CLASSICAL IPC PROBLEMS

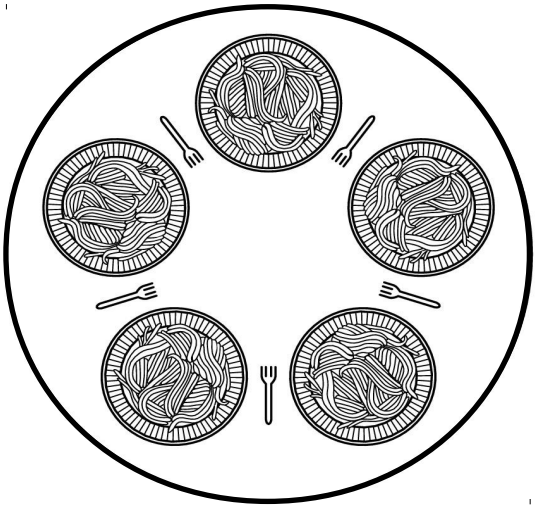
Classical IPC Problems

1. Dining Philosophers Problem
2. The Readers and Writers Problem
3. The Sleeping Barber Problem

1. Dining philosophers problems:

There are N philosphers sitting around a circular table eating spaghetti and discussing philosphy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

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



The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible.

One idea is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up
- think until the right fork is available; when it is, pick it up
- eat
- put the left fork down
- put the right fork down
- repeat from the start

This solution is incorrect: it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a

deadlock.
We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation

The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

Solution:

```
#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 N1 */
{
    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 */
    }
}
void take_forks(int i)          /* i: philosopher number, from 0 to N1 */
{
    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 N1 */
{
    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 N1* /
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Readers Writer problems:

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database (Courtois et al., 1971). Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader. The question is how do you program the readers and the writers? One solution is shown below.

Solution to Readers Writer problems

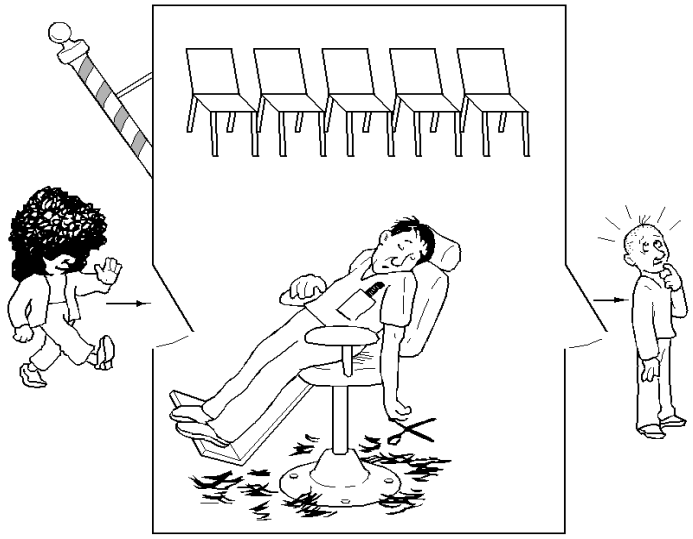
```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */
void reader(void)
{
    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 */
    }
}
```

In this solution, the first reader to get access to the data base does a down on the semaphore db. Subsequent readers merely have to increment a counter, rc. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

Sleeping Barber Problem

customers arrive to a barber, if there are no customers the barber sleeps in his chair. If the barber is asleep then the customers must wake him up.



The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.

Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, then the customer leaves. Based on a naive analysis, the above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a

number of problems that can occur that are illustrative of general scheduling problems.

The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time. For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While he is on his way, the barber finishes the haircut he is doing and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps. The barber is now waiting for a customer and the customer is waiting for the barber.

In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Solution:

Many possible solutions are available. The key element of each is a mutex, which ensures that only one of the participants can change state at once. The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair. A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair. This eliminates both of the problems mentioned in the previous section. A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room.

A multiple sleeping barbers problem has the additional complexity of coordinating several barbers among the waiting customers