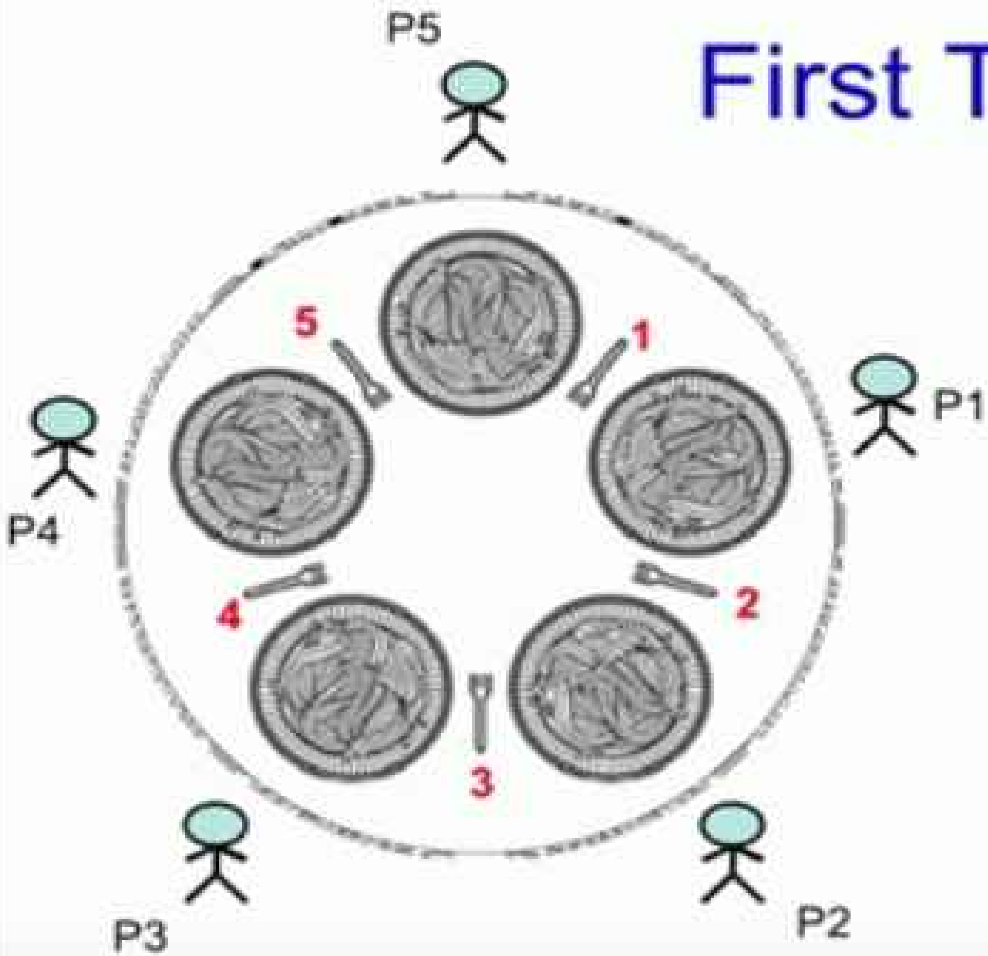
Dining Philosophers Problem

- The Dining Philosophers problems is a classic synchronization problem.
- There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more.
- In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.



Solution - 1

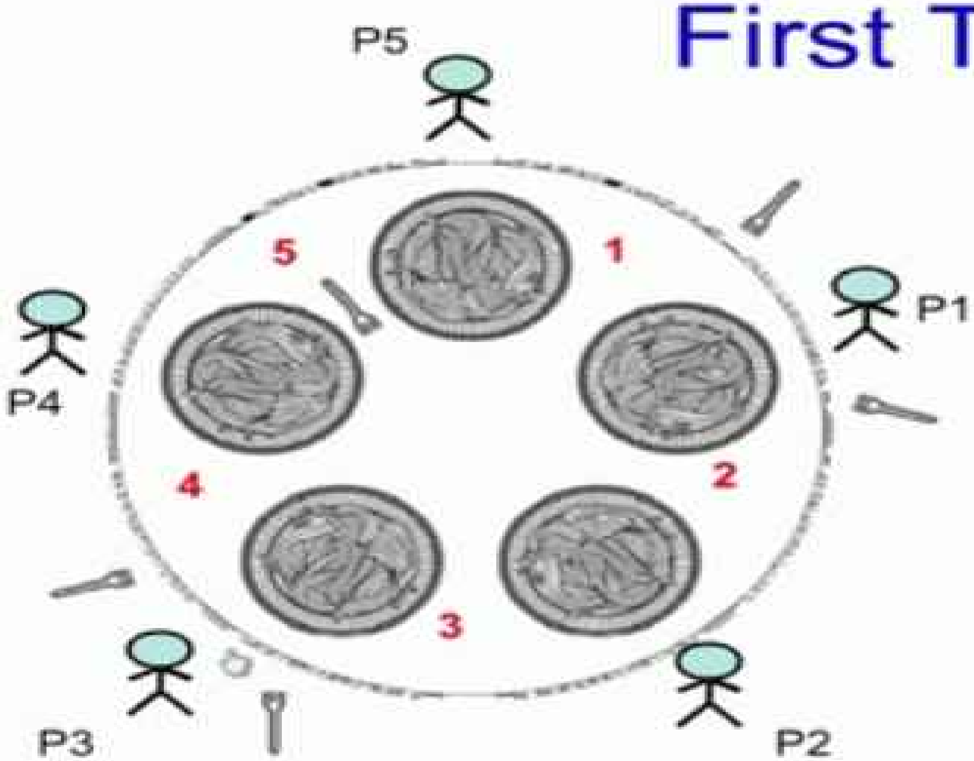
First Try




```
#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);
  }
}
```

Solution - 1

First Try



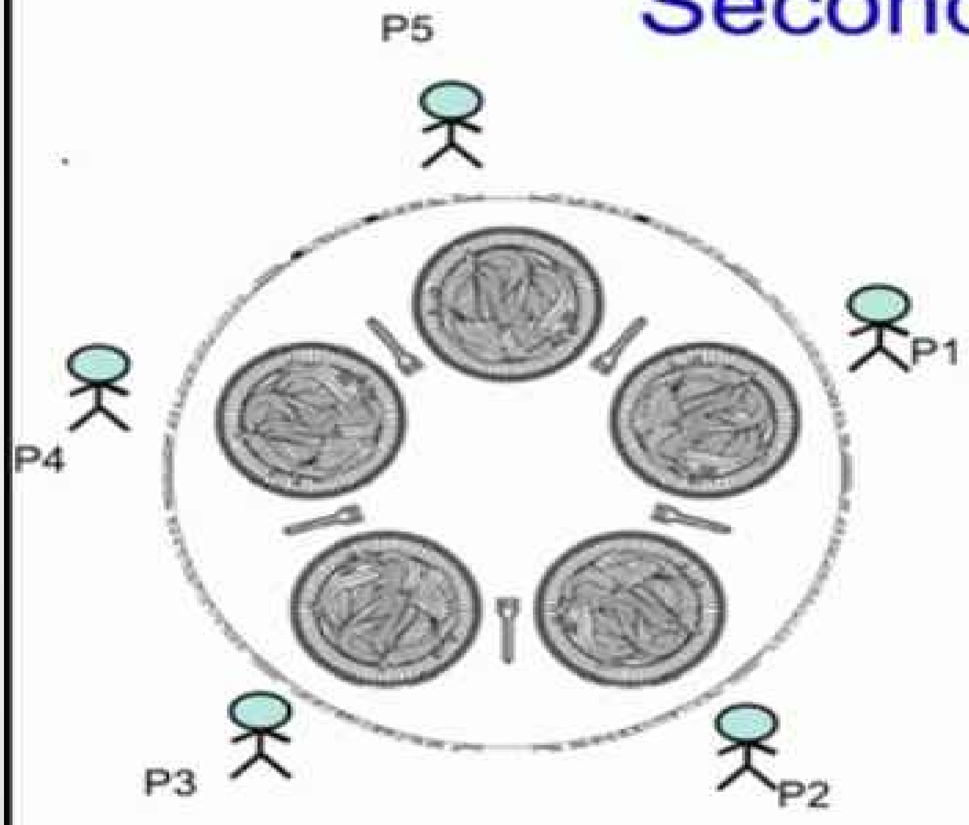

```
#define N 5

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

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

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

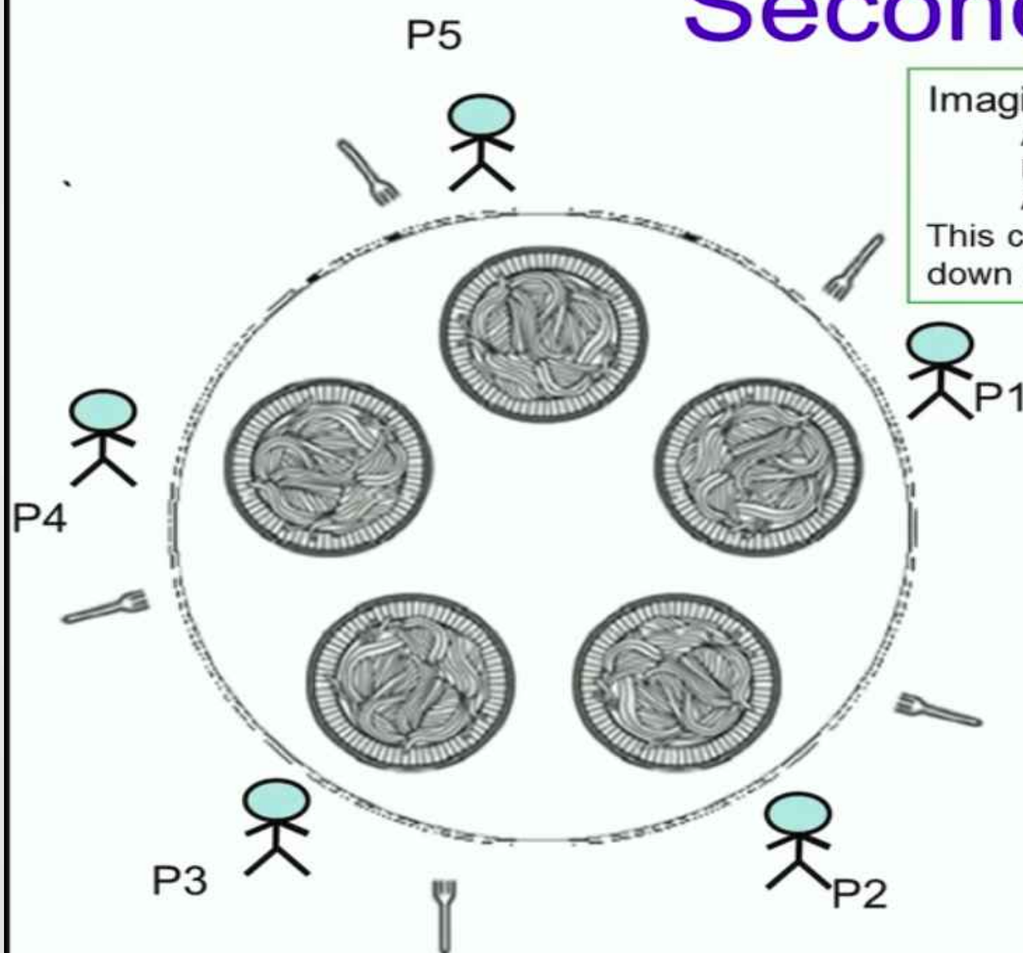
Solution - 2

Second try




```
#define N 5

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

Second try



Imagine,

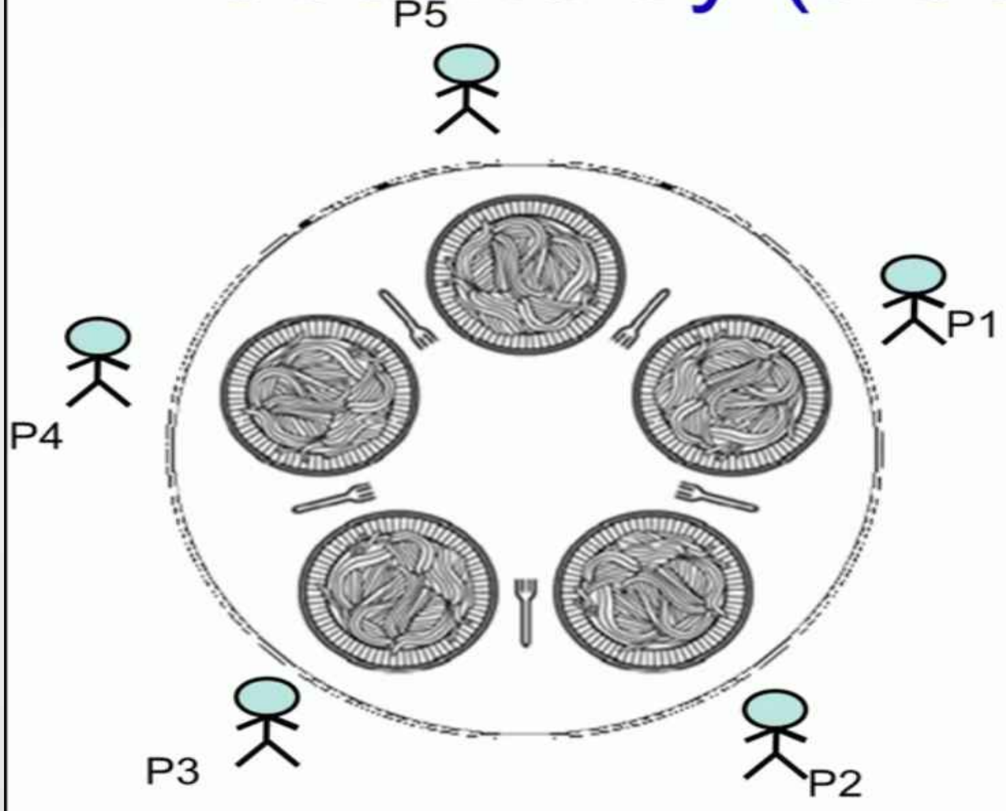
All philosophers start at the same time
Run simultaneously
And think for the same time

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)
  }
```

Solution - 2

Second try (a better solution)




```
#define N 5

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

Solution with Mutex

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(R_i);
      take_fork(L_i);
      eat();
      put_fork(L_i);
      put_fork(R_i);
      unlock(mutex);
  }
}
```

Solution with Semaphores

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]);
    }
}
```