Operating Systems

Jinghui Zhong （钟竞辉）
Office：B3-515
Email ：jinghuizhong@scut.edu.cn

Classical IPC Problems

① Readers and Writers Problem

② Sleeping Barber Problem

③Dining Philosophers Problem

Readers and Writers Problem

Reader

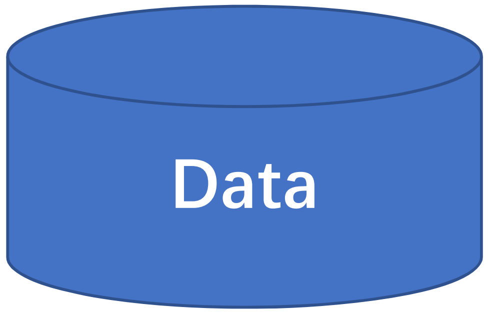
Multiple readers can read simultaneously

When reading, writing is not allowed.

Writer

Only one writer is allowed to write at a time

When writing, reading is not allowed.



Solution 1

```
Reader

Down(mutex)
Read data
Up(mutex)
```

```
Writer

Down(mutex)
Write data
Up(mutex)
```

Problem?

Only one reader can read the data at a time.

Solution 2

Initial: reader = 0; wmutex = 1;

```
Reader
reader ++;
If(reader == 1) Down(wmutex);
Read data
reader--;
If(reader == 0) Up(wmutex);
```

```
Writer

Down(wmutex)
Write data
Up(wmutex)
```

Problem ?

Will cause race condition.

Solution 3

Reader

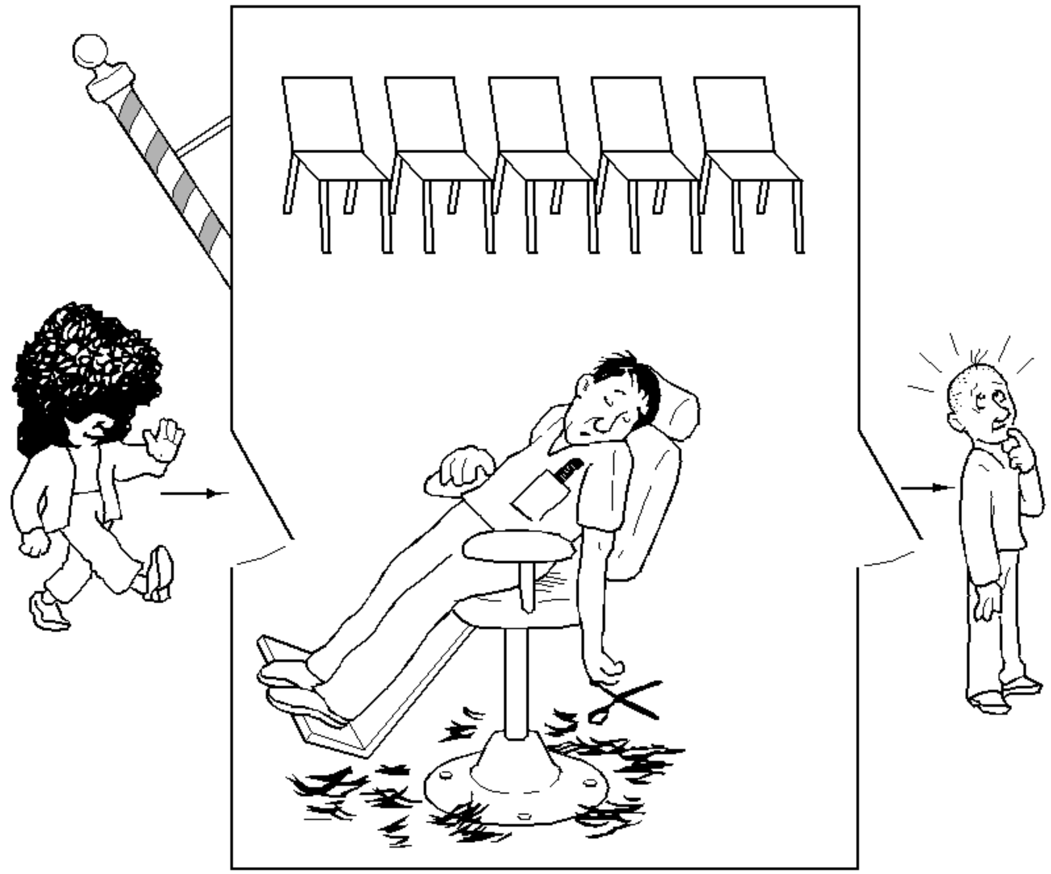
```
Down(mutex);
reader ++;
If(reader == 1) Down(wmutex);
Up(mutex);
Read data
Down(mutex);
reader--;
If(reader == 0) Up(wmutex);
Up(mutex);
```

Writer

```
Down(wmutex)
Write data
Up(wmutex)
```

The Sleeping Barber Problem

- Problem: one barber, one barber chair, and *n* chairs.
  - If there is no customer, the barber will fall asleep.
  - The first customer arrived will wake up the barber.
  - If additional customers arrive , sit down or leave the shop.



Solution

barber_ready

cust_ready

mutex

seat_num

Semaphores

Initial:

barber_ready = 0;
cust_ready = 0;
mutex = 1;

Barber

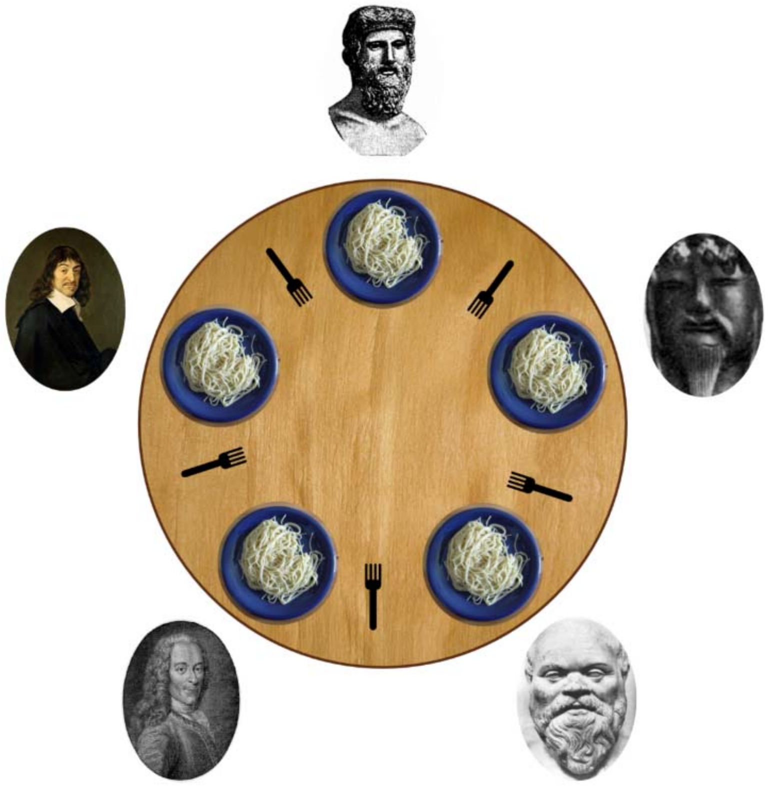
```
While (True) do {
   Down(cust_ready);
   Down(mutex);
   seat_num++;
   Up(barber_ready);
   Up(mutex);
   # cut hair here
}
```

Customer

```
Down(mutex);
If(seat_num > 0){
  seat_num--;
   Up(cust_ready);
   Up(mutex);
    Down(barber_ready)
  }else   Up(mutex);
}
```

Dining Philosophers Problem

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher cannot start eating before getting both forks.



Solution 1

```
void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_fork(i);
                take_fork((i+1) % N);
                eat( );
                put_fork(i);
                put_fork((i+1) % N);
        }
}
```

The procedure **take-fork** waits until the specified fork is available and then seizes it.

Problem?

If five philosophers become hungry at the same time, they will pick their left forks first, this will make them keep waiting for their right forks.

Solution 2

```
void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_fork(i);
                take_fork((i+1) % N);
                eat( );
                put_fork(i);
                put_fork((i+1) % N);
        }
}
```

Critical Region

Down(mutex)

Up(mutex)

Problem?

If a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled.

Solution 3（Tanenbaum’s Solution）

Hungry Thinking Eating	State of philosopher
S[5]	Semaphore, all set to 0
mutex	Semaphore, set to 1

```
Void philosopher(int i)
{
   While (True) do {
         Thinking;
         take_forks(i);
         Eating;
         put_forks(i);
     }
}
```

```
Void take_forks(int i){
         down(mutex);
          state[i] = Hungry;
          test(i);
          up(mutex);
          down(s[i]);
}
```

```
Void put_forks(int i){
         down(mutex);
         state[i] = Thinking;
          test(LEFT);
          test(RIGHT);
          up(mutex);
}
```

```
Void test(int i)
{
        if(state[i] == Hungry &&
         state[LEFT] != Eating &&
         state[Right] != Eating ){
              state[i] = Eating;
              up(s[i]);
        }
}
```

If i want to eat and the condition is satisfied, then give i a ticket

Objective: To check whether (i) can start eating. If (i) can start eating, then update (i)'s state and allow (i) to eat. Otherwise, (i) will be blocked when executing the next down operation.

Monitors

- A **monitor** is a collection of procedures, variables, and data structures that can only be accessed by one process at a time (for the purpose of mutual exclusion).
- To allow a process to wait within the monitor, a **condition variable** must be declared, as **condition x, y;**
- Condition variable can only be used with the operations **wait** and **signal** (for the purpose of synchronization).
- The operation

  x.wait();

  means that the process invoking this operation is suspended until another process invokes

  x.signal();
- The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Monitors

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

- Producer-consumer problem with monitors

  only one monitor procedure active at one time

Monitors

- Monitors in Java
  - supports user-level threads and methods (procedures) to be grouped together into classes.
  - By adding the keyword **synchronized** to a method, Java guarantees that once any thread has started executing that method, no other thread can execute that method.
- Advantages: **Ease of programming**.
- Disadvantages:
  - Monitors are a programming language concept, so they are difficult to add to an existing language; e.g., how can a compiler know which procedures were in monitor?
  - Monitors are **too expensive to implement.**

Comparing monitor with Semaphores

The **wait** and **signal** operations on condition variables in a monitor are similar to **P** and **V** operations on counting semaphores. A **wait** statement can block a process's execution, while a **signal** statement can cause another process to be unblocked. However, there are some differences between them.

What are the differences between monitor and Semaphores?

Comparing monitor with Semaphores

● **Differences:**

① When a process executes a **P** operation, it does not necessarily block that process because the counting semaphore may be greater than zero. In contrast, when a **wait** statement is executed, it always blocks the process.

② When a task executes a **V** operation on a semaphore, it either unblocks a task waiting on that semaphore or increments the semaphore counter if there is no task to unlock. On the other hand, if a process executes a **signal** statement when there is no other process to unblock, there is no effect on the condition variable.

③ Another difference between semaphores and monitors is that users awaken by a **V** operation can resume execution without delay. Contrarily, users awaken by a **signal** operation are restarted only when the monitor is unlocked.

http://courses.cs.vt.edu/~cs5204/fall00/monitor.html

Message Passing

- **Possible Approaches:**
  - Assign each process a unique address such as addr. Then, send messages directly to the process, e.g., signals in UNIX.

    send(addr, msg);

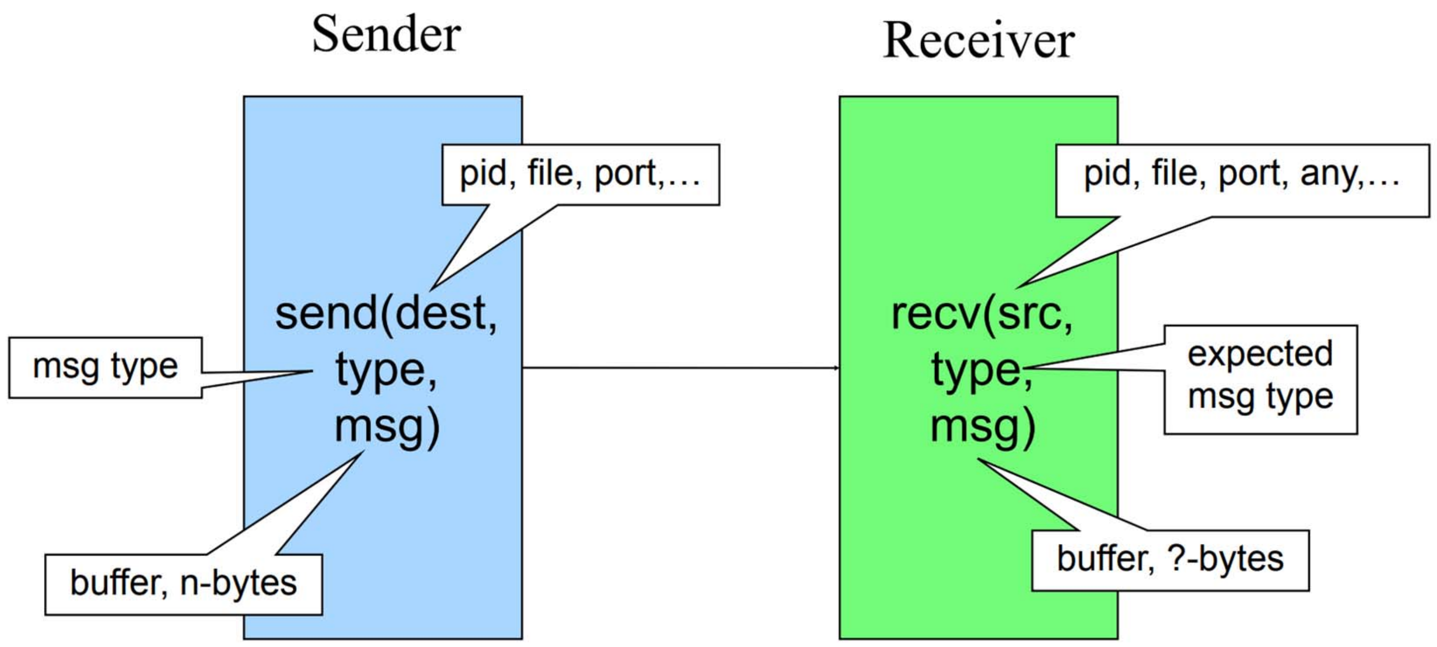
    recv(addr, msg);

- **Message passing is commonly used in parallel programming systems.**

  e.g., MPI (Message-Passing Interface).

Message Passing API

- Message passing is applicable for:
  - ✓processes inside the same computer.
  - ✓processes in a networked/distributed system.



Many ways to design the message passing API

Synchronization in message passing (1)

- Message passing may be blocking or non-blocking.
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization in message passing (2)

- Sender: it is more natural not to be blocked after issuing send:
  - can send several messages to multiple destinations.
  - but sender usually expect acknowledgment of message receipt (in case receiver fails).
- Receiver: it is more natural to be blocked after issuing receive:
  - the receiver usually needs the information before proceeding.
  - but could be blocked indefinitely if sender process fails to send.

Synchronization in message passing (3)

- **Other methods are offered, e.g., blocking send, blocking receive:**
  - ✓both are blocked until the message is received.
  - ✓provides tight synchronization (*rendezvous*).

- So, there are three combinations that make sense:

(1) Blocking send, Blocking receive;

(2) Nonblocking send, Nonblocking receive;

(3) Nonblocking send, Blocking receive – most popular .

Check points

1. What is Race Condition?
2. What is Critical Region?
3. What is IO-bound process?
4. What is Turnaround time?
5. What is the drawback of the SJF algorithm?
6. What is the advantage of RR scheduling?
7. What is condition variable in monitor?
8. What are the two operations in monitor?