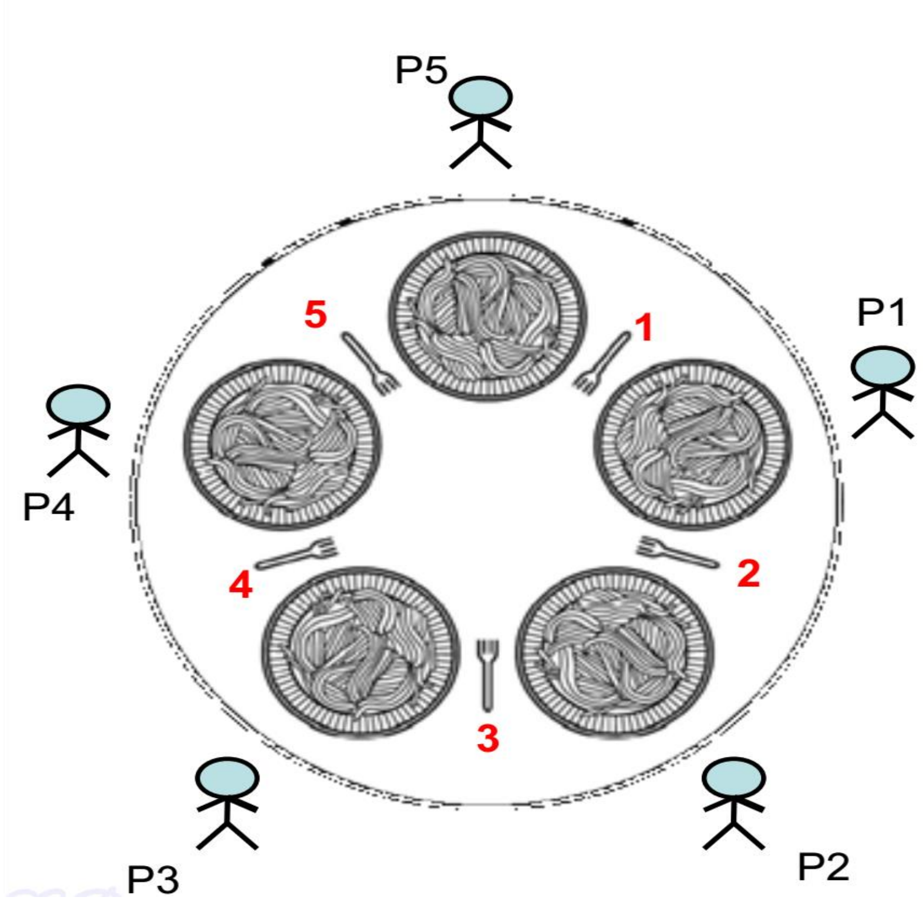
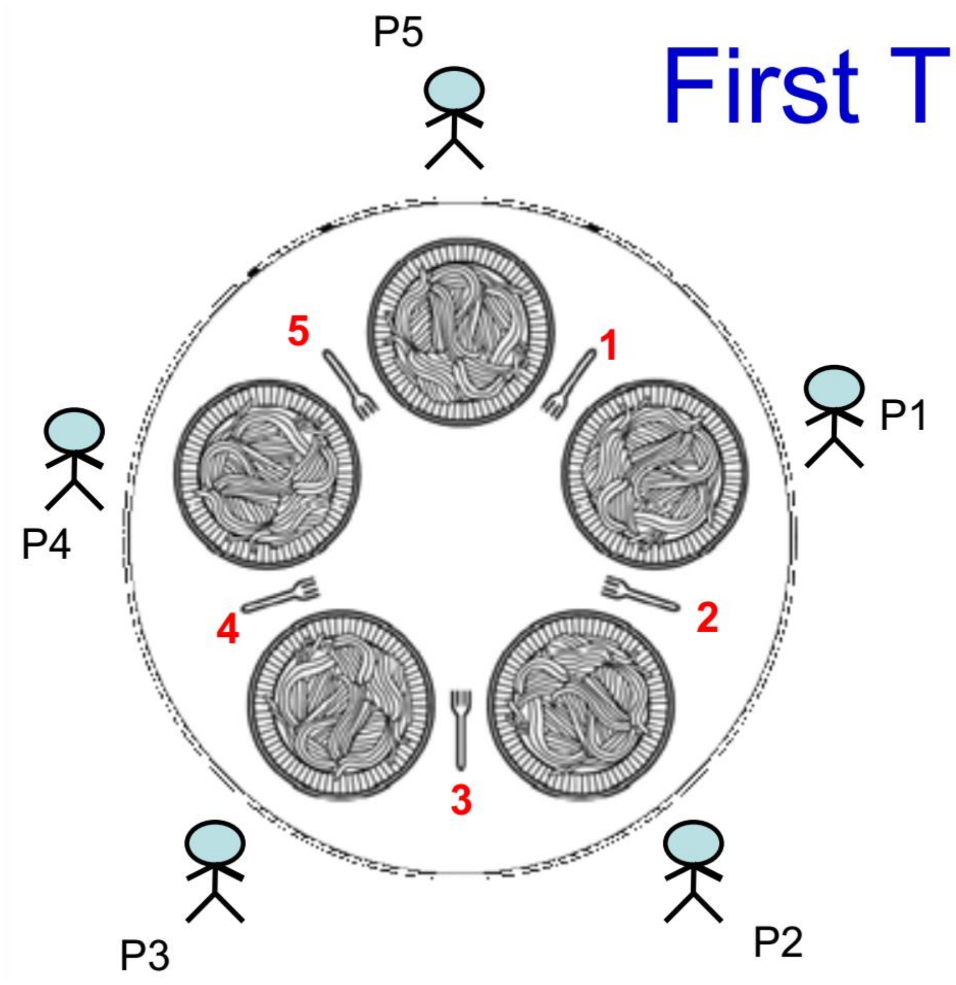
LECTURE 17
FEB 12, 2024

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.

First Try

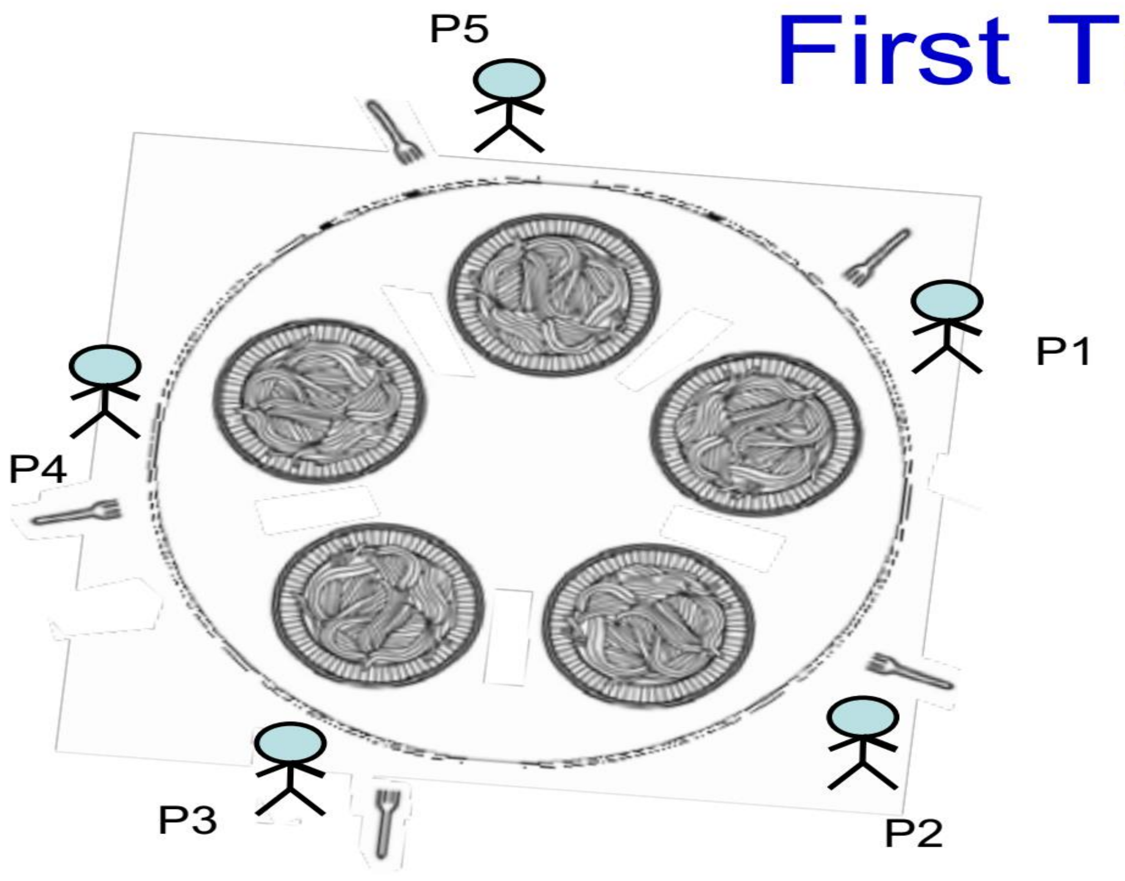


```
#define N 5

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

First Try

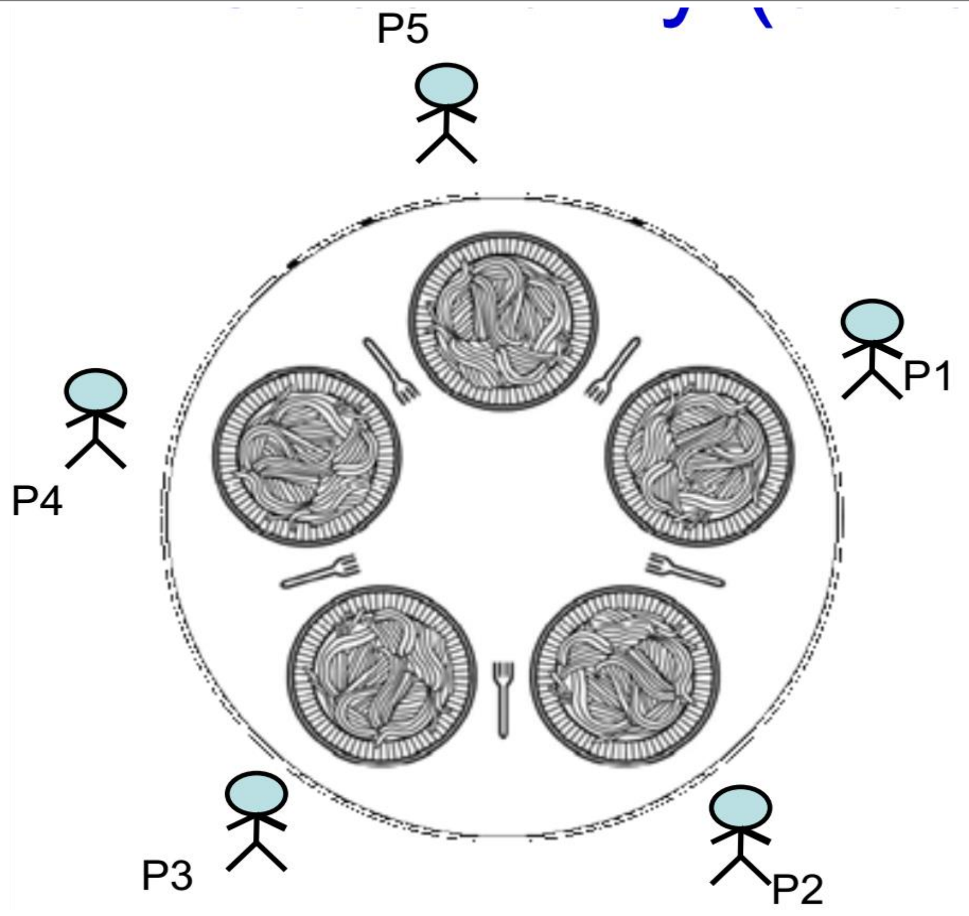


```
#define N 5

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

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

Possible starvation due to deadlock

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();
      }else{
          put_fork(R_i);
          sleep(T);
     }
}
```

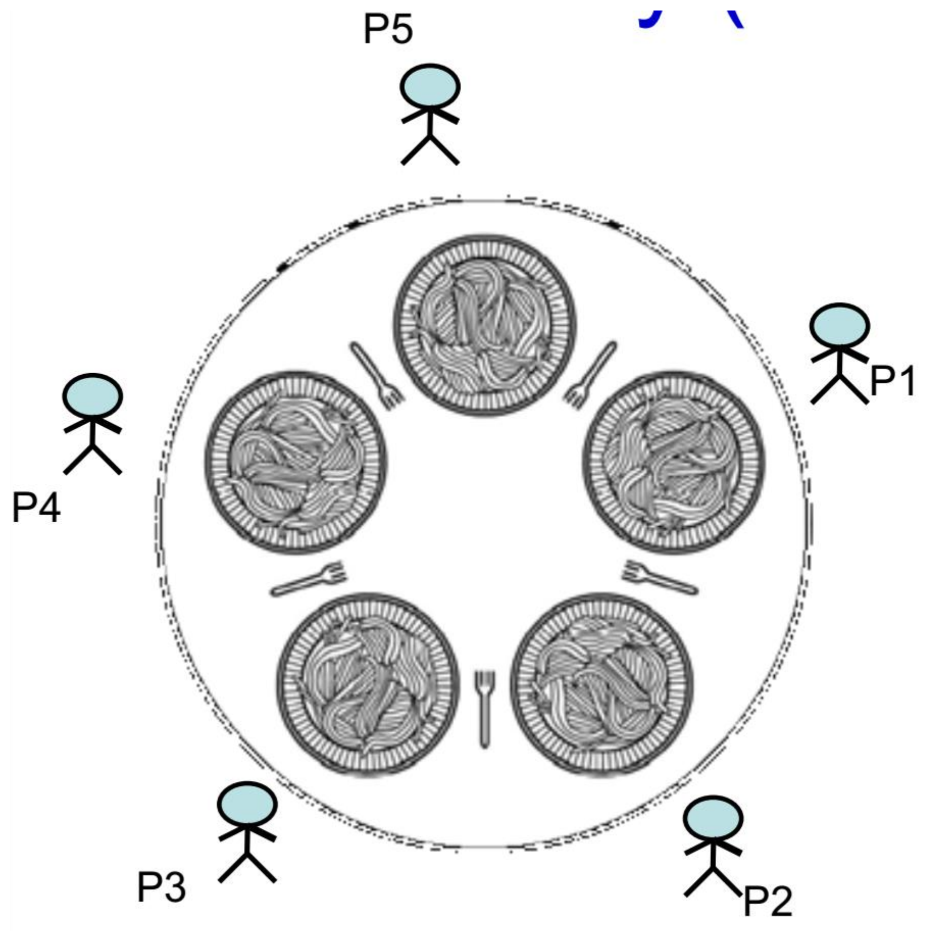
```
#define N 5

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

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.

A better solution



```
#define N 5

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

Solution with 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
      wait(mutex);
      take_fork(R_i);
      take_fork(L_i);
      eat();
      put_fork(R_i);
      put_fork(L_i);
      signal(mutex);
  }
}
```

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]);
    }
}
```

```
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

```
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);
}
```

s[i] is 1, so down will not block.
The value of s[i] decrements by 1.

```
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	E	T	T
semaphore	0	0	0	0	0

```
   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	E	~~T~~H	T
semaphore	0	0	0	0	0

```
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	E / T	H / E	T
semaphore	0	0	0	0	0

Consider the following scenario. A town has a very popular restaurant. The restaurant can hold N diners. The number of people in the town who wish to eat at the restaurant, and are waiting outside its doors, is much larger than N. The restaurant runs its service in the following manner. Whenever it is ready for service, it opens its front door and waits for diners to come in. Once N diners enter, it closes its front door and proceeds to serve these diners. Once service finishes, the backdoor is opened and the diners are let out through the backdoor. Once all diners have exited, another batch of N diners is admitted again through the front door. This process continues indefinitey. The restaurant does not mind if the same diner is part of multiple batches.

We model the diners and the restaurant as threads in a multithreaded program. The threads must be synchronized as follows. A diner cannot enter until the restaurant has opened its front door to let people in. The restaurant cannot start service until N diners have come in. The diners cannot exit until the back door is open. The restaurant cannot close the backdoor and prepare for the next batch until all the diners of the previous batch have left.

Below is given unsynchronized pseudocode for the diner and restaurant threads. Your task is to complete the code such that the threads work as desired. Please write down the complete synchronized code of each thread in your solution.

You are given the following variables (semaphores and initial values, integers) to use in your solution. The names of the variables must give you a clue about their possible usage. You must not use any other variable in your solution.

```
sem (init to 0): entering_diners, exiting_diners, enter_done, exit_done
sem (init to 1): mutex_enter, mutex_exit
Integer counters (init to 0): count_enter, count_exit
```

All changes to the counters and other variables must be done by you in your solution. None of the actions performed by the unsynchronized code below will modify any of the variables above.

(a) Unsynchronized code for the restaurant thread is given below. Add suitable synchronization in your solution in between these actions of the restaurant.

```
openFrontDoor()
closeFrontDoor()
serveFood()
openBackDoor()
closeBackDoor()
```

(b) Unsynchronized code for the diner thread is given below. Add suitable synchronization in your solution around these actions of the diner.

```
enterRestaurant()
eat()
exitRestaurant()
```

restautant thread:

```
openFrontDoor()
do N times: up(entering_diners)
down(enter_done)

closeFrontDoor()
serveFood()

openBackDoor()
do N times: up(exiting_diners)
down(exit_done)
closeBackDoor()
```

diner thread:

```
down(entering_diners)
enterRestaurant()

down(mutex_enter)
  count_enter++
  if(count_enter == N)
    up(enter_done)
    count_enter = 0
    }
up(mutex_enter)

eat()

down(exiting_diners)
exitRestaurant()

down(mutex_exit)
  count_exit++
  if(count_exit == N) {
    up(exit_done)
    count_exit = 0
    }
up(mutex_exit)
```