Preview – Classical IPC problems

- The Dining Philosophers Problem
- The Readers and Writers Problem
- The Sleeping Barber Problem
Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock
Dining Philosophers (2)

#define N 5

void philosopher(int i)
{
    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 */
    }
}

A non-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;
int state[N];                /* semaphores are a special kind of int */
semaphore mutex = 1;         /* array to keep track of everyone's state */
semaphore s[N];              /* mutual exclusion for critical regions */
/* 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 */
    }
}
Dining Philosophers (4)

```c
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 */
{
    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]);
    }
}
```

Solution to dining philosophers problem (part 2)
The Readers and Writers Problem

typedef int semaphore;  /* use your imagination */
semaphore mutex = 1;    /* controls access to 'rc' */
semaphore db = 1;       /* controls access to the database */
int rc = 0;            /* # of processes reading or wanting to */

void reader(void)  
{  
    while (TRUE) {   /* repeat forever */
        down(&mutex);  /* get exclusive access to 'rc' */
        rc = rc + 1;   /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);    /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex);   /* get exclusive access to 'rc' */
        rc = rc - 1;    /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex);     /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}

void writer(void)  
{   
    while (TRUE) {   /* repeat forever */
        think_up_data(); /* noncritical region */
        down(&db);      /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db);         /* release exclusive access */
    }
}

A solution to the readers and writers problem
The Sleeping Barber Problem (1)
The Sleeping Barber Problem (2)

```c
#define CHAIRS 5  /* # chairs for waiting customers */
typedef int semaphore;  /* use your imagination */
semaphore customers = 0;  /* # of customers waiting for service */
semaphore barbers = 0;  /* # of barbers waiting for customers */
semaphore mutex = 1;  /* for mutual exclusion */
int waiting = 0;  /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);  /* go to sleep if # of customers is 0 */
        down(&mutex);  /* acquire access to 'waiting' */
        waiting = waiting - 1;  /* decrement count of waiting customers */
        up(&barbers);  /* one barber is now ready to cut hair */
        up(&mutex);  /* release 'waiting' */
        cut_hair();  /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);  /* enter critical region */
    if (waiting < CHAIRS) {
        waiting = waiting + 1;  /* increment count of waiting customers */
        up(&customers);  /* wake up barber if necessary */
        up(&mutex);  /* release access to 'waiting' */
        down(&barbers);  /* go to sleep if # of free barbers is 0 */
        get_haircut();  /* be seated and be serviced */
    } else {
        up(&mutex);  /* shop is full; do not wait */
    }
}
```

Solution to sleeping barber problem.