**Classical**
- Producer-consumer problem
  - Producer-consumer problem with infinite buffer
    - Description
    - Solution
  - Producer-consumer problem with finite buffer
    - Description
    - Solution
- Readers-writers problem
  - Original problem
    - Description
    - Solution
  - Solve writer starvation
    - Description
    - Solution
  - Writer-priority
    - Description
    - Solution
    - Explanation
- No-starve mutex
  - Description
  - Solution
- Dining philosophers
  - Description
  - Solution

**Less Classical**
- The dining savages problem
  - Description
  - Solution
- The barbershop problem
  - Description
  - Solution

信号量经典例题归档

2021.6.6 wzh

# Classical

## Producer-consumer problem

### Producer-consumer problem with infinite buffer

#### Description

1. assumption:

- producers create items and add them to the buffer
- consumer remove items from buffer and process them

2. constraint

- threads must have exclusive access to the buffer

- If a comsumer thread arrives while the buffer is empty, it block until producer adds a new item
- The item-get and item-add operation can run concurrently.

### Solution

```
Semaphore mutex = 1; //provide exclusive access to the buffer
Semaphore items = 0; //indicate the number of items in the buffer

//producer
void producer() {
    item = waitForEvent();
    P(mutex);
        buffer.add(item);
    V(mutex);
    V(items);
}

//comsumer
void consumer() {
    P(items);
    P(mutex);
        item = buffer.get();
    V(mutex);
    item.process();
}
```

## Producer-consumer problem with finite buffer

### Description

1. assumption:
    - finite buffer
2. constraint:
    - same as infinte buffer problem
    - If a producer arrives when the buffer is full, it blocks until a consumer remove an item from buffer

### Solution

- **CANNOT write like**:

```
if(items >= bufferSize)
    block();
```

**CANNOT check semaphore value, the only operation is P,V**

```
Semaphore mutex = 1
Semaphore items = 0
Semaphore space = bufferSize;

//producer
void producer() {
    P(space);
    item = waitForEvent();
    P(mutex);
        buffer.add(item);
    V(mutex);
    V(items);
}

//consumer
void consumer() {
    P(items);
    P(mutex);
        item = buffer.get();
    V(mutex);
    V(space);
    item.process();
}
```

# Readers-writers problem

## Original problem

### Description

1. Assumption
2. Constraint:
    - Any number of readers can be in the critical section simultaneously
    - Writers must have exclusive access to the critical section

### Solution

```
int readers = 0;
Semaphore mutex = 1; //provide readers exclusive access
Semaphore roomEmpty = 1; //provide reader and writer exlusive access to the room

//reader
void reader() {
    P(mutex);
        readers += 1;
        if(readers == 1)
            P(roomEmpty);
    V(mutex);

    reader();

    P(mutex);
        readers-=1;
        if(readers == 0)
            V(roomEmpty);
    V(mutex);
}

//writer
void writer() {
    P(roomEmpty);
        write();
    V(roomEmpty);
}
```

**Writer Starvation!!!**

## Solve writer starvation

### Description

1. Constraint
    - when a writer arrives, the existing readers can finish, but no additional readers may enter

### Solution

```
Semaphore roomEmpty = 1;
Semaphore turnstile = 1;
Semaphore mutex = 1;
int readers = 0;

//writer
void writer() {
    P(turnstile);
    P(roomEmpty);
        write();
    V(turnstile);
    V(roomEmpty);
}

//reader
void reader() {
    P(turnstile);
    V(turnstile);

    P(mutex);
        readers += 1;
        if(readers == 1)
            P(roomEmpty);
    V(mutex);

    read();

    P(mutex);
        readers-=1;
        if(readers == 0)
            V(roomEmpty);
    V(mutex);
}
```

## Writer-priority

## Description

1. Constraint
   - Once a writer arrives, no reader should be allowed to enter until all writers have left the system.

## Solution

```
Semaphore noReaders = 1;
Semaphore noWriters = 1;
Semaphore readersMutex = 1;
Semaphore writersMutex = 1;
int readers = 0;
int writers = 0;

//reader
void reader() {
    P(noReaders);
        P(readersMutex);
            readers += 1;
            if(readers == 1)
                P(noWriters);
        V(readersMutex);
    V(noReaders);

    read();

    P(readersMutex);
        readers -= 1;
        if(readers == 0)
            V(noWriters);
    V(readersMutex);
}

//writer
void writer() {
    P(writersMutex);
        writers += 1;
        if(writers == 1)
            P(noReaders);
    V(writersMutex);

    P(noWriters);
        write();
    V(noWriters);

    P(writersMutex);
        writers -= 1;
        if(writers == 0)
            V(noReaders);
    V(writersMutex);
}
```

### Explanation

- If a reader is in critical section, it holds noWriters, but it doesn't hold noReaders. Thus if a writer arrives it can lock noReaders, caues subsequent readers to queue.
- If a writer is in the critical section it holds both noReaders and noWriters. This has the side effect of insuring there are no readers and no other writers in the critical section. Also, the code allows multiple writers to queue on noWriters. Only when the last writer exits can the readers enter.

## No-starve mutex

- This is a more basic level of reader-writer problem
- It means the posibility that one thread might wait indefinitely while others procees.

### Description

- Puzzle: write a solution to the mutual exlusion problem using weak semaphores. The Solution should provide the following guarantee: once a thread arrives and attempts to enter the mutex, there is a bound on the number of threads that can proceed ahead of it. You can assume that the total number of threads is finite.

### Solution

```
int room1 = 0;
int room2 = 0;
Semaphore mutex = 1;
Semaphore t1 = 1;
Semaphore t2 = 0;

P(mutex);
    room1 += 1;
V(mutex);

P(t1);
    room2 += 1;
    P(mutex);
    room1 -= 1;

    if(room1 == 0) {
        V(mutex);
        V(t2);
    } else {
        V(mutex);
        V(t1);
    }

P(t2)
    room2 -= 1;
    //critical section
    if(room2 == 0)
        V(t1);
    else
        V(t2);
```

## Dining philosophers

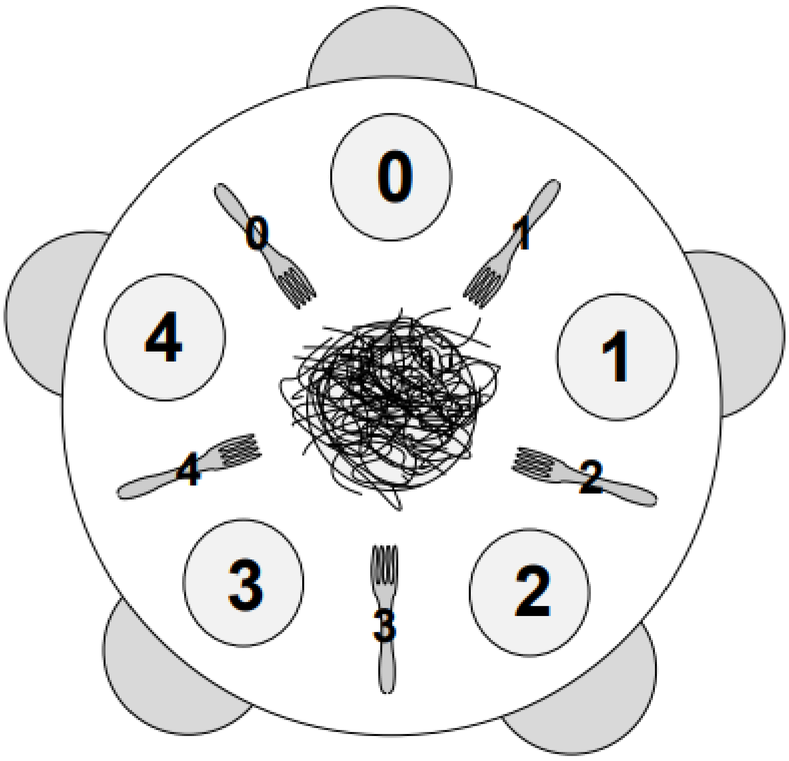
## Description

1. Assumption

- each philosopher execute the following loop:

```
while(true) {
    think();
    get_forks();
    eat();
    put_forks();
}
```

- the outline of the situation is as following



2. Constraint

- Only one philosopher can hlod a fork at a time
- It must be impossible for a deadlock to occur
- It must be impossible for a philosopher to starve waiting for a fork
- It must be impossible for more than one philosopher to eat at the same time

## Solution

- which fork?

```
int left(int i) {
    return i;
}

int right(int i) {
    return (i + 1) % 5;
}
```

- solution 1
  If there are only four philosophers at the table, no deadlock will occur

```
Semaphore fork[5] = { 1 };
Semaphore footman = 4;

void get_forks(int i) {
    P(footman);
    P(fork[right(i)]);
    P(fork[left(i)]);
}

void put_forks(int i) {
    V(fork[right(i)]);
    V(fork[left(i)]);
    V(footman);
}
```

- solution 2
  If at least one leftie and at least one rightie, then deadlock is impossible.

# Less Classical

## The dining savages problem

### Description

1. Assumption

- Any number of savage threads run the following code

```
while(true) {
    getServingFromPot();
    eat();
}
```

- One cook thread runs following code

```
while(true) {
    putServingsInPot(M);
}
```

2. Constraint

- Savages cannot invoke `getServingFromPot()` if the pot is empty
- the cook can invoke `putServingsInPot()` only if the pot is empty

### Solution

```
int servings = 0;
Semaphore mutex = 1;
Semaphore emptyPot = 0;
Semaphore fullPot = 0;

```

```
void cook() {
    while(true) {
        P(emptyPot);
            putServingsInPot(M);
        V(fullPot);
    }
}

void savage() {
    while(true) {
        P(mutex)
            if(servings == 0) {
                V(emptyPot);
                P(fullPot);
                serving = M;
            }
            serving -= 1;
            getSevingFromPot();
        V(mutex);

        eat();
    }
}
```

## The barbershop problem

### Description

1. Assumption

- 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 the chairs are available, then the customer sits in one of the free chairs. if the barber is asleep, the customer wakes up the barber.

2. Constraint

- Customer threads should invoke a function named `getHairCut`
- If a customer thread arrives when the shop is full, it can invoke `balk`, which does not return
- The barber thread should invoke `cutHair`
- When the barber incokes `cutHair` there should be exactly one thread invoke `getHairCut` concurrently

### Solution

```
const int n = 4; //the total number of customers that can be in the shop(3 in waiting room, 1 in barber room)
int customers = 0; //count the number of customers in the shop
Semaphore mutex = 1; //protect customers
Semaphore customer = 0;
Semaphore barber = 0;
Semaphore customerDone = 0;
Semaphore barberDone = 0;

void customer() {
    P(mutex);
        if(customers == n) {
            V(mutex);
            balk();
        }
        customers += 1;
    V(mutex);

    V(customer);
    P(barber);
    //getHairCut()
    V(customerDone);
    P(barberDone);

    P(mutex);
        customer -= 1;
    V(mutex);
}

void barber() {
    P(customer);
    V(barber);
    //cutHair()
    P(customerDone);
    V(barberDone);
}
```