DINING PHILOSOPHERS

Module Number 2. Section 12
COP4600 – Operating Systems
Richard Newman

CLASSIC SYNCHRONIZATION PROBLEMS

Critical Section Problem: Mutual exclusion – only one process can be in Critical Region at a time

Producer-Consumer Problem: Producers produce items and write them into buffers; Consumers remove items from buffers and consume them. A buffer can only be accessed by one process at a time.

Bounded Buffer Problem: P-C with finite # buffers

Dining Philosophers: Philosophers arranged in a circle share a fork between each pair of neighbors. Both forks are needed for a philosopher to eat, and only one philosopher can use a fork at a time.

Readers-Writers: Any number of readers can read from a DB simultaneously, but writers must access it alone

THE DINING PHILOSOPHERS (1)

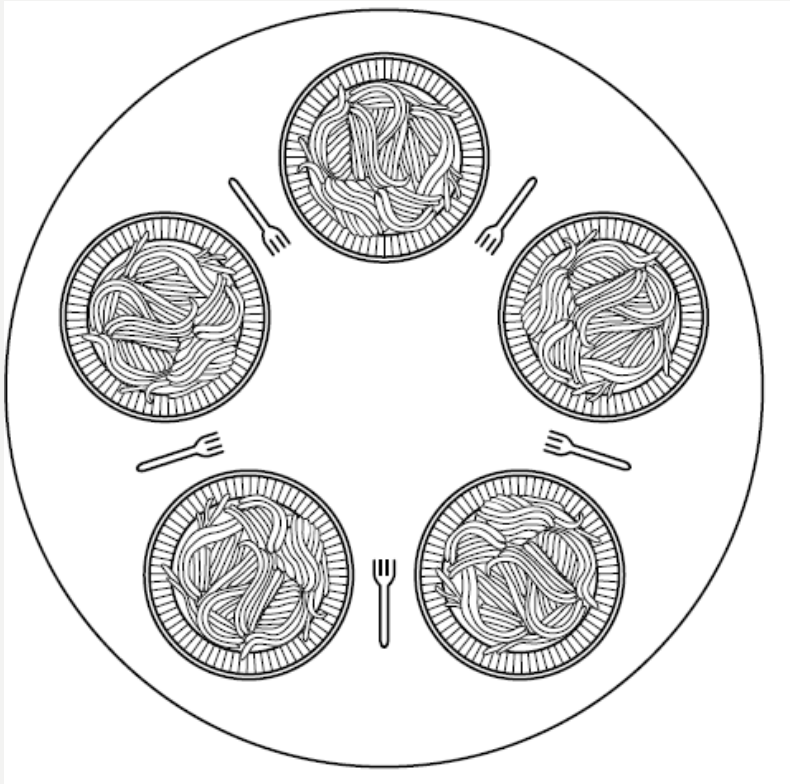


Figure 2-45. Lunch time in the Philosophy Department.

DINING PHILOSOPHERS

Only one type of process: Philosopher, which thinks for some amount of time, then gets hungry and eats, then goes back to thinking

There are five philosophers arranged in a ring with a single fork between each adjacent pair of philosophers

In order to eat, a philosopher must have exclusive access to both forks next to him/her

Concurrency: (**C1**) No assumptions can be made about how long any philosopher thinks or eats

(**C2**) Any two philosophers that are not seated next to each other can eat simultaneously

Safety: (**S1**) No two adjacent philosophers can eat at the same time (to eat, a philosopher must have exclusive access to both forks)

Liveness: (**L1**) No philosopher that is not eating can prevent any other philosopher from eating

RESOURCE ALLOCATION

Dining Philosophers type of problems arise when processes need to obtain exclusive access to multiple resources before they can proceed.

Resources may be devices, memory locations, objects, etc.

More on this in Deadlocks unit

THE DINING PHILOSOPHERS (2)

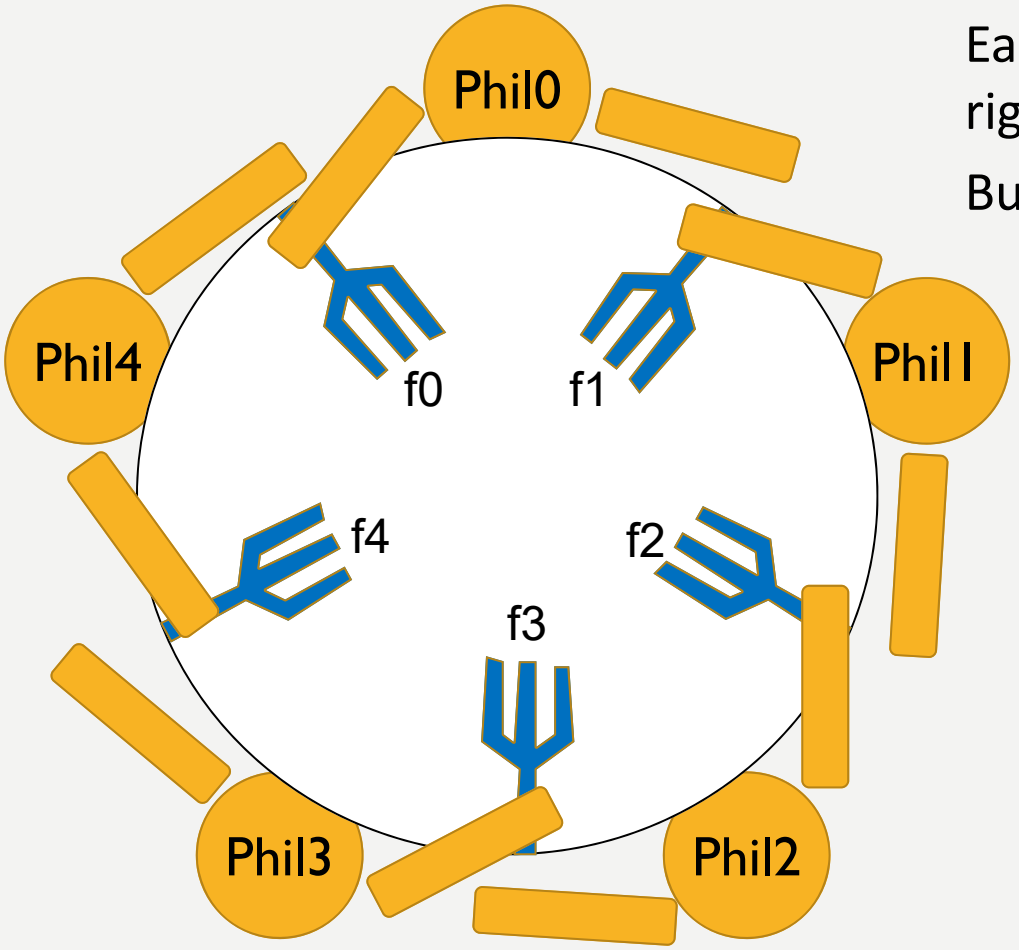
```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

What is the problem?

Figure 2-46. A nonsolution to the dining philosophers problem.

THE DINING PHILOSOPHERS



Each philosopher grabs right fork

But none can get left fork

They all starve!

THE DINING PHILOSOPHERS (3)

Figure 2-47. A solution to the dining philosophers problem.

```
#define N              5                    /* number of philosophers */
#define LEFT           (i+N–1)%N            /* number of i's left neighbor */
#define RIGHT          (i+1)%N              /* number of i's right neighbor */
#define THINKING       0                    /* philosopher is thinking */
#define HUNGRY         1                    /* philosopher is trying to get forks */
#define EATING         2                    /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
        while (TRUE) {                      /* repeat forever */
                think( );                   /* philosopher is thinking */
                take_forks(i);              /* acquire two forks or block */
                eat( );                     /* yum-yum, spaghetti */
                put_forks(i);               /* put both forks back on table */
        }
}
```

THE DINING PHILOSOPHERS (4)

```
            put_forks(i);                       /* put both forks back on table */
        }
}

void take_forks(int i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                           /* enter critical region */
        state[i] = HUNGRY;                      /* record fact that philosopher i is hungry */
        test(i);                                /* try to acquire 2 forks */
        up(&mutex);                             /* exit critical region */
        down(&s[i]);                            /* block if forks were not acquired */
}

void put_forks(i)                               /* i: philosopher number, from 0 to N–1 */
```

Figure 2-47. A solution to the dining philosophers problem.

THE DINING PHILOSOPHERS (5)

```
}

void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Figure 2-47. A solution to the dining philosophers problem.

COMMENTS ON DINING PHILOSOPHERS

- Why five philosophers? Why not four or six?
  - See discussion
- Solution given has philosopher run *non-blocking test in critical sections*, set up semaphore to release any philosopher that can eat (including self)
- Blocking call on semaphore done *outside of critical section* protecting state variables

Challenges:

- Can you devise a semaphore solution that only uses semaphores for the forks (a more natural approach)?
- Can you devise a monitor-based solution?

SUMMARY

- Defined Dining Philosophers Problem
  - Safety, Liveness, Concurrency
- Analyzed flawed non-solution
  - Race conditions, deadlock
- Developed semaphore solution
- Participate in discussion on how to solve with one semaphore for each fork