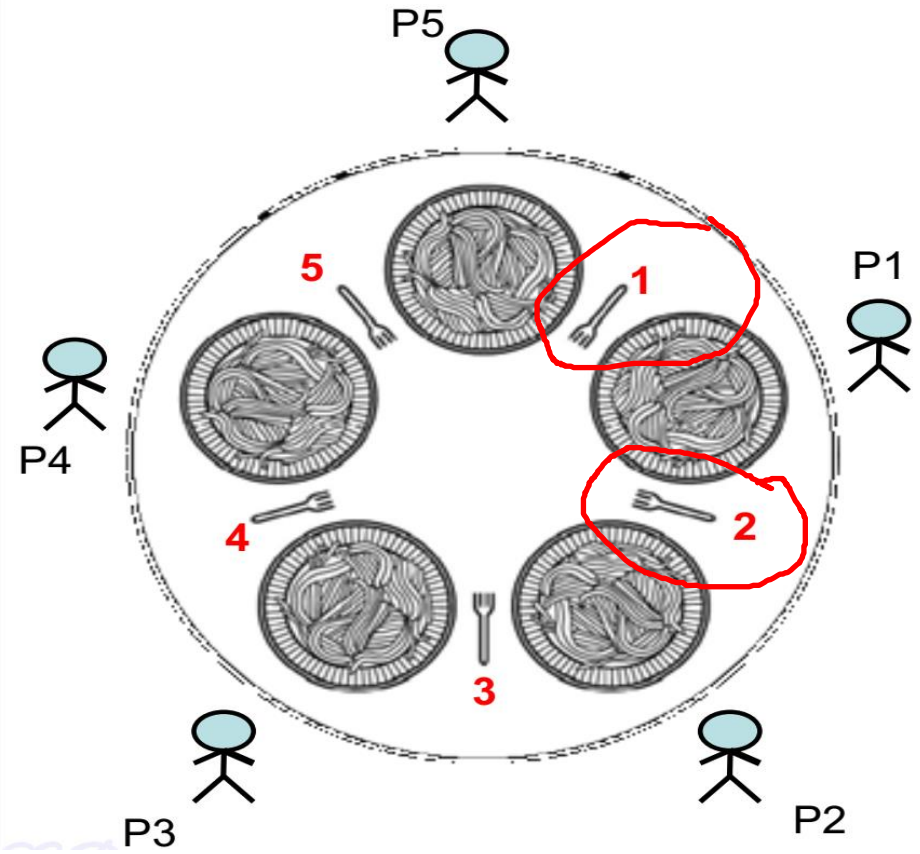
CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

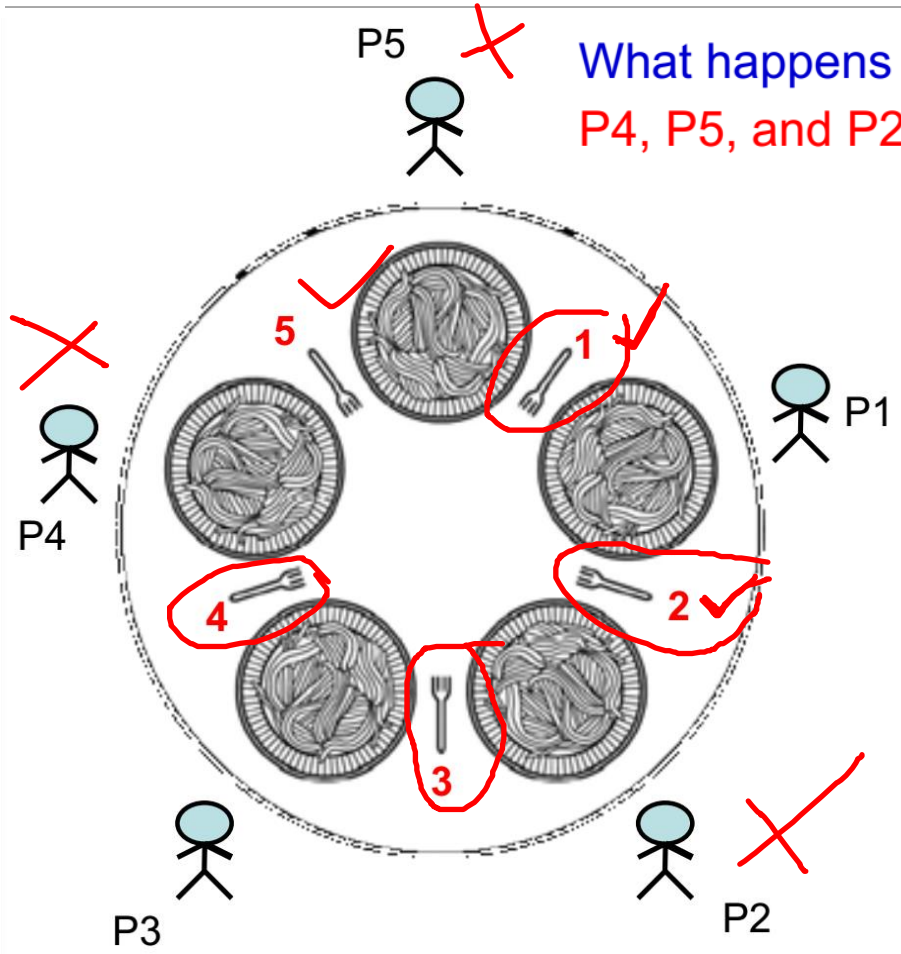
Materials in these slides have been borrowed from textbooks and existing operating systems courses

Feb 12th, 2021
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

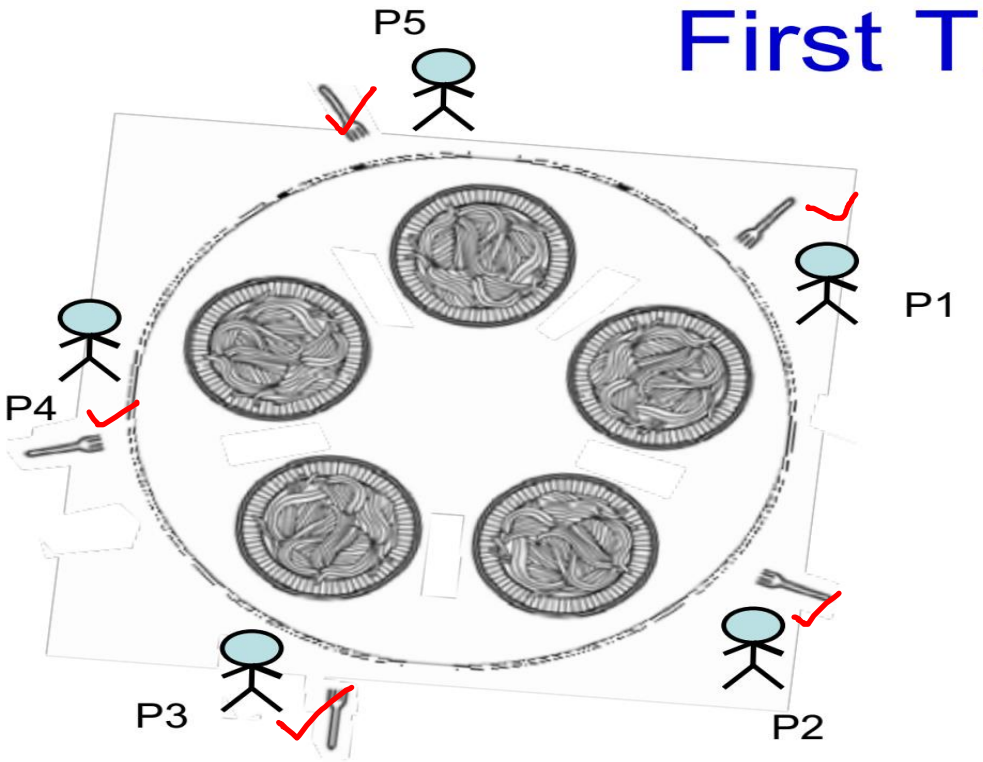


What happens if only philosophers P1 and P3 are always given the priority?

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

```
#define N 5

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

First Try

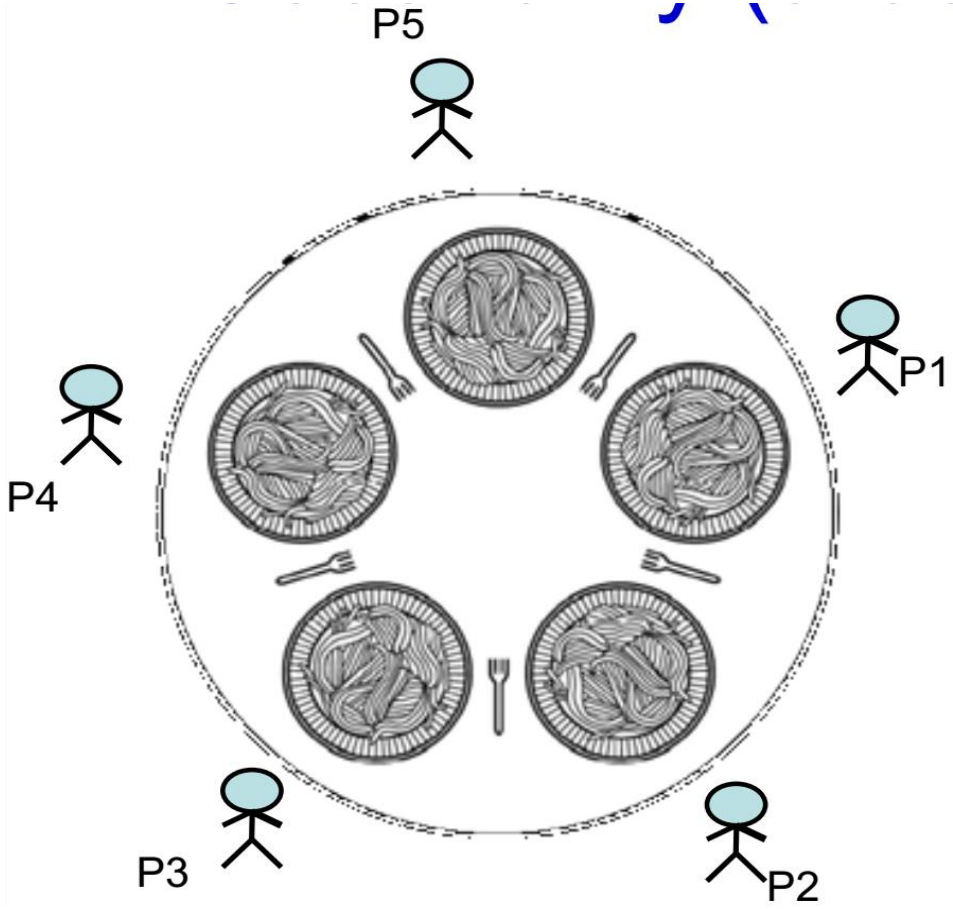



```
#define N 5

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

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

if fTrue

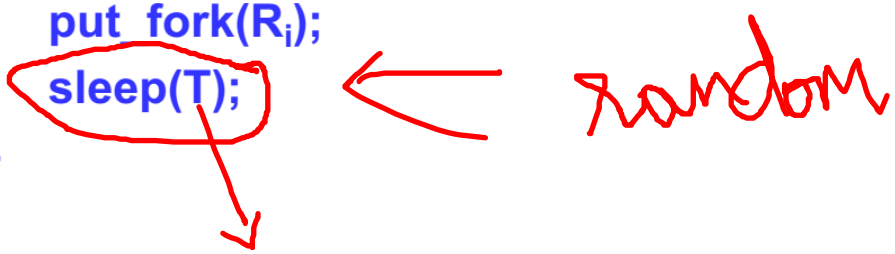
T - fixed

```
#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(T);
      }
}
```

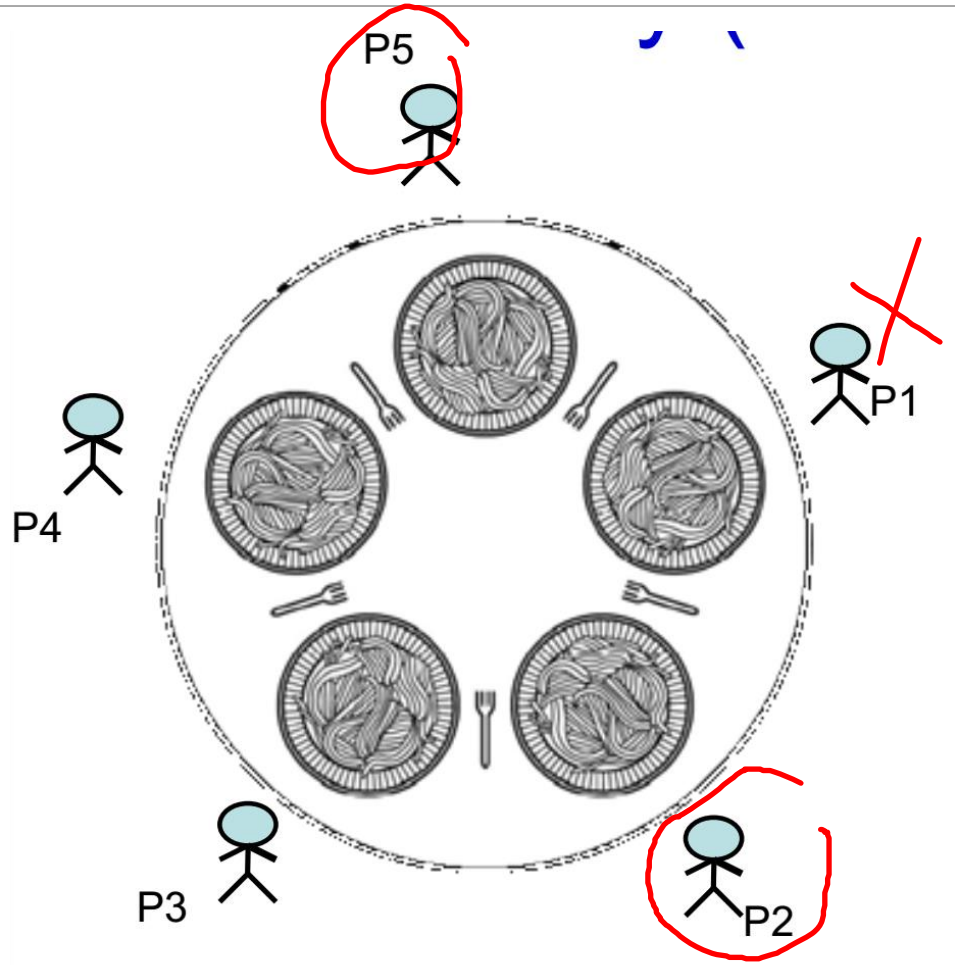
Imagine,

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

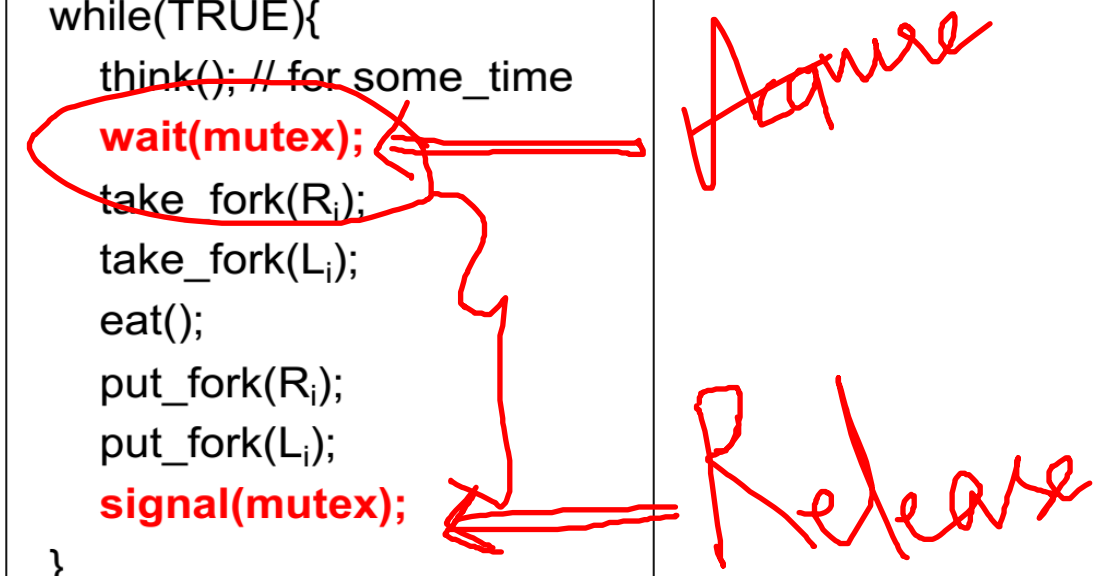



```
#define N 5

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 to 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(Ri);
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
      signal(mutex);
  }
}
```



Solution with Semaphores

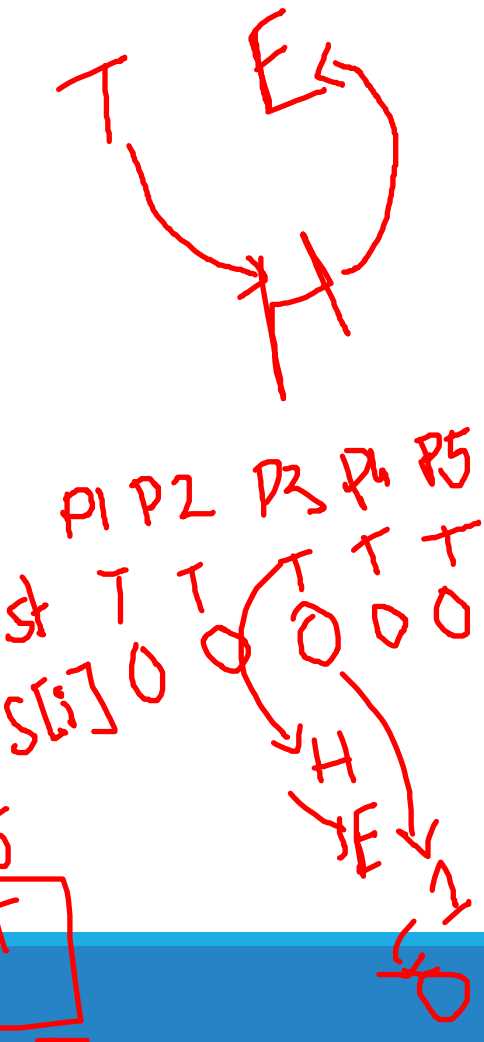
Uses N semaphores (s[1], s[2], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
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	H	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	H	T
semaphore	0	0	0	0	0

T → H → E

```
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	H	T
semaphore	0	0	0	0	0