In This Module...

- Dining Philosophers
- Readers and Writers
In this problem, we simulate \(N\) (4 or more) philosophers seated around a circular table.

The problem was originally devised by Edgser Dijkstra (of semaphore fame) as an exercise for students; Tony Hoare (of monitor fame) reformulated it as we see it today.

Philosophers spend their entire lives alternating between two states: eating and thinking.

Eating requires the use of a pair of forks (or chopsticks, or some other implement of your choice).

Once a philosopher picks up one fork, it is held until the other fork is picked up and eating is completed.

One fork is placed between each pair of philosophers.

Can you develop an algorithm that will allow these philosophers to think and eat harmoniously?
/* Assume forks and philosophers are numbered 0 to N-1 */
/* Fork 0 is to the left of philosopher 0, and fork */
/* 1 to the right. Each philosopher executes this */
/* function, called with the philosopher number. */
#define N 4
void philosopher (int i) {
    while (TRUE) {
        think();
        pick_up_fork(i);
        pick_up_fork((i+1) % N);
        eat();
        put_down_fork(i);
        put_down_fork((i+1) % N);
    }
}

Unfortunately, this obvious effort at a solution has a serious problem. Can you see it?
The key to finding problems in proposed solutions to IPC problems is to consider every allowable sequence of event occurrences, regardless of how unlikely they might be (think of S. Holmes!)

In the proposed solution, consider what would happen if each philosopher simultaneously picked up their left fork.

Each philosopher would then wait forever for their right fork, which is being held by their neighbor to the right. All philosophers will then eventually starve!
Holmes’ (Conan Doyle’s) Quote

The quote from Sherlock Holmes of relevance: “How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?”

This appears in *The Sign of the Four* (1890), and slightly differently in *The Adventure of the Blanced Soldier* (1927).
Another Solution Attempt

- Observe that each philosopher must have two forks before beginning to eat.
- We need some way to guarantee the availability of both forks before proceeding to pick up either one.
- In this solution, each pair of forks is controlled by a single semaphore.
- Each philosopher now has an additional state, hungry, in which they express their desire to eat.
Dining Philosopher’s Solution – Part 1

/*---------------------*/
/* Global Declarations */
/*---------------------*/
#define N 4               /* # of philosophers */
#define LEFT (i-1)%N      /* left neighbor # */
#define RIGHT (i+1)%N     /* right neighbor # */
#define THINKING 0         /* philosopher states */
#define HUNGRY 1
#define EATING 2
int state[N];           /* each philosophers state */
semaphore mutex = 1;     /* to modify state */
semaphore s[N];          /* fork pair availability */
Dining Philosopher’s Solution – Part 2

Each philosopher calls this function, with the parameter i set to the philosopher’s identification (in the range 0 to N-1).

```c
void philosopher (int i) {
    while(TRUE) {
        think();
        pick_up_forks(i);
        eat();
        put_down_forks(i);
    }
}
```
To get the forks, each philosopher atomically sets its state to HUNGRY, then tests for the availability of the forks, doing an “up” on the fork availability semaphore s[i] if both are available. After leaving the critical section, a “down” on s[i] will succeed if both forks were available. Otherwise the philosopher will wait for a neighbor to do an “up” on s[i].

```c
void pick_up_forks (int i) {
    down(&mutex);    /* state, s mods are atomic */
    state[i] = HUNGRY;   /* say I'm now hungry */
    test(i);       /* up s[i] if I'm hungry & forks free */
    up(&mutex);     /* allow others to change state, s */
    down(&s[i]);    /* continue when forks available */
}
```
A philosopher relinquishes its forks and resumes thinking. It also tests to see if its neighbors are hungry, and if a fork is put down, it may enable a neighbor to eat now.

```c
void put_down_forks (int i) {
    down(&mutex); /* atomic action */
    state[i] = THINKING; /* We are now thinking */
    test(LEFT); /* check left philosophers state */
    test(RIGHT); /* and right philosophers state */
    up(&mutex); /* no more mutex access is reqd */
}
```
If philosopher \( i \) is hungry, and neither of its neighbors is eating, then both forks are available. An “up” is done on the appropriate semaphore \((s[i])\) in this case. This code is only called from within a critical section.

```c
void test (int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```
This solution to the dining philosopher’s problem eliminates the possibility of deadlock (a topic which we’ll consider in detail in module 7).

It does not, however, address the problem variously called “Livelock,” “Indefinite Postponement,” or “Starvation.”

With livelock, some processes may continually be able to make progress, but other processes may be continually prevented from getting the resources they need.
Consider the following scenario, using our four-philosopher setup:

1. Only philosopher 1 is eating. Philosopher 3 is thinking. The others are hungry.
2. Just before philosopher 1 puts its forks down, philosopher 3 becomes hungry. Since the forks to its left and right are available, it begins eating.
3. Philosopher 1 puts down its forks and begins thinking, but the hungry philosophers are still hungry, and cannot get both of their needed forks.
4. Now steps 1, 2 and 3 repeat, with philosophers 1 and 3 changing roles. Philosopher 1 begins eating, then philosopher 3 begins thinking.

Philosophers 1 and 3 can continually eat and think; the others are starving.

One approach to solving this problem involves a technique called “aging.” We'll discuss the aging concept in the next module.
In this problem there are two classes of processes, readers and writers. Processes in each class eventually want to access a shared resource (e.g. the database).

Readers will never modify the resource, so many of them may be “inside” the database at the same time.

Writers, of course, will potentially modify the database, so only one writer can be “inside” the database at any instant.
The problem is to develop a solution for a generic reader and a generic writer that will permit them to share access to the database, permit multiple access by readers, and atomic access by writers.

There are multiple variations on this problem:
- Waiting readers have priority over writers
- Waiting writers have priority over readers
- First-come first-served (a single queue for all waiting processes)
- ...and others

We will consider only the reader priority problem here.
The solution uses two semaphores:

- **db** is a binary semaphore (one whose count will always be zero or one) used to guarantee mutually exclusive access to the database by either one or more readers, or by one writer.
- **mutex** is another binary semaphore used to guarantee mutually exclusive access to the global variable **rc**, which holds the count of the number of active reader processes (that is, those that are “inside” the database).
The Reader’s Code

down(&mutex);                /* access rc*/
rc = rc + 1;                /* one more reader */
if (rc == 1)               /* first reader? */
  down(&db);                /* yes, access resource */
up(&mutex);                /* end access to rc */
read_database();          /* reading resource... */
down(&mutex);              /* access rc again */
rc = rc - 1;               /* one less reader */
if (rc == 0)               /* last one out? */
  up(&db);                  /* yes, relinquish resource */
up(&mutex);                /* end access to rc */
use_data();               /* use the data just read */
The Writer’s Code

Writers are considerably simpler than readers as they don’t need to allow other writers to enter while they’re using the resource. All that’s required is mutually exclusive access to the resource.

```
thinkup_data();
down(&db);
write_data();
up(&db);
```
In the next module we'll look at processor scheduling. In particular, we'll examine:

- the basic goals of scheduling,
- job and process scheduling algorithms and their properties, and
- problems that may occur with certain scheduling algorithms and some solution approaches.