OPERATING SYSTEM: CSET209

READERS-WRITERS PROBLEM

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Table 1

CASES	Process 1	Process 2	Allowed/Not allowed
Case 1	Writer	Writer	Not allowed
Case 2	Reader	Writer	Not allowed
Case 3	Writer	Reader	Not allowed
Case 4	Reader	Reader	Allowed

READERS-WRITERS PROBLEM

- The structure of a writer process

```
do {
      wait(rw_mutex);
         ...
      /* writing is performed */
         ...
   signal(rw_mutex);
} while (true);
```

The structure of a reader process

```
do {
      wait(mutex);
      read_count++;
      if (read_count == 1)
         wait(rw_mutex);
      signal(mutex);
         ...
      /* reading is performed */
         ...
      wait(mutex);
      read count--;
      if (read_count == 0)
      signal(rw_mutex);
      signal(mutex);
} while (true);
```

READERS-WRITERS PROBLEM (PRACTICE QUESTION)

```
Reader and writer problem
int R=0, W=0;
semaphore mutex=1;
void reader(void)
{
 L1: down(mutex);
   if(w==1)
    {
      [          ]  → 1
      goto L1;
    }
  Else
    {
     R=R+1;
     [          ]  → 2
    }
   DB
   down(mutex);
   R=R-1;
   up(mutex);
}
```

```
void writer(void)
{
  L2 : down(mutex);
  If ( [          ] ) → 3
  {
     up(mutex);
     goto L2;
  }
  W=1;
  up(mutex);
  DB
  down(mutex);
  W=0;
  up(mutex);
}
```

What should be the values of blanks 1, 2, 3 to synchronization classical reader and writer algorithm?

(a) up(mutex), up(mutex), W = 1.

(b) down(mutex), up(mutex), W = 1.

(c) down(mutex), up(mutex), R >= 1 or W = 1.

(d) up(mutex), up(mutex), R >= 1 or W = 1.

READERS-WRITERS PROBLEM (PRACTICE QUESTION)

Let m[0]…m[4] be mutexes (binary semaphores) and P[0] …. P[4] be processes. Suppose each process P[i] executes the following:

```
wait (m[i]);
wait(m[(i+1) mode 4]);
------
release (m[i]);
release (m[(i+1)mod 4]);
```

Let initial value of mutex is 1

CS

This could cause: (A) Mutual exclusion is satisfied (B) mutual exclusion is not satisfied (C) deadlock

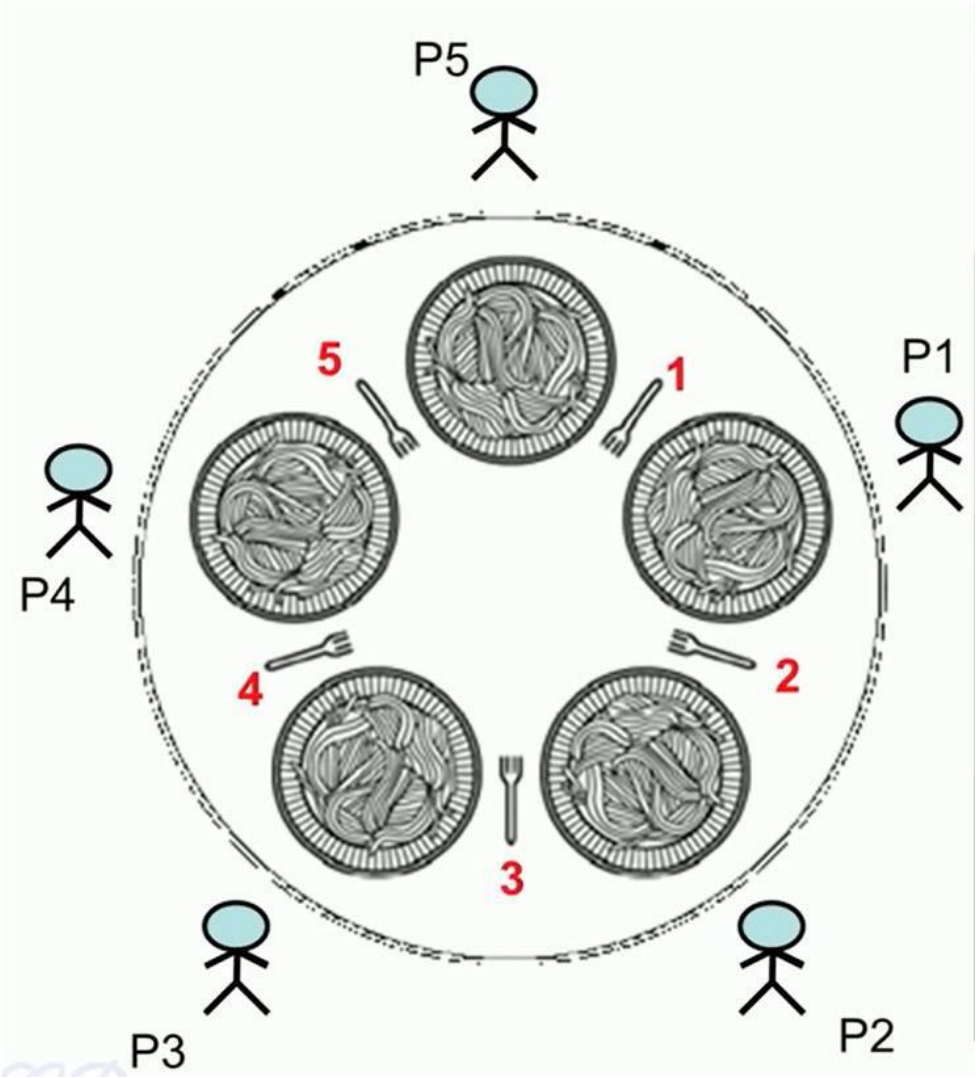
Which of the above are true

1) Only A
2) Only B
3) Only A and C
4) Only B and C

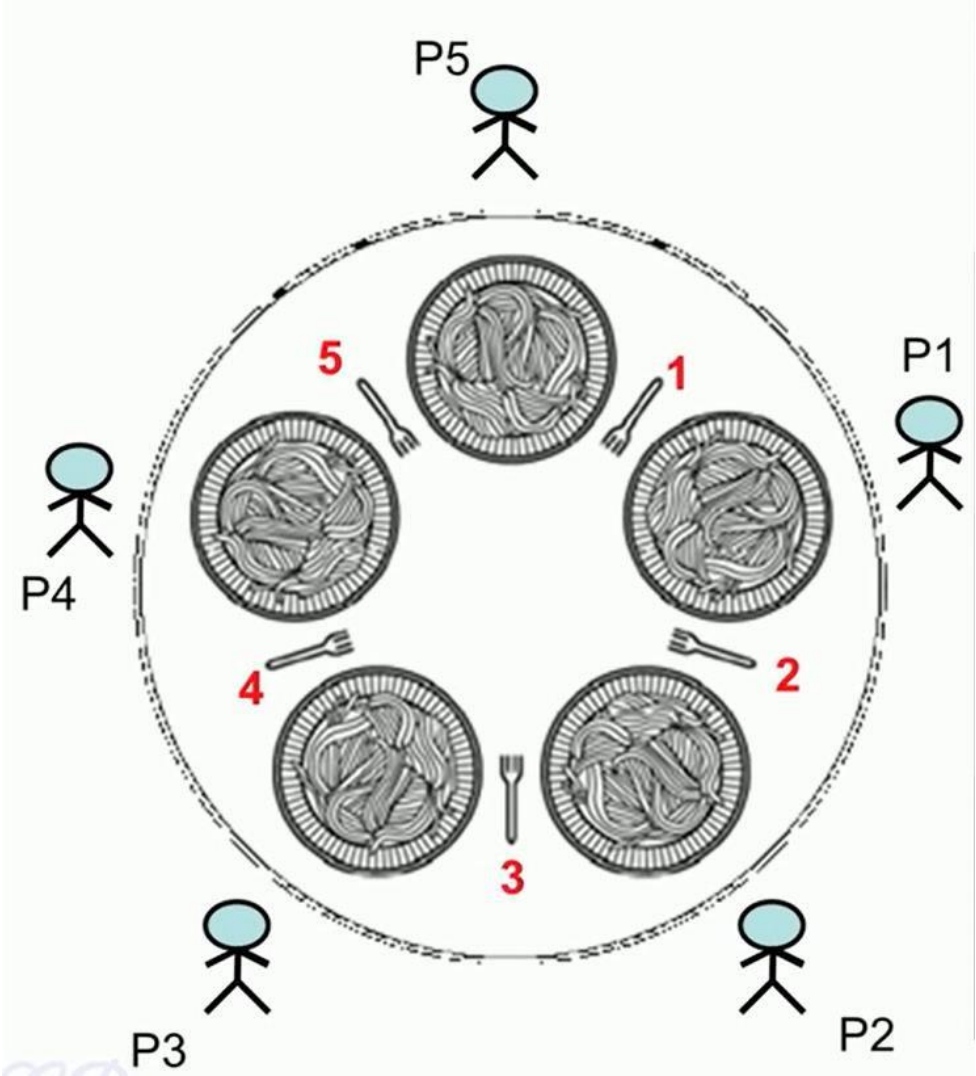
DINING PHILOSOPHERS PROBLEM

The dining philosophers problem is a classic synchronization problem involving the **allocation of limited resources** amongst a group of processes in a **deadlock-free and starvation-free manner.**

- Consider 5 philosophers (processes), sitting around a table.
- Alternating between two activities: eating and thinking.
- To eat, philosopher needs to **hold both forks** (left and right).
- When a philosopher thinks, it puts down both **forks** (or chopsticks) in their original locations.



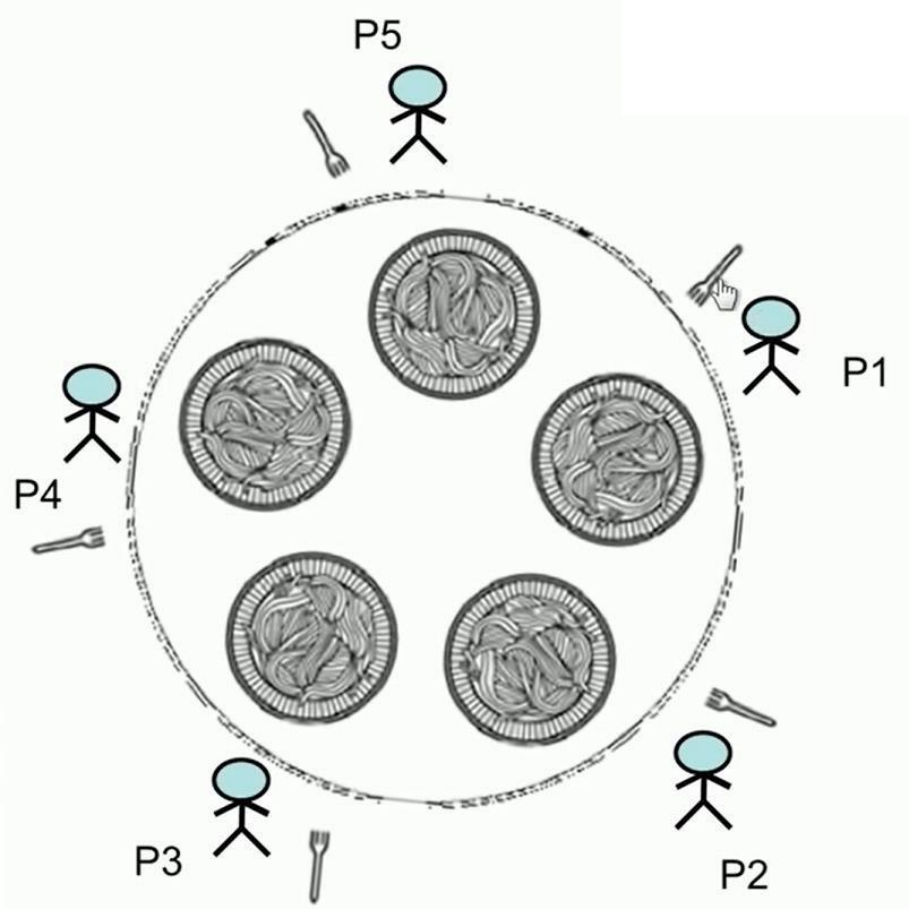
DINING PHILOSOPHERS: **FIRST SIMPLE TRY**




```
#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: **FIRST SIMPLE TRY**

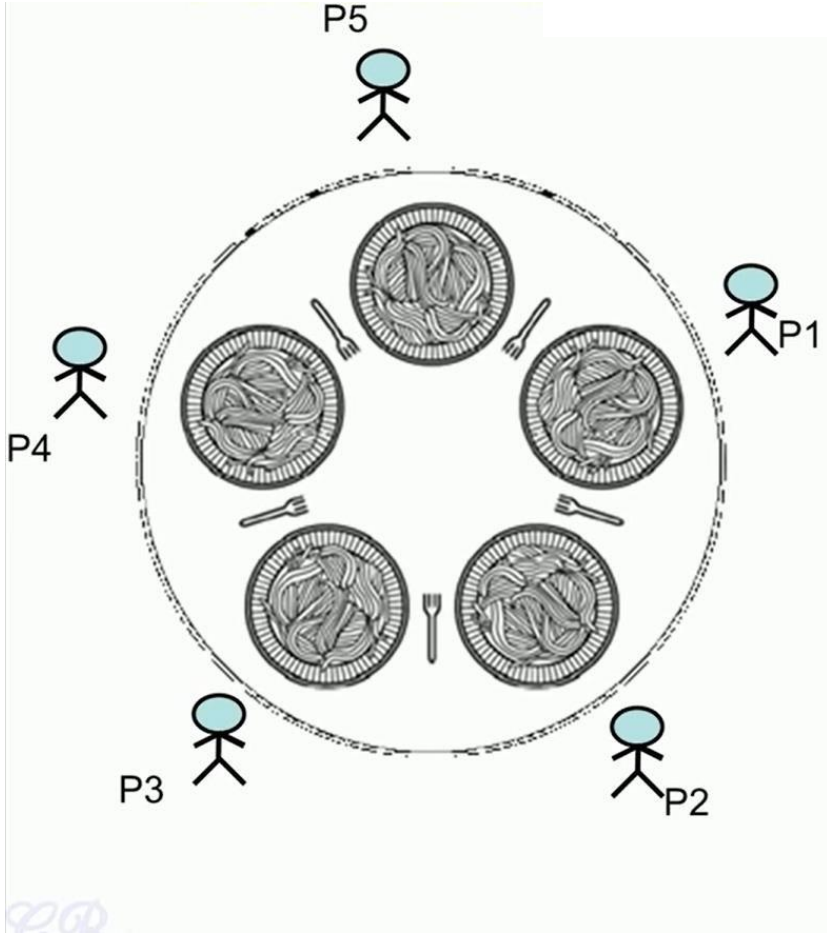



```
#define N 5

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

If all the philosophers decide to take their **right fork at the same time**, then **it will be a deadlock** i.e. a situation where processes continue to run indefinitely without making any progress.

DINING PHILOSOPHERS: SECOND TRY




```
#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);
     }
}
```

This solution **reduces the possibility** of starvation but does not guarantee it.

DINING PHILOSOPHERS: THIRD TRY

- Protects the critical sections using a **mutex**.
- Is deadlock prevention guaranteed here?
  Yes
- Issue involved:
  Only one process 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: FINAL 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: FINAL SOLUTION WITH **SEMAPHORES**

```
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: FINAL SOLUTION WITH SEMAPHORES

```
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: FINAL SOLUTION WITH **SEMAPHORES**

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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

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

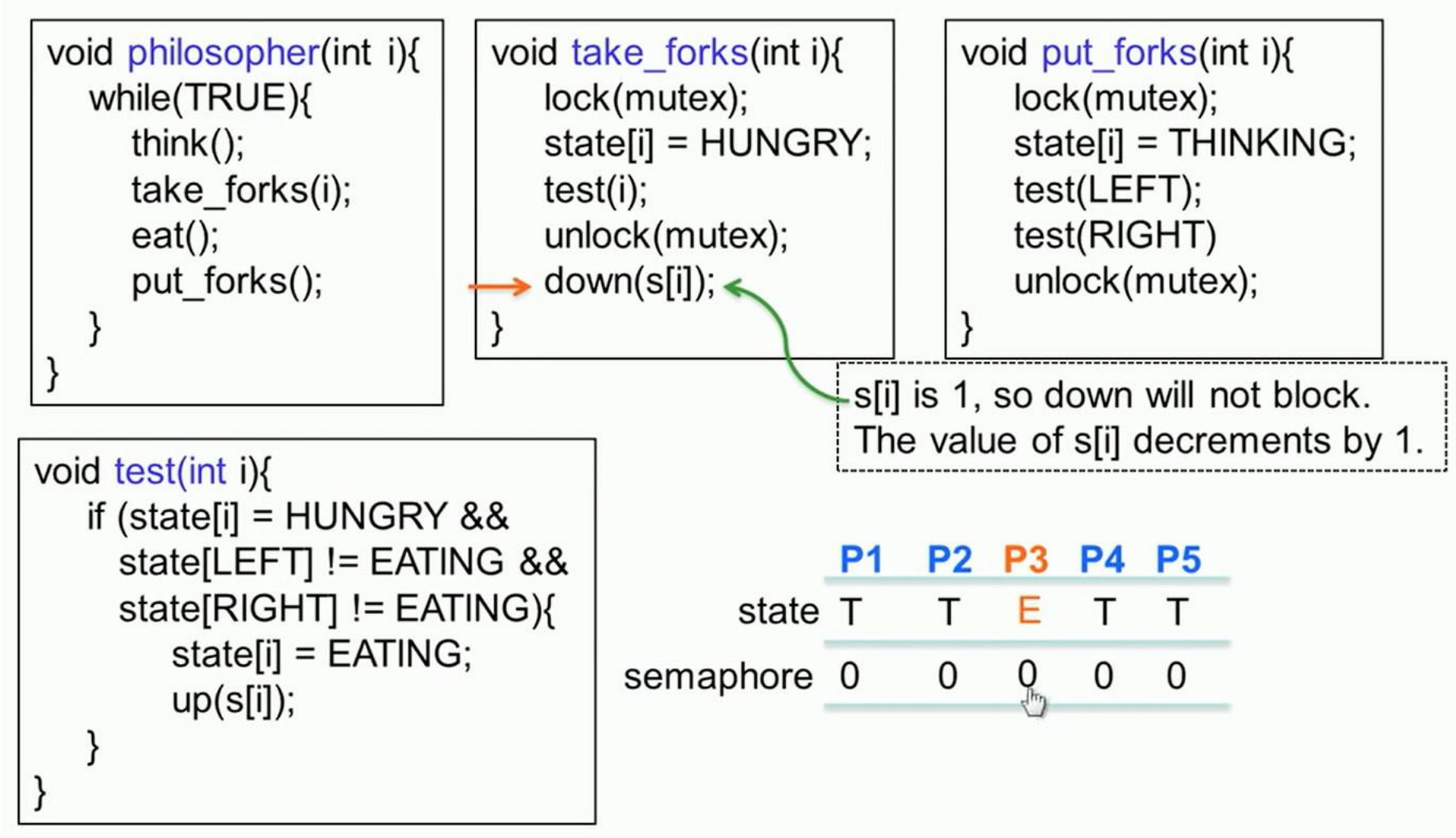
```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
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	E	T	T
semaphore	0	0	1	0	0

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**



DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

```
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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

```
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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

```
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	H	T
semaphore	0	0	0	0	0

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

```
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

DINING PHILOSOPHERS: FINAL SOLUTION WITH **SEMAPHORES**

```
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]);
    }
}
```

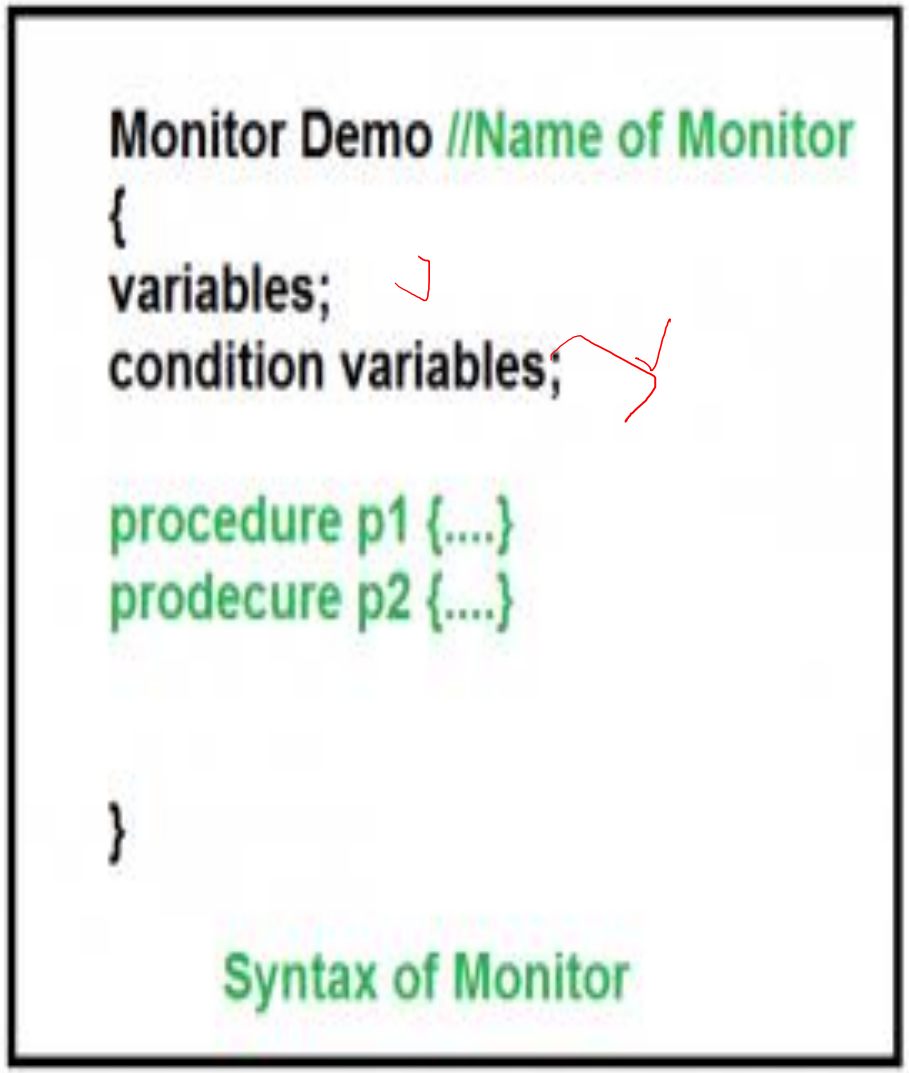
wakeup

P1	P2	P3	P4	P5
T	T	T		T
0	0	0	0	0

MONITORS

- Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization.
- Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.
- A monitor is essentially a module that encapsulates a shared resource and provides access to that resource through a set of procedures.
- The procedures provided by a monitor ensure that only one process can access the shared resource at any given time, and that processes waiting for the resource are suspended until it becomes available.

MONITORS



Condition Variables: Two different operations are performed on the condition variables of the monitor.

- Wait e.g. x.wait() or wait(x)
- Signal e.g. x.signal() or signal (x)

Thank you