

Lecture 13:

Concurrency Part III


Professor G. Sandoval

Some slides derived from : Tanenbaum/Bo, and Brendan Dolan-Gavitt
Thanks !!

Review from Last Time

- Among others so far we have seen:
 - Spinlocks
 - Mutexes
 - Semaphores
 - Barriers

Today

- 
- Semaphores Synchronization Patterns
 - Classic & Not So Synchronization Problems:
 - Dining Philosophers
 - Readers/Writers
 - Barbershop

Synchronization patterns

- Remember that for mutual exclusion we have 2 calls.
Different authors/libraries call them differently:
- Lock, unlock
- Acquire, release
- Down, up
- Wait, Signal
- P, V

Mutex

- Enforce mutual Exclusion
- Puzzle: Add mutexes to the following example to enforce mutual exclusion to the shared variable count.

Thread A

```
1 count = count + 1
```

Thread B

```
1 count = count + 1
```



Signaling

- One thread sends a signal to another thread that something has happened.
- Signaling makes it possible to guarantee that a section of code in one thread will run before a section of code in another thread.



Rendezvous

- Generalize the Signaling pattern so that it works both ways. Thread A has to wait for Thread B and viceversa.



Rendezvous

- Generalize the Signaling pattern so that it works both ways. Thread A has to wait for Thread B and viceversa.
- Hint: Create two semaphores called aArrived and bArrived and init them both to zero.



Today

- Semaphores Synchronization Patterns
- Classic & Not So Classic Synchronization Problems:

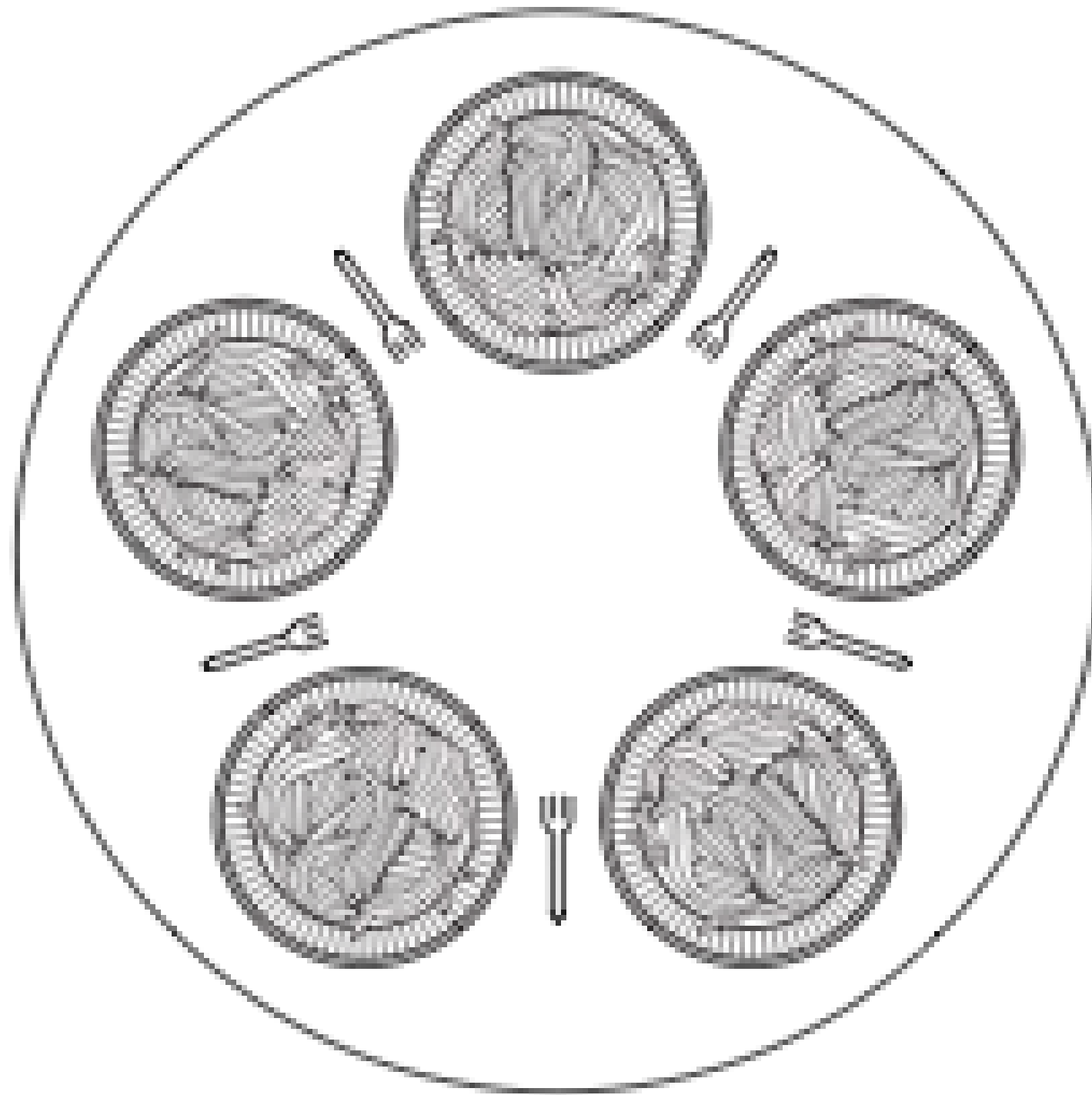


- Dining Philosophers
- Readers/ Writers
- Barbershop

Dining Philosophers Problem

- Classic synchronization problem that can be solved using many different synchronization primitives
- N philosophers around a table with N forks and N plates are either thinking or eating
- Eating requires two forks, the left fork and the right fork

Delicious Spaghetti



Dining Philosophers Problem

- A naïve solution: when a philosopher gets hungry, just pick up the left fork, then the right fork, waiting until each is available

Dining Philosophers

```
#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 */
    }
}
```

Solution to the dining philosophers problem?

Dining Philosophers Problem

- Some solutions:
 - Have a single mutex. When a philosopher wants to eat, acquire the mutex, then pick up the left and right forks, eat, and release the mutex.
 - But now only one philosopher can eat at a time, which is very inefficient
 - If you can't get your fork, wait a random amount of time before trying again
 - But sometimes we need a deterministic algorithm that will never fail, no matter how many unlikely events happen

Dining Philosophers (1)

```
#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 */
    }
}
```

~~~~~

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

# Dining Philosophers (2)

```
        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.



# Dining Philosophers (3)

```

}

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.

# Today

- Semaphores Synchronization Patterns
- Classic & Not So Classic Synchronization Problems:
  - Dining Philosophers
  - Readers Writers
  - Barbershop



# Readers/ Writers Problem

- Imagine an airline reservation system:
- Many competing processes trying to read/write
- Acceptable to have **multiple** processes **reading** at the same time.
- If one process is updating(writing) to the database, no other process should have access to it.



# Barbershop Problem

A barbershop consists of a waiting room with  $n$  chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

# Barbershop Problem

- To make this more concrete we have 2 threads: Barber & Customer
- Customer threads invoke a function named **getHaircut**
- If a customer thread arrives and the shop is full it can call a function called **balk** and it never returns
- The barber thread should invoke **cutHair**
- When the barber invokes **cutHair** there should be exactly one thread invoking **getHaircut** concurrently



# Barbershop Problem

Barbershop hint

```
1 n = 4
2 customers = 0
3 mutex = Semaphore(1)
4 customer = Semaphore(0)
5 barber = Semaphore(0)
6 customerDone = Semaphore(0)
7 barberDone = Semaphore(0)
```

`n` is the total number of customers that can be in the shop: three in the waiting room and one in the chair.

`customers` counts the number of customers in the shop; it is protected by `mutex`.

The barber waits on `customer` until a customer enters the shop, then the customer waits on `barber` until the barber signals him to take a seat.

After the haircut, the customer signals `customerDone` and waits on `barberDone`.