OPERATING SYSTEM

Process Management

Chapter 2

Processes and Threads

2.4 Classical IPC problems
(from earlier versions)

Popular IPC Problems

Producer-Consumer Problem – managing shared buffers without overflow or underflow.

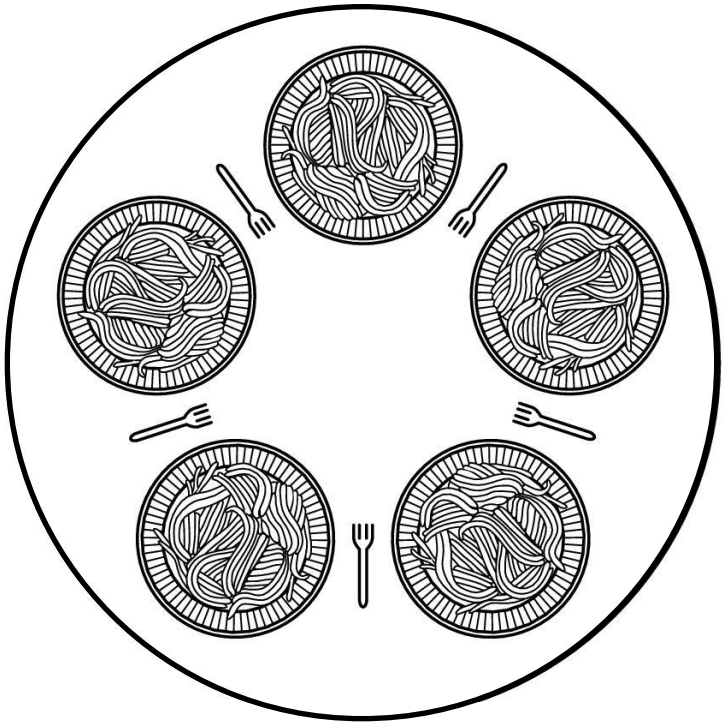
Readers-Writers Problem – balancing concurrent reads and exclusive writes.

Dining Philosophers Problem – preventing deadlock and starvation in shared resource usage.

Sleeping Barber Problem – handling synchronization and fairness in service systems.

Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to program the scenario properly?



Dining Philosophers: A Solution

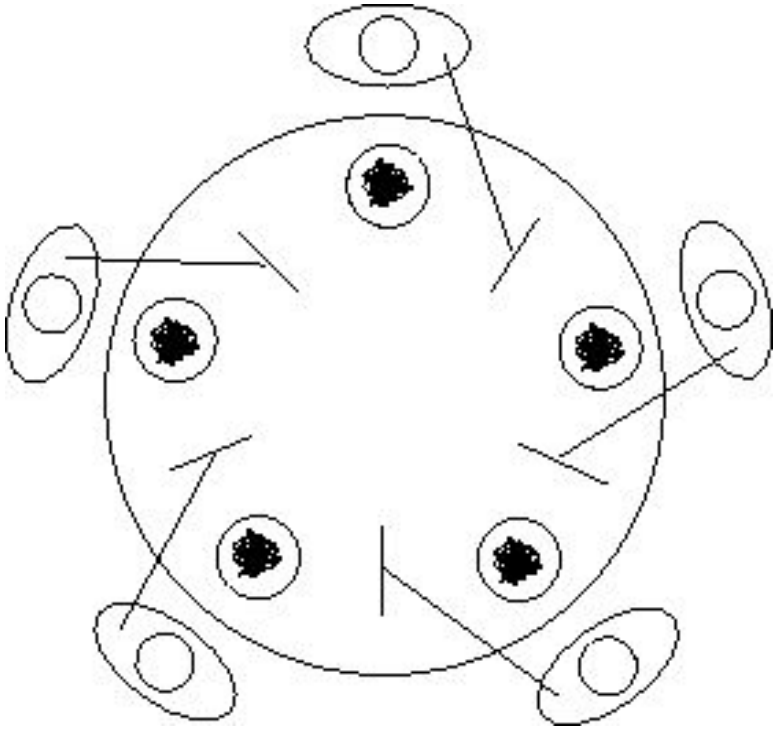
```
#define N 5                                     /* number of philosophers */

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 */
    }
}
```

A nonsolution to the dining philosophers problem

Dining Philosophers: Problems with Previous Solution

- Deadlock may happen
  - Everyone takes the left fork simultaneously



Dining Philosophers: Problems with Previous Solution

Tentative Solution:

- After taking left fork, check whether right fork is available.
- If not, then return left one, wait for some time and repeat again.

Problem:

- All of them start and do the algorithm synchronously and simultaneously: STARVATION (A situation in which all the programs run forever but fail to make any progress)
- Solution: Random wait; but what if the most unlikely of same random number happens?)

Dining Philosophers:Another Attempt, Successful!

```
void philosopher(int i)
{
  while (true)
  {
      think();
      down(&mutex);
      take_fork(i);
      take_fork((i+1)%N);
      eat();
      put_fork(i);
      put_fork((i+1)%N);
      up(&mutex);
  }
}
```

- Theoretically solution is OK- no deadlock, no starvation.
- Practically with a performance bug:
  - Only one philosopher can be eating at any instant: absence of parallelism

Dining Philosophers: Solution (Part 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 */
    }
}
```

Solution to dining philosophers problem (part 1)

Dining Philosophers: Solution Part 2

```
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(1)

- Dining Philosopher Problem: Models processes that are competing for exclusive access to a limited resource
- Readers Writers Problem: Models access to a database

Example: An airline reservation system- many competing process wishing to read and write-

- Multiple readers simultaneously- acceptable
- Multiple writers simultaneously- not acceptable
- Reading, while write is writing- not acceptable

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write asap

```
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 Readers and Writers Problem(3)

Issue regarding the first solution

- Inherent priority to the readers
- Say a new reader arrives every 2 seconds and each reader takes 5 seconds to do its work. What will happen to a writer?

Issue regarding second variation

- Writer don't have to wait for readers that came along after it
- Less concurrency, lower performance

The Sleeping Barber Problem (1)

The Sleeping Barber Problem

- One barber, n chairs for waiting customers
- No customer => barber sleeps
- Customer has to wake a sleeping barber
- Additional customers will either wait or leave
- Models a queuing situations

```
#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) {         /* if there are no free chairs, leave */
        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 */
    }
}
```

Further Reading

https://www.geeksforgeeks.org/operating-systems/classical-ipc-problems/