VNU–HCMUS
FACULTY OF INFORMATION TECHNOLOGY

Chapter 4B

Synchronization

Instructor: **Vu Thi My Hang, Ph.D.**
TA/Lab Instructor: **Le Quoc Hoa, M.Sc.**

Plan

- Critical Section Problem
- Synchronization
- Classic synchronization problems

Plan

- **Critical Section Problem**
- Synchronization
- Classic synchronization problems

Race conditions (Tình huống tương tranh)

What if several processes/threads access a shared variable concurrently?

☹ Data inconsistencies and errors

```
Int availTicket = 1;
```

```
Client A ( ) {
1 if (availTicket > 0) {
    4 availTicket -= 1;
  }
}
```

```
Client B ( ) {
2 if (availTicket > 0) {
    3 availTicket -= 1;
  }
}
```

~~availTicket = -1~~

Critical Section Problem (Vấn đề tương tranh/tranh chấp)

```
Int availTicket = 1;
```

```
Client A ( ) {
    if (availTicket > 0) {
        availTicket -= 1;
    }
}
```

```
Client B ( ) {
    if (availTicket > 0) {
        availTicket -= 1;
    }
}
```

Critical Section (CS) (Miền găng):
Code segment in which multiple processes/threads access shared resources concurrently

(e.g., availTicket)

Solution: synchronization (Đồng bộ hoá)

Plan

- Critical Section Problem
- **Synchronization**
  - *Busy Waiting Solutions*
  - *Sleep and Wakeup Solutions*
- Classic synchronization problems

Synchronization Requirements

- Mutual Exclusion (Độc quyền truy xuất)
  - ✓ Only one process/thread can be executing inside a critical section at a time
- Progress (Có sự tiến triển)
  - ✓ A process/thread outside a critical section cannot block others to enter this critical section
- Bounded Waiting (Có giới hạn thời gian cho sự chờ đợi)
  - ✓ Processes/Threads should not wait indefinitely for entering a critical section

Synchronization Solutions

Busy Waiting

while (*no permission to enter CS*);
critical_section;

- **Software solutions**
  - ✓ Lock variable
  - ✓ Strict Alternation
  - ✓ Peterson's solution
- **Hardware solutions**
  - ✓ Interrupt disabling
  - ✓ TSL

Sleep and Wakeup

if (*no permission to enter CS*) **sleep**();
critical_section;
wake_up(another);

- ***Semaphore***
  - ✓ *Binary semaphore (mutex)*
  - ✓ *Counting semaphore*
- **Monitor**

Lock variable

```
int lock = 0;
```

```
Process_A() {
    while (1) {
        while (lock);
        lock = 1;
        critical_section;
        lock = 0;
        …
    }
}
```

```
Process_B() {
    while (1) {
        while (lock);
        lock = 1;
        critical_section;
        lock = 0;
        …
    }
}
```

☹ Two processes/threads may be inside CS at a time

Strict Alternation

```
int turn = 0;
```

```
Process_A() {
    while (1) {
        while (turn != 0);
        critical_section;
        turn = 1;
        …
    }
}
```

```
Process_B() {
    while (1) {
        while (turn != 1);
        critical_section;
        turn = 0;
        …
    }
}
```

☹ A process/thread outside CS may prevent another thread from entering CS

Peterson's solution

```
#define N 2              //two processes/threads solution
int turn;                //current turn
int interested[N];       //initial value = 0, N: number of threads
```

```
enterCS (int process) {//process = 0 or 1
    int other = 1 – process;
    interested[process] = 1;
    turn = other;
    while (turn == other &&
          interested[other] == 1);
}
leaveCS (int process) {
    interested[process] = 0;
}
```

```
Process_A() {
  while(1){
    enterCS(0);
    critical_section;
    leaveCS(0);
  }
}
```

```
Process_B() {
  while(1){
    enterCS(1);
    critical_section;
    leaveCS(1);
  }
}
```

☹ CPU wasting

☹ Priority Inversion

Interrupt Disabling *Hardware support*

- Interrupts: signals emitted by a hardware or a software
  - ✓ Timer Interrupts: handle CPU Scheduling
  - ✓ I/O device Interrupts: inform I/O completion
  - ✓ ...

➔ *Perform a context switch*

- Interrupt-based solution for critical section problem

Disable interrupts
Enter CS
critical_section
Leave CS
Enable interrupts

No interrupts
→ No CPU Scheduling
→ No critical section problems

☹ What if a process/thread is blocked inside CS or time-consuming?
☹ What if systems have multiple processors?

TSL (Test-And-Set) *Hardware support*

```
Test_And_Set_Lock (bool lock) {
        bool test = lock;
        lock = true;
        return test;
}
bool lock = false;
Thread A {
        while (Test_And_Set_Lock ( lock ));
        critical_section;
        lock = false;
        return test;
}
```

Atomic operation, which cannot be suspended by OS scheduling

Semaphore

- **Integer variable** used to restrict access to a CS via two atomic operations: **down** and **up**
  - ✓ Down (also termed **P** or **wait**) called before entering a CS to verify if the calling thread has permission to enter the CS
  - ✓ Up (also termed **V** or **signal**) called when exiting CS to release this CS and wake up another sleeping thread (if there is any)

- Two types of semaphores
  - ✓ Binary semaphore
    - Alike to lock solution (also termed **mutex lock**).
    - Value ranging from 0 to 1
  - ✓ Counting semaphore
    - Used to control access to a resource having a finite number of instances
    - Initialized to the number of resources available

Semaphore (cont.)

```
semaphore S = value;
down(S)
critical_section
up(S)
```

```
typedef struct semaphore{
    int value;
    struct thread *list; //blocking threads
} semaphore;
```

```
down(semaphore *S) {
    S→value--;
    if (S→value < 0) {
        add calling thread to S→list;
        block();
    }
}
```

```
up(semaphore *S) {
    S→value++;
    if (S→value <= 0) {
        remove a thread A from S→list;
        wakeup(A);
    }
}
```

Semaphore (cont.)

```
semaphore mutex = 1;
processA (){
        down(mutex);
        critical_section
        up(mutex);
}
```

```
semaphore count = 5;
processX (){
        down(count);
        critical_section
        up(count);
}
```

```
semaphore mutex = 0; //synchronize execution order: threadB ➔ threadA
processA (){                              processB (){
        down(mutex);                              …
        …                                         up(mutex);
}                                         }
```

Monitor

- A programming construct for controlling access to shared data, which encapsulates:
  - ✓ Shared variables
  - ✓ Condition variables, along with 2 operations (**wait** and **signal**) for synchronization
  - ✓ Procedures operating on shared variables
- Also termed "thread-safe" class
  - ✓ Only procedures (*operations*) inside a monitor can access its variables
    - o Processes/Threads access shared variables by invoking procedures
  - ✓ Only one process/thread may be executing in the monitor at a time

Monitor (cont.)

```
monitor M{
    variable shared_item;
    condition c;

    procedure get_item () {
        ...c.wait();...
    }
    procedure putback_item () {
        ...c.signal();...
    }

    initialization_code
}
```

Entry queue for entering Monitor

← **processA** ← **processB** ← **...** ←

```
processX (){
        M.get_item();
            ...
        M.put_item();
}
```

And other ...

- Message-based solution
- Barrier

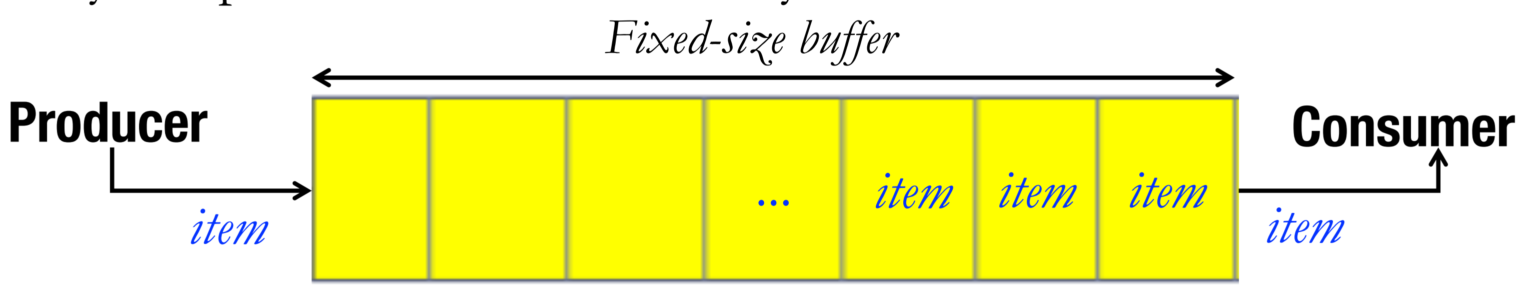
Self-learning

Plan

- Critical Section Problem
- Synchronization
- **Classic synchronization problems**

Producer-Consumer Problem

- Also known as **bounded-buffer problem**
- Producer and consumer share a fixed size buffer
  - ✓ Producer produces an item and places it in the shared buffer
  - ✓ Consumer picks an item from the shared buffer and consume it
- Regulations
  - ✓ Buffer is FULL → producer will be blocked
  - ✓ Buffer is EMPTY → consumer will be blocked
  - ✓ Only one producer or consumer may access the shared buffer at a time



Classic synchronization problems

Producer-Consumer Solution

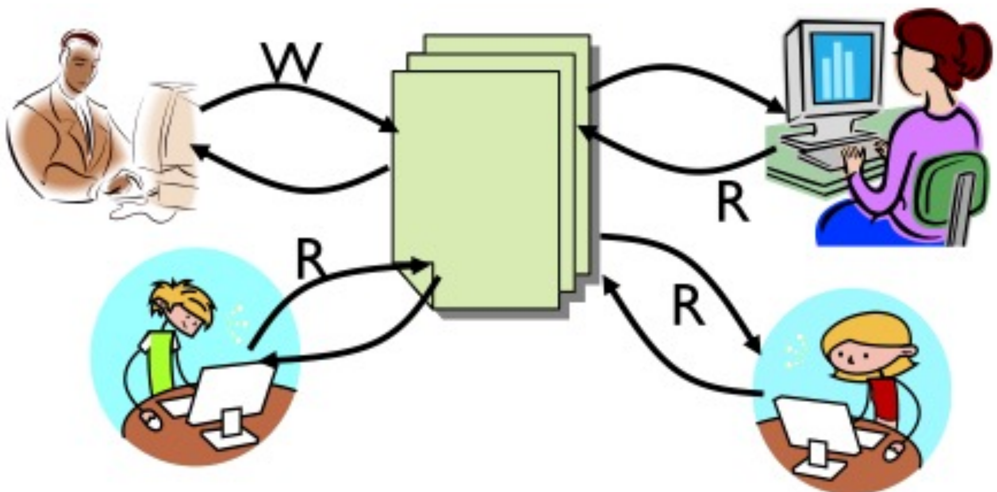
```
#define N 50
semaphore mutex, empty, full;
init (mutex, 1), init (empty, 0), init (full, N);
```

```
producer() {
        down(full);
        down(mutex);
        insertItem(item);
        up(mutex);
        up(empty);
}
```

```
consumer() {
        down(empty);
        down(mutex);
        item = removeItem()
        up(mutex);
        up(full);
}
```

Readers-Writers Problem

- Classic database problem: readers and writers share a database
- Regulations
  - ✓ Several readers can read from database at the same time
  - ✓ If a writer is writing to database, other threads cannot access database
  - ✓ If there are any readers in database, a writer cannot access database



Readers-Writers Solution

```
semaphore db, mutex;
init (db, 1), init (mutex, 1);
int nReaders = 0;
```

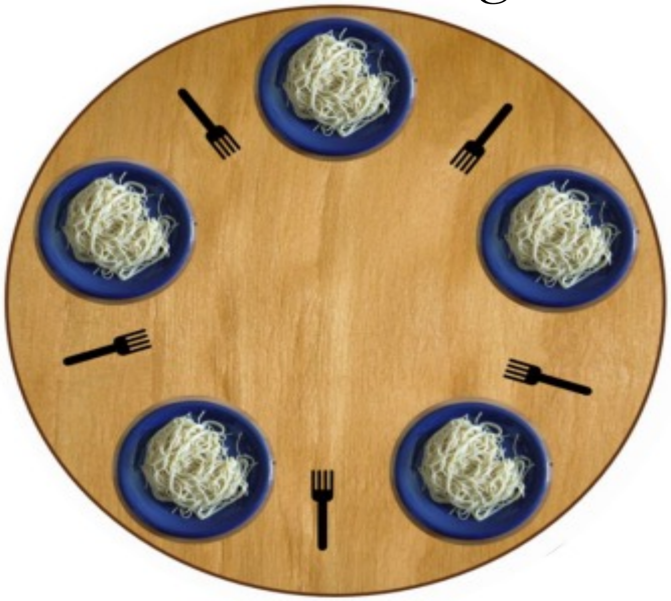
```
Writer ( ) {
    down(db);
    writeToDB( );
    up(db);

}
```

```
Reader ( ) {
    down(mutex);
    if (nReaders==0) down(db);
    nReaders+=1;
    up(mutex);
    readFromDB( );
    down(mutex);
    nReaders-=1;
    if (nReaders==0) up(db);
    up(mutex);
}
```

Dining Philosophers Problem

- Five philosophers are seated around a circular table, which is laid with five forks
- Each philosopher must take two nearest forks for eating, but:
  - ✓ He can only pick up one fork at a time
  - ✓ He cannot pick up the fork

    which has been taken by his neighbors



Classic synchronization problems

Dining Philosophers Solution

```
semaphore forks[5];                    //forks[1, 1, 1, 1, 1]
```

```
Philosopher (int i) {                  //i: philosopher number
    while(true) {
        down(forks[i]);                //take left fork
        down(forks[(i + 1)%5]);        //take right fork
        eating( );
        up(forks[i]);                  //put left fork back
        up(forks[(i + 1)%5]);          //put right fork back
    }
}
```

What if all five philosophers take left fork at once?

☹ Deadlock

Dining Philosophers Solution (cont.)

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING };
    state [5];
    condition self [5];
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait();
    }
    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
    void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
        }
    }
    initialization_code () {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

```
DiningPhilosophers.pickup(i);
        /** EAT **/
DiningPhilosophers.putdown(i);
```

☺ No deadlock

☹ Starvation