Operating system

We also call the mechanism to overcome this risk as synchronization

Part V: Synchronization risk [1]

Data Inconsistency

By KONG LingBo (孔令波)

Outline of topics covered by this course

- **INTRODUCTION** – why should we learn OS?
- **OVERVIEW** – what problems are considered by modern OS in more details?
- **EXECUTION** – CPU management
  - Process and Thread
  - CPU scheduling
- **EXECUTION** – competition (synchronization problem)
  - Synchronization (Data Inconsistency – shared data access)
  - Deadlock (Resource competition – shared resources allocation)

Goals

- Know the related concepts and definitions
  - Race conditions, Critical sections, and Atomic operations etc
- Know the mechanism of different solutions
  - Software solutions – algorithms who's correctness does not rely on any other assumptions.
    - Peterson's algorithm
  - Hardware solutions – rely on some special machine instructions.
    - TestAndSet(), Swap()
  - Operating System solutions – provide some functions and data structures to the programmer through system/library calls.
    - Semaphores
  - Programming Language solutions – Linguistic constructs provided as part of a language
    - Monitors
- Understand the codes for some classic synchronization problems
  - Producer-Consumer, Thinking philosophers, Sleeping barbers

Synchronization

- Background & basic concepts
  - Race conditions, Critical sections, etc.
- Problems & Solutions for synchronization
  - Problems
    - Producer-Consumer problem, Readers-Writers Problem, The Barbershop Problem, Dining philosopher problem
  - Tasks
    - Mutual exclusion, deadlock-free, starvation-free
  - Solutions
    - **LOCK** mechanism is the basis (for mutual exclusion)
    - **PV** (Signal-Wait) operations are the first classic prototype
    - **SEMAPHORE**/**MONITOR** (for efficiency & convenience)

Problems with Concurrent Execution

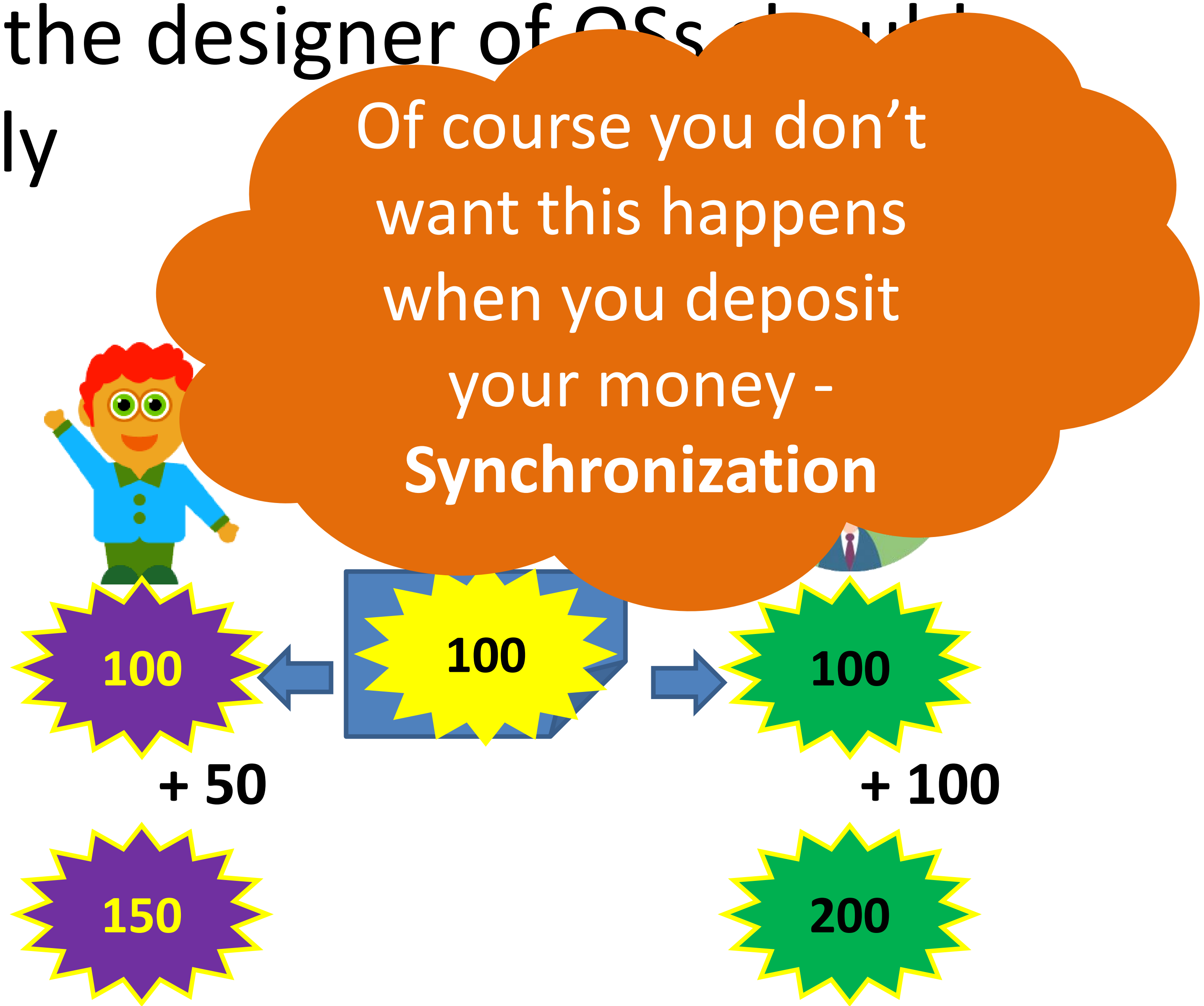
- Concurrent processes (or threads) often need to **share data** (maintained either in shared memory or files) and **compete resources**
  - If there is no proper policy to assign resources among processes, it may result in that all the processes get blocked → **Deadlock** [死锁]
- If there is **no controlled access** to shared data, execution of the processes on these data can inter… **…ration.**
  - **The results will then depe d o the … ere modified→ Data Inconsistency**
    - i.e. the results are non-deterministic.

It always implies
“Share resources”
at the same time

PPTs from others\flame.cs.dal.ca_~hawkey_312… …ronization.ppt

Some functions the designer of OSs should consider carefully

- Resource competition
  - **Data inconsistency** is another issue we should consider carefully when competing.



Summarized as **<u>Mutual Exclusion</u>** [互斥] problem

http://en.wikipedia.org/wiki/Mutual_exclusion

- Mutual exclusion, in computer science, refers to the problem of **ensuring that no two processes or threads** (henceforth referred to only as processes) **can be in their code to access the shared data among those processes at the same time**
  - It was formally defined by **Edsger Dijkstra** in 1965.



Died at
August 6, 2002
(aged 72)

Turing award 1972

http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

For memorizing: **Edsger Wybe Dijkstra**

- Among his contributions to computer science are
  - the **shortest path algorithm**, also known as **Dijkstra's algorithm**;
  - Reverse Polish Notation and related Shunting yard algorithm;
  - **the THE multiprogramming system**, an important early example of structuring a system as a set of layers; "**T**echnische **H**ogeschool **E**indhoven"
  - **Banker's algorithm**;
  - and the **semaphore construct** for coordinating multiple processors and programs.

We call that situation as Race Condition

- A situation in which multiple threads or processes read and write a shared data item and **the final result depends on the relative timing of their execution**

Initial balance: $100

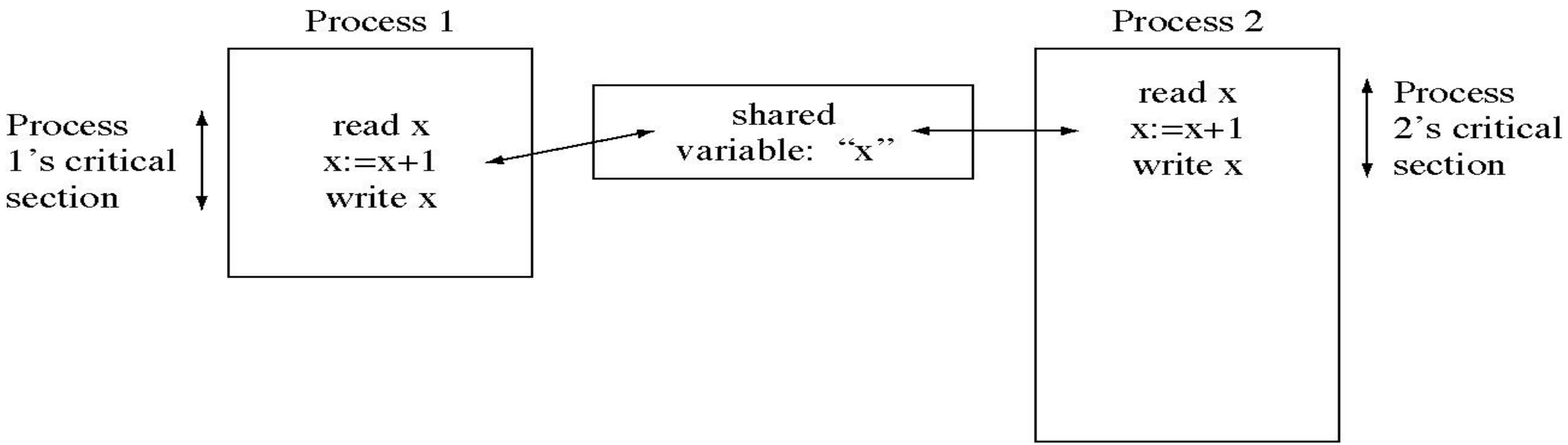
Thread 1:deposit(10)	Time line	Thread 2:deposit(20)
Load R1, balance		
		Load R1, balance
Add R1, amount		
		Add R1, amount
Store R1, balance		
		Store R1, balance

What is the final balance?

A condition in which the value of a shared data item d_s resulting from execution of operations a_i and a_j on ds in interacting processes may be different from both $f_i(f_j (d_s))$ and $f_j (f_i(d_s))$.

We call those codes as Critical sections

- A **section of code within a process** that requires **access shared resources** and which may not be executed while another process in a corresponding section of code



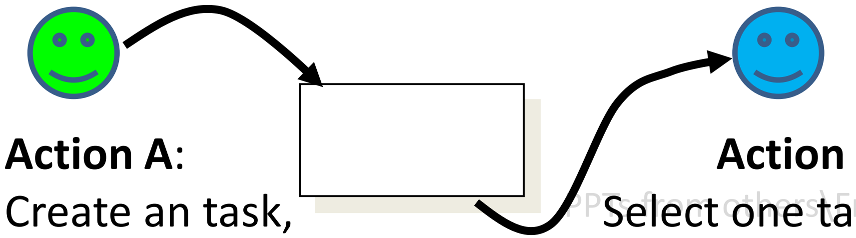
CLASSIC SYNCHRONIZATION MODELS

- **Producer-Consumer model**
- **Readers-Writers Problem**
- **The Barbershop Problem**
- **Dining philosopher problem**

Producer/Consumer (P/C) Problem

- Producer/Consumer is a common paradigm for **cooperating processes** in OS
  - Producer process produces information that is consumed by a Consumer process.
- **Example 1**: a print program produces characters that are consumed by a printer.
- **Example 2**: an assembler produces object modules that are consumed by a loader.

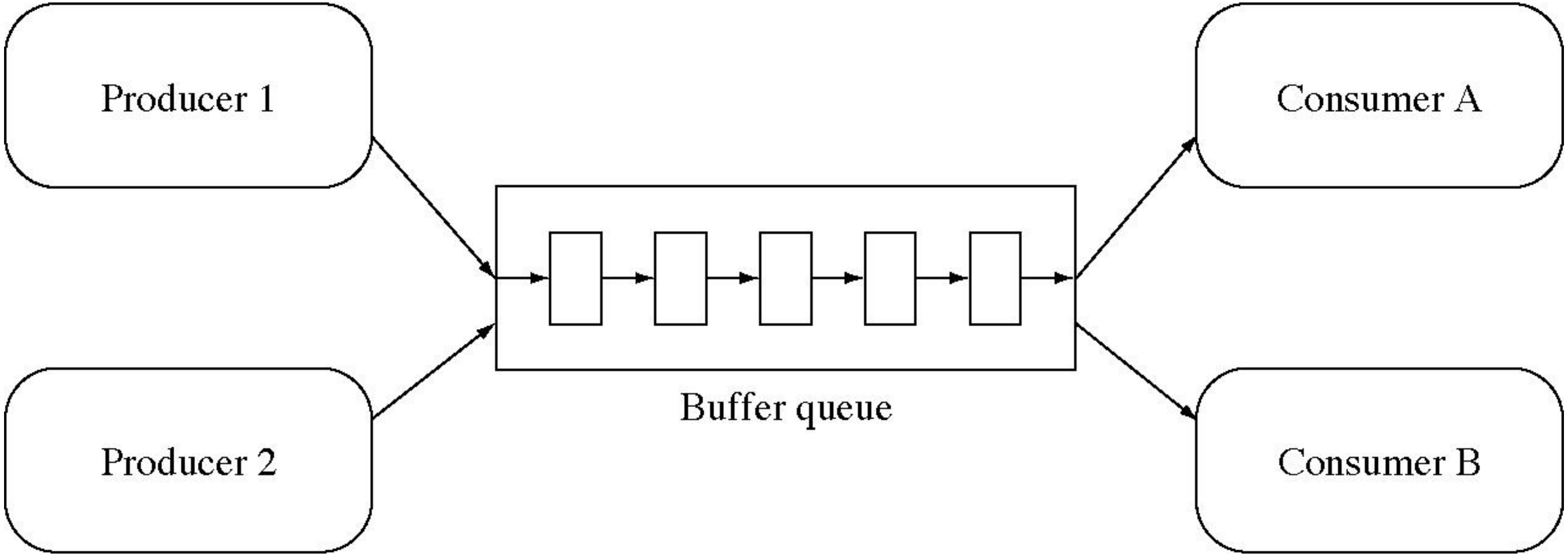
Two actions model



Action A:
Create an task,
and put it into the
waiting zone

Action B:
Select one task
from the waiting
zone, and serve it

Multiple Producers and Consumers



The key of synchronization (access to the shared data)

TO CONTROL THE EXECUTION OF CRITICAL SECTIONS AMONG CONCURRENT PROCESSES/THREADS

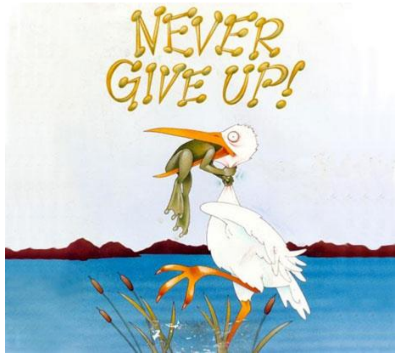
⇨ Ensuring **Mutual Exclusion**

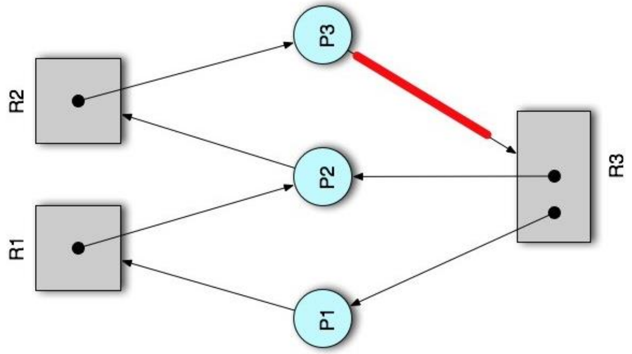
Every time, no more than one process is accessing the shared data

But we also should avoid

For **EFFICIENCY**

- **Deadlock**
  - None of the involved processes could move further





- **Starvation**
  - Some process is never executed!
  - Process/ thread waits indefinitely

Rules for robust synchronization

- Of course, **Mutual exclusion** should be guaranteed (**consistency**) [互斥]
  - – No more than one process/thread in critical section at a time
- Progress (**deadlock-free**) [有空让进]
  - – If several simultaneous requests, must allow one to proceed
  - – Must not depend on threads outside critical section
- Bounded (**starvation-free**) [有限等待]
  - – Must eventually allow each waiting thread to enter

PPTs.2012\PPTs from others\pages.cs.wisc.edu_~remzi_Classes_537_Fall2005\lecture4.ppt

Synchronization

- Background & basic concepts
  - Multiprogramming/Concurrency + Cooperation
  - Race conditions, Critical sections, and Atomic operations etc.
- Problems & Solutions for synchronization
  - Problems
    - Producer-Consumer problem, Readers-Writers Problem, The Barbershop Problem, Dining philosopher problem
  - Tasks
    - Mutual exclusion, deadlock-free, starvation-free
  - Solutions
    - **LOCK** mechanism is the basis (for mutual exclusion)
    - **PV** (Signal-Wait) operations are the first classic prototype
    - **SEMAPHORE/MONITOR** (for efficiency & convenience)

Types of solutions to CS problem

- **Software** solutions –
  - algorithms who’s correctness does not rely on any other assumptions.
- **Hardware** solutions –
  - rely on some special machine instructions.
- **Operating System** solutions –
  - provide some functions and data structures to the programmer through system/library calls.
- **Programming Language** solutions –
  - Linguistic constructs provided as part of a language.

ALL ARE TRYING TO IMPLEMENT
LOCK MECHANISM!

Solution t … Locks

- T … is:

You also should ensure the “acquire” and “release” are primitive operations.

do {

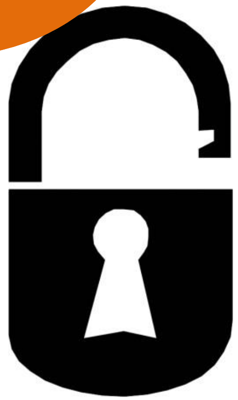
acquire **lock**

critical section

release **lock**

remainder section

} while (TRUE);



Software Solutions

- We consider first the case of **2** processes:
  - **Peterson's algorithm** is correct.
- Then we generalize to **n** processes:
  - The **Bakery algorithm**.
- Initial notation:
  - Only 2 processes, P_0 and P_1
  - When usually just presenting process P_i (Larry, I, i), P_j (Jim, J, j) always denotes other process (i != j).

- One attempt to implement lock

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

busy-wait (spin-wait) for lock to be released

  – This is called a **spin-lock** because a thread spins waiting for the lock to be released
  – Does this work?

- No.
  - Two independent threads may both notice that a lock has been released and thereby acquire it.

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition

The Critical-Section Problem: 2-Processes: Algorithm N (**Peterson 1981**)

```
bool flag[2] = {false, false};
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] && turn ==
1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] && turn ==
0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

- Meets all three requirements;
- solves the critical-section problem for two processes.

https://zh.wikipedia.org/wiki/Peterson%E7%AE%97%E6%B3%95
https://en.wikipedia.org/wiki/Peterson's_algorithm

Peterson's Algorithm: Proof of Correctness

- Mutual exclusion holds since:
  - For both P_0 and P_1 to be in their CS
    - both flag[0] and flag[1] must be true and turn=0 and turn=1 (at same time) ➔ this is **impossible**

- Another demonstration

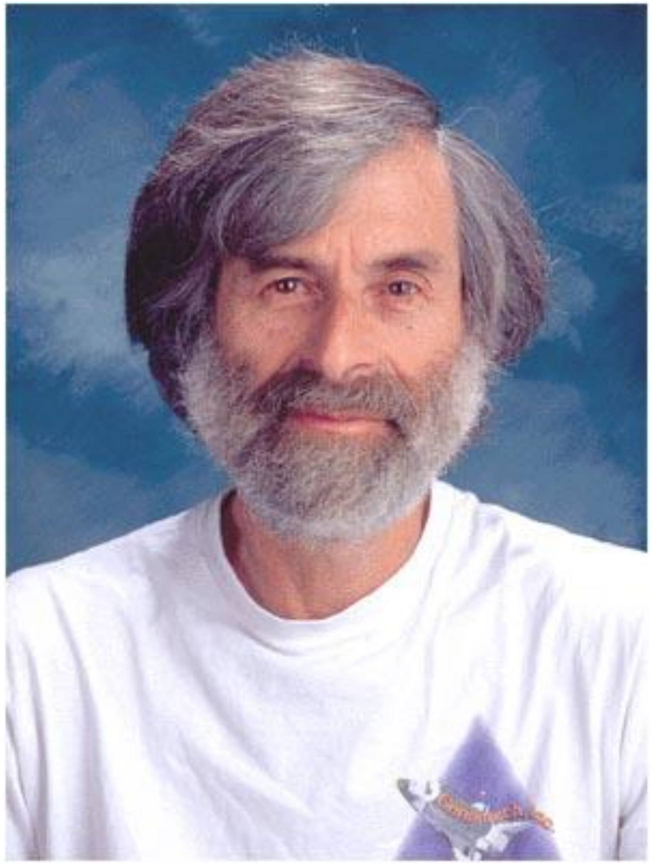
```
struct lock {
    int turn = 0;
    int interested[2] = [FALSE, FALSE];
}
void acquire (lock) {
    lock->interested[this_thread] = TRUE;
    turn = other_thread;
    while (lock->interested[other_thread] && turn==other_thread);
}
void release (lock) {
    lock->interested[this_thread] = FALSE;
}
```

How about for N-processes?

http://en.wikipedia.org/wiki/Leslie_Lamport

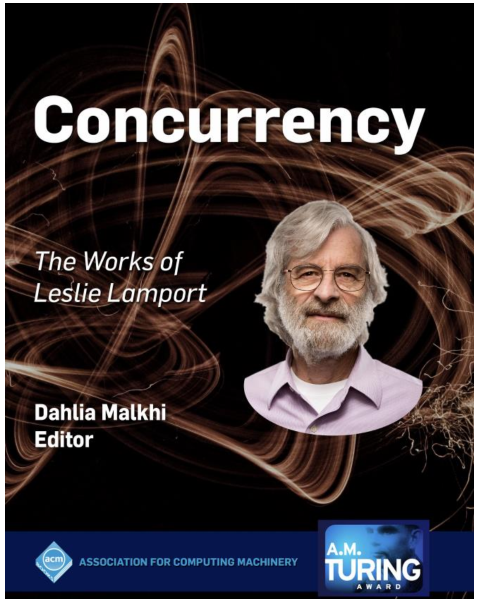
- Critical section for n processes
  - We have **Bakery algorithm**
  - It is also known as **Lamport's** bakery algorithm. Lamport?
  - Dr. **Leslie L**
    - I know him
    - You can check ... om my homepage.

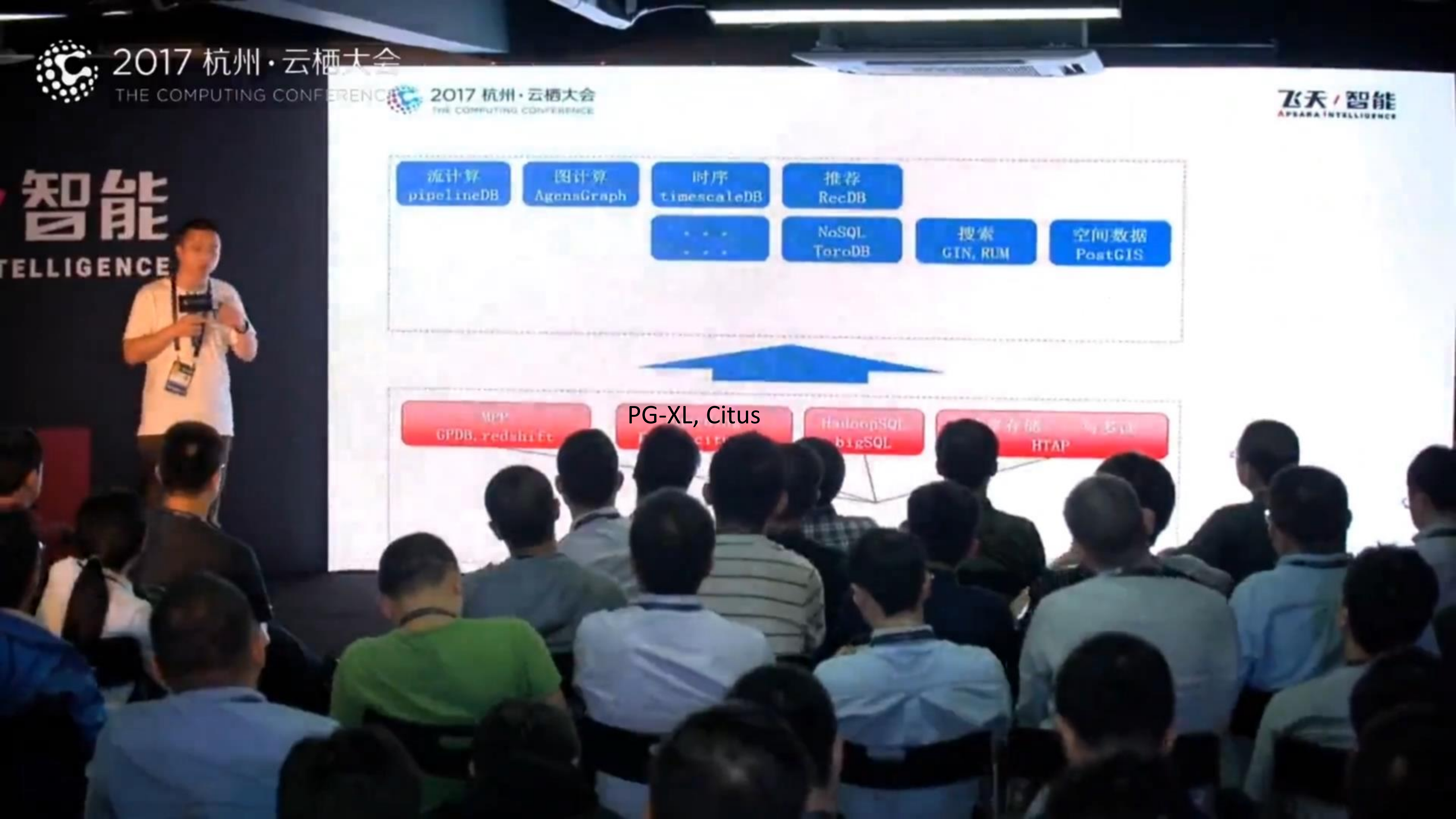
You could learn this by yourself and finish a report.



PAXOS

- Lamport also proposed PAXOS protocol for synchronization in distributed system (called **Consensus**)
  - In the paper "The Part-Time Parliament" published in 1988
  - He rewrite a paper "Paxos Made Simple" in 2001 for he believed the previous one is hard for readers to get his humor
  - He discussed several situations when meeting distributed synchronization in an island he named it "PAXOS"
- Mr. Leslie Lamport got his **Turing prize in 2013** with PAXOS
  - Now work in Microsoft




2017 杭州·云栖大会
THE COMPUTING CONFERENCE
智能
TELLIGENCE
2017 杭州·云栖大会
THE COMPUTING CONFERENCE
飞天/智能
APSARA INTELLIGENCE
流计算
pipelineDB
图计算
AgensGraph
时序
timescaleDB
推荐
RecDB
. . .
. . .
NoSQL
ToroDB
搜索
GIN, RUM
空间数据
PostGIS
MPP
GPDB, redshift
PG-XL, Citus
HadoopSQL
bigSQL
HTAP

2017 杭州·云栖大会
THE COMPUTING CONFERENCE
WORLDWIDE PARTNER
内核专场
Session
核心技术
Global Shopping Festival
2017 杭州·云栖大会
THE COMPUTING CONFERENCE
X-Cluster部署案例——Multi-Region
真正Region级的强一致能力
单个Region不可用0数据丢失
高性能
跨Region强同步下依然保持高性能
灵活的切换策略
优先切换同Region
定制跨Region切换顺序
高伸缩性
可无限制的扩充Region/AZ的节点
可自由的调节Region/AZ内节点
Region 1
App
AZ 1
AZ 2
Leader
Follower
Region 2
App
AZ 3
AZ 4
Loger
Follower
Region 3
AZ 5
Loger
intel


2017 杭州·云栖大会
THE COMPUTING CONFERENCE
数据库内核专场
Alibaba Database Kernel Session
十一背后的数据库核心技术
base Technology Behin 11.11 Global Shopping Festival
X-DB 2.0 基于共享存储的分布式数据库
设计方法：
✓ 基于X-Engine引擎
✓ 基于X-Paxos
能力：
✓ 存储、只读扩展
✓ 高性能
✓ 兼容MySQL生态
Read Node
Write Node
Read Node
X-Engine
Block Cache
Memtable
Meta
Pangu
Data Log
SSTable Data
Data Log
Meta Log
Control Log
intel

2017 杭州·云栖大会
THE COMPUTING CONFERENCE
POLARDB 赋能多场景应用
POLARDB Enables Multi-Application Scenarios
飞天 / 智能
商用数据库
RDS MySQL
Backup 物理复制
DTS/BLS 上云
DTS/BLS 迁移
OSS
上海可用区1
上海可用区2
同城容灾
POLARDB 只读节点
POLARDB 主节点
高速网路
节点计算能力 Scale Up/Down
异地容灾
应用服务器
OLTP 请求
集群数据库连接 读写分离负载均衡
北京可用区1
同城容灾
北京可用区2
R/W
R/O
POLARDB 主节点
POLARDB 只读节点
集群并发能力 Scale Out/In
Failover
高速网络
低延迟、高吞吐

Drawbacks of Software Solutions

- Even software solutions are very delicate ☺, they are complicated to program ☹
- Busy waiting (wasted CPU cycles)
  – It would be more efficient to *block* processes that are waiting (just as if they had requested I/O).
  – This suggests implementing the permission/waiting function in the Operating System → Semaphores & Monitors

In DBMS, the **SELF-SPIN LOCK** [自旋锁] is stilled used - a kind of **BUSY WAITING** lock

Hardware solutions

- Many systems provide hardware support for critical section code.
- Uniprocessor – could **disable interrupts**:
  - Currently running code would execute without preemption.
  - Generally too inefficient on multiprocessor systems.
- Modern machines provide special **atomic** (non-interruptible) hardware instructions:
  - Either test memory word and set value at once.
  - Or swap contents of two memory words.

PPTs from others\From Ariel J. Frank\OS381\os4-3_cop.ppt

Hardware Solution ①: Disable Interrupts

```
Process Pi:
repeat
  disable interrupts
   critical section
  enable interrupts
   remainder section
forever
```

- On a uniprocessor, mutual exclusion is preserved : while in CS, nothing else can run
  - because preemption impossible
- On a multiprocessor: mutual exclusion is not achieved
  - **Interrupts are "per-CPU"** (interrupts are not disabled on other processors).
- Generally not a practical solution for user programs, but could be used inside an OS

Hardware Solution ②: **Special Machine Instructions**

- Normally, the memory system restricts access to any particular memory word to one CPU at a time
- Useful extension:
  - machine instructions that perform 2 actions *atomically* on the same memory location (ex: testing and writing)
- The execution of such an instruction is *mutually exclusive* on that location (even with multiple CPUs)
- These instructions can be used to provide mutual exclusion
  - but need more complex algorithms for satisfying the requirements of progress and bounded waiting

PPTs from others\flame.cs.dal.ca_~hawkey_3120\May26ConcurrencyCont.ppt

Test-and-Set Synchronization Hardware

- Test and set (modify) the content of a word atomically (a Boolean version):

```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

- The Boolean function repr… the corresponding machin…

What is returned when target = TRUE?

PPTs from others\From Ariel… 81\os4-3_cop.ppt

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = FALSE;

- Process P_i

  do {

  while (TestAndSet(&lock));

  critical section

  lock = FALSE;

  remainder section

  }

PPTs from others\From Ariel J. Frank\OS381\os4-3_cop.ppt

Swap Synchronization Hardware

- Atomically swap two variables:

```
void Swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

- The procedure represents the essence of the corresponding machine instruction.

PPTs from others\From Ariel J. Frank\OS381\os4-3_cop.ppt

- Mutual Exclusion with Swap
  - Shared data (initialized to false):

    boolean lock = false;

  - Process Pi

```
do {
        key = true;         // key is a local variable
        while (key == true)
                Swap(lock,key);
        critical section
        lock = false;
        remainder section
} while (TRUE);
```

Machine Instructions for Mutual Exclusion

- Advantages
  - **<u>Applicable to any number of processes</u>** on either a single processor or multiple processors ***sharing main memory***
  - It is simple and therefore easy to verify
- Disadvantages
  - **<u>Busy-waiting</u>** consumes pro
  - Starvation is possible when a pro
    process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Any idea to overcome these disadvantages?

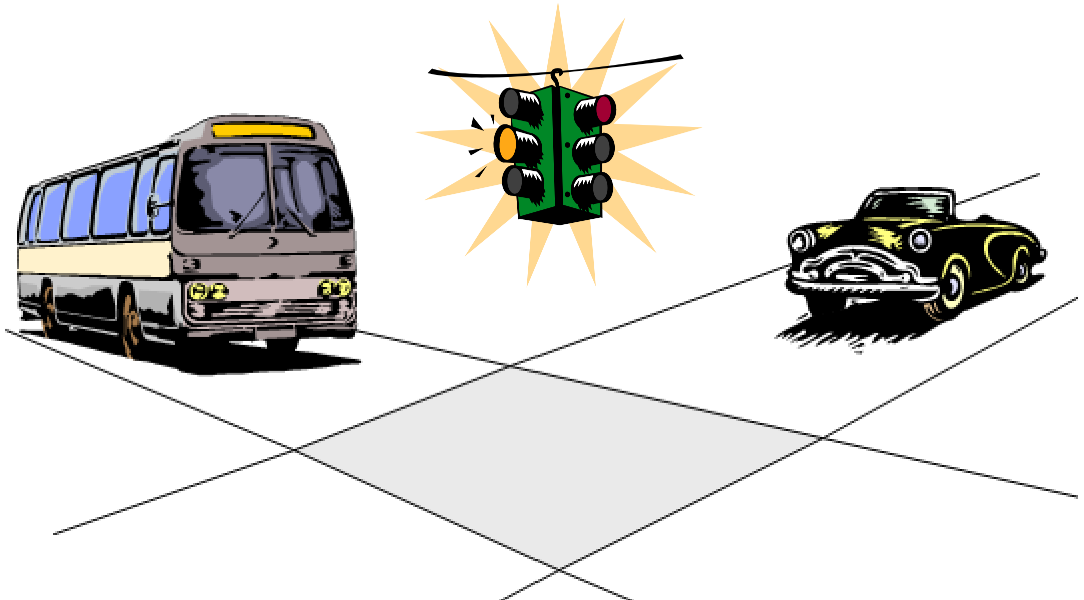
PPTs from others\OS5e after William Stallings\Chapter05.ppt

Lock + Manager + Waiting Room

- If A tries to use the shared variable, its request will be trapped by the manager, and it's manager's responsibility to check if the variable has been occupied or not
  - If yes, A will be put in waiting room; otherwise, A goes into its CS
- After A finishes its job, it should release the lock, and manager will check if there are waiting processes in WR for this lock (resource)

Semaphore - Operating System solutions

- Semaphore [信号量]
  - Software construct that can be used to enforce mutual exclusion
  - Contains a protected variable
    - Can be accessed only via wait and signal commands
    - Also called P and V operations, respectively



Semaphores

- A Semaphore **S** is an integer variable that, apart from initialization, can only be accessed through 2 *atomic and mutually exclusive* operations:
  - wait(S)
    - sometimes called **P()**
      - Dutch ***proberen***: “to test”
  - signal(S)
    - sometimes called **V()**
      - Dutch ***verhogen***: “to increment”

Semaphore & Edsger W. Dijkstra

http://en.wikipedia.org/wiki/Edsger_Dijkstra
http://en.wikipedia.org/wiki/Semaphore_%28programming%29

- Invented in the 1965
  - **Basis of all contemporary OS synchronization mechanisms**
- In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment.
  - A semaphore generally takes one of two forms: **binary** and **counting**.
- Either semaphore type may be employed to prevent a race condition.
  - On the other hand, a semaphore **is of no value in preventing resource deadlock**, such as the dining philosophers problem.

http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

May 11, 1930 – August 6, 2002

Binary Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

We can use BS to

1. Support **Mutual Exclusion [MU]** for critical section problem of course
   – Shared data:

     semaphore mutex; // initiallized *to* 1

   – Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (TRUE);
```

```
void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
```

We can also use BS to

2. Support **ordered execution** of two processes (**Order Scheduling [SCH]**)
   - Execute B in P_j only after
   - Use semaphore *fl*
   - Code:

P_j will wait P_i to finish A first, then P_j can execute B

P_i	**P_j**
⋮	⋮
A	***wait(flag)***
signal(flag)	***B***

```
semaphore s1 = 0;
semaphore s2 = 0;
```

```
A() {
  write(x);
  V(s1);
  P(s2);
  read(y);
}
```

```
B() {
  P(s1);
      read(x);
      write(y);
  V(s2);
}
```

Counting Semaphore

```
struct semaphore {
      int count;
      queueType queue;
}

void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0)
      {
            place this process in s.queue;
            block this process
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0)
      {
            remove a process
            place process P on
      }
}
```

So s.Count value indicates the number of waiting processes?

Figure 5.3 A Definition of Semaphore Primitives

- In a computer system, there is 1 printer, and a semaphore S is used to control the share of the printer among many processes. Among following values, _______________ is impossible to be true for the semaphore.
- a. -1000 b. 0 c.1 d.2

Counting semaphores

- Initialized with values **larger than one**
- Can be used to **control the competition access to a pool of identical resources**
  - Decrement the semaphore's counter when taking resource from pool ← wait()
    - If no resources are available, ...
      becomes available
  - Increment the semapho... ...ol
    → signal()
    - If there are processes waiting for this type of resources, wake up one of them

Two types of competitions – ME and OrderS

Implementation

- A semaphore can be defined as a C struct along these lines:

```
typedef struct {
    int value;
    struct process *list;
} semaphore
```

Implementation

- down() operation can be defined as

```
down(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
    }
```

- The block() operation suspends the process that invokes it.

Implementation

- up() operation can be defined as

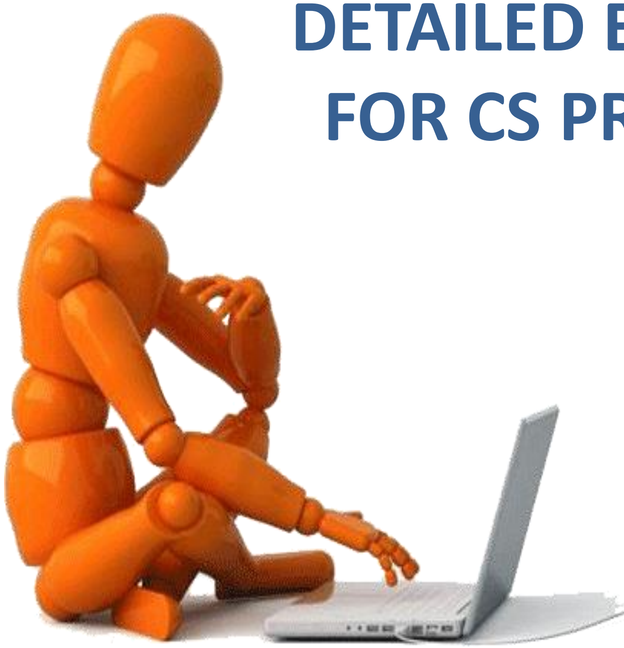
```
up(semaphore *S) {
        S->value++;
        if (S->value ≤ 0) {
                remove process P from S->list;
                wakeup(P);
        }
}
```

- The wakeup() operation sends a signal that represents an event that the invoking process is no longer in the critical section

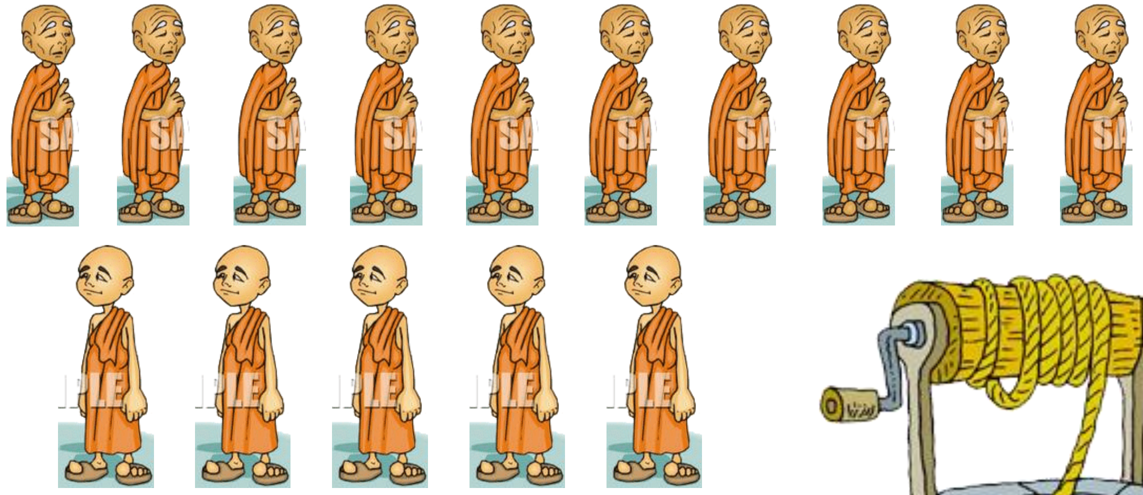
Implementation

- BTW, the implementation just described is how Linux implements semaphores
- The up and down operations represent require access to a critical section which is the semaphore variable
- Need hardware/OS support e.g.,
  - Signals, TSL
  - Signals allow for a “message” to be sent to processes

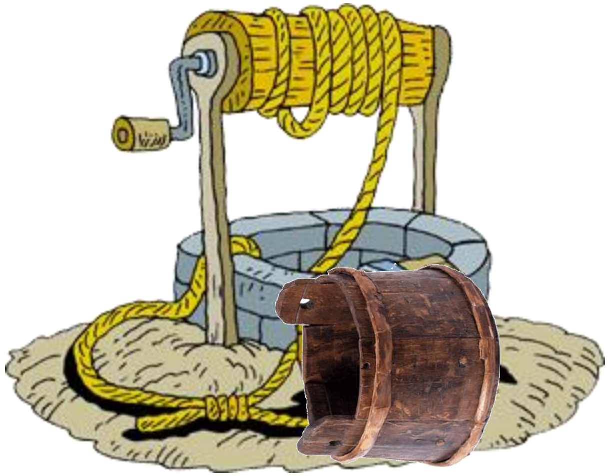
DETAILED EXAMPLE FOR CS PROBLEM



A story: monks drink water



- Many monks
  - Some are **old**
  - Some are **young**
- One well
  - Only one bucket in well every time



- One Vat
  - Can contain 10 buckets of water
  - One bucket to put water into and fetch water from the vat

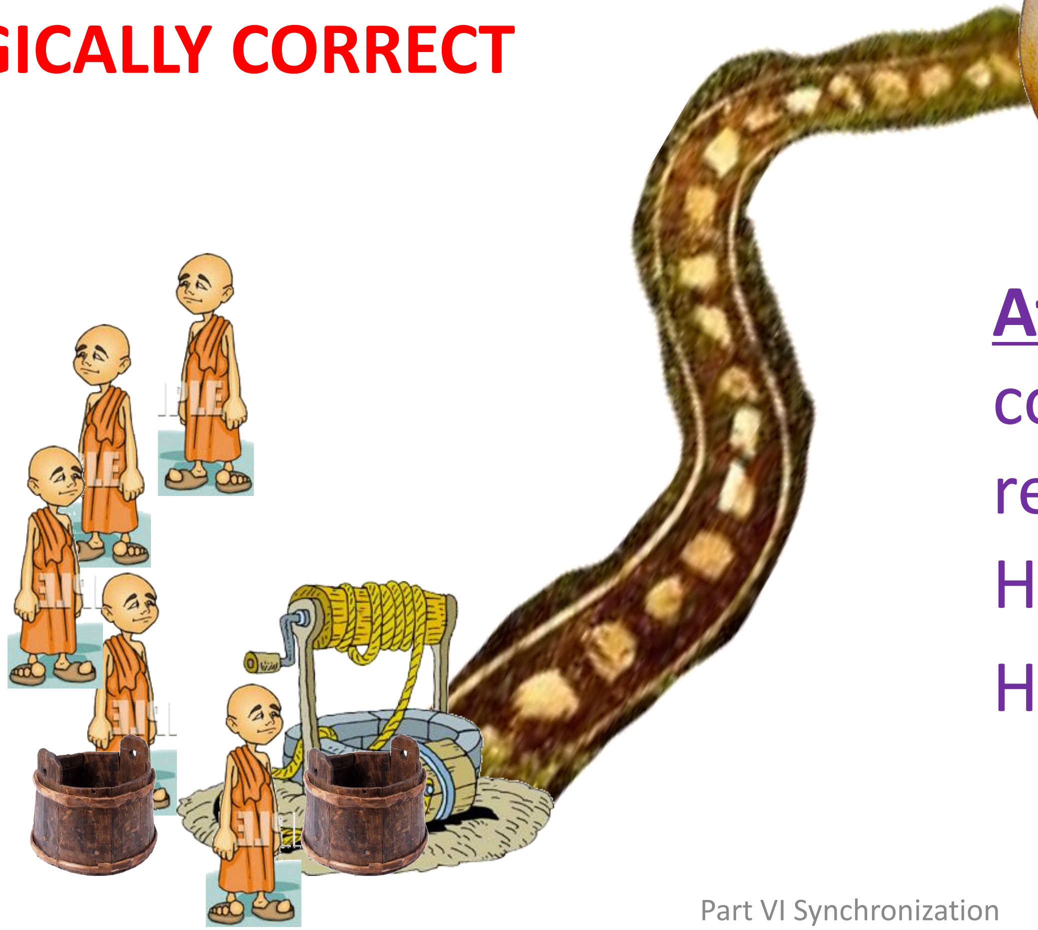
- Three buckets in total

General rules to cope with CS problem using semaphores

1. Find the types of actors
   - To determine the **processes**
2. Recognize the shared resources between actors → initial values of semaphores
3. Conclude the constraints based on the situations when actors use those shared resources
   - **ME or SCH**?
     - To determine semaphores and their initial values
     - To determine the code (nested for ME, and scattered for SCH)
4. Use semaphores to finish those processes

<u>Good habits</u>: write down the detailed steps of the actors to carry out the story, especially clarifying the constraints they should obey

LOGICALLY CORRECT



<u>Attention</u>： they have to compete the shared resources!!

How about the young?

How about the old?

In our story

1. Find the types of actors
   - Two types **actors** (**processes**): young monks and old monks
     - The young picks a bucket, goes to the well to fetch water and puts the water into the vat
     - The old picks a bucket to get water from the vat
2. Recognize **the shared resources** between actors
   - Three types of shared resources:
     - **Vat**: among all monks
     - **Well**: among young monks
     - **3 buckets**: among all monks (because all monks should get one bucket to pick water from the well or vat)

(cont')

3. Infer the **constraints** based on the situations when actors use those shared resources (**ME** or **SCH**?) – checking SCH is always first!
   – Conclude the action scripts
   - For young:
     1. Check if the vat could contain more buckets of water. No, no more action – wait the old to consume the water (at first, the vat could contain **10** buckets of water →**SCH** for the young).
     2. Compete a bucket (→ **ME**: share buckets with other monks)
     3. Go to the well and get a bucket of water from the well (→ **ME** of well: only one bucket could access the well every time)
     4. Try to pour the bucket of water back to vat (→ **ME** of vat:)
     5. Release the bucket
     6. Inform the old that they could consume the water (←**SCH:** for the old: initially, the vat has 0 bucket of water) .

(cont')

3. Infer the constraints based on the situations when actors use those shared resources (ME or SCH?) – checking SCH is always first!
   - Conclude the action scripts
     - For old:
       1. Check if the vat contains water or not (→**SCH:** for the old: initially, the vat has 0 bucket of water)
       2. Compete a bucket (→ **ME**: share buckets with other monks)
       3. Try to fetch a bucket of water from the vat (**ME** of vat:)
       4. Release the bucket
       5. Inform the young because the vat now could contain one more bucket of water (←**SCH**: for the young. at first, the vat could contain **10** buckets of water)

(cont')

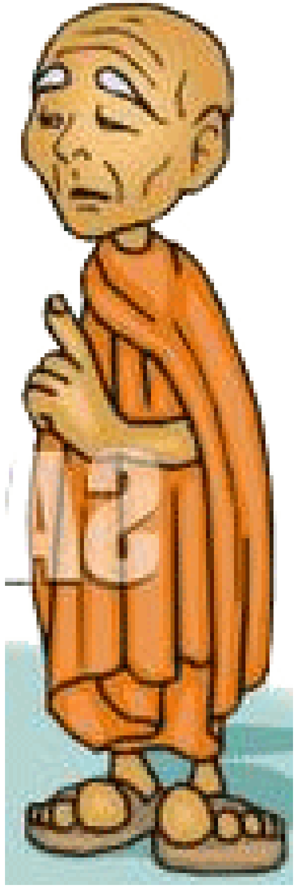
4. Use semaphores to finish those processes
   - Binary semaphores for **ME**: Well, vat
   - Counting semaphore: idleBuckets (=3) for **<u>ME</u>**, VatCouldContain (= 10), WaterInVat (= 0) for **<u>SCH</u>**
     - For young:
       1. when VatCouldContain =0, the young should not compete the buckets;
       2. Once a young gets a bucket, he should finish 2 steps wholly: fetch a bucket of water and put that water into the vat
       3. Once he finishes, the occupied bucket will be released and the WaterInVat will be incremented
     - For old:
       1. when WaterInVat=0, the old should not compete the buckets
       2. Once an old get a bucket, he drinks that water, releases that bucket and decremented the VatCouldContain

- For the young:

```
P(VatCouldContain) // initial = 10
    P(idleBuckets); // initial = 3
        P(well);
            // get a bucket of water
        V(well);
        P(vat);
            // put the water into the vat
        V(vat);
    V(idleBuckets);
V(WaterInVat); // initial = 0
```

- For the old:

```
P(WasterInVat) // initial = 0
   P(idleBuckets); // initial = 3
      P(vat);
         // get a bucket and drink the water
      V(vat);
   V(idleBuckets);
V(VatCouldContain); // initial = 10
```



DETAILED EXAMPLE FOR CS PROBLEM

Another problem

- The unisex bathroom problem
  - there should **never be** more than 10 people in the bathroom at once
  - there should **never be** both males and females in the bathroom at once
  - there is **no** starvation and **no** deadlock

In the story

- Shared resources and constraints
  - The **bathroom**: accessed by male and female for 10 persons
  - Empty bathroom **constraint**: for both male and female
  - **Occupation state**: occupied by male or female
    - Once the 1st one (male or female) enters the bathroom, he (or she) should change the state of using bathroom (from empty to occupied) so that the bathroom can only be used by male (or female).

- Script for male:
  - Try to enter the bathroom;
  - If this is the 1^{st} (the number of male is 0), indicate the bathroom is occupied now for male;
  - Number of male ++ (this increment should be ME)
  - Other males (10 at most) could further use the bath room
  - Once finished, a male leaves the bathroom (Number of male – [this increment should be ME])
  - When this is the last male (the number of male is 0), indicate the bathroom now is available

- int num_men = 0, num_women = 0;
- semaphore male_mutex = 1;
- semaphore female_mutex = 1;
- semaphore male_token = 10;
- semaphore female_token = 10;
- semaphore S = 1

```
male() {
    while (1) {
        wait(male_mutex);
        if (num_men ==0) wait(S);
        num_men = num_men + 1;
        signal(male_mutex);

        wait(male_token);
        // use the bathroom
        signal(male_token);

        wait(male_mutex);
        num_men = num_men – 1;
        if (num_men ==0) signal(S);
        signal(male_mutex);
    }
}
```

```
female() {
    while (1) {
        wait(female_mutex);
        if (num_women ==0) wait(S);
        num_women = num_women + 1;
        signal(female_mutex);

        wait(female_token);
        // use the bathroom
        signal(female_token);

        wait(female_mutex);
        num_women = num_women – 1;
        if (num_women ==0) signal(S);
        signal(female_mutex);
    }
}
```

Try this by yourself

1. Figure 1 illustrates a bridge, and the arrows show the directions of the corresponding cars. Only one car is allowed on the bridge at any time, but several cars are allowed to pass the bridge one by o ne if they are for the same direction. You are required to fill the blanks in following code which is used to cope with thi synchronization problem using P and V operations.

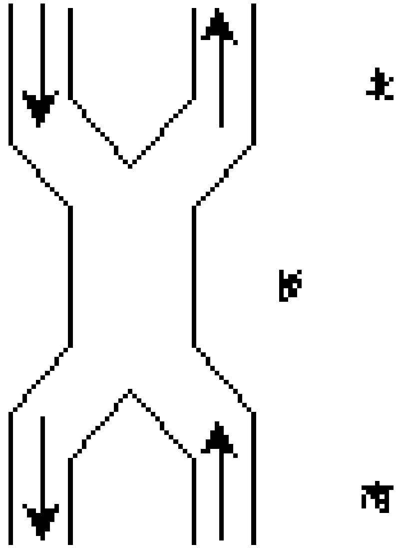
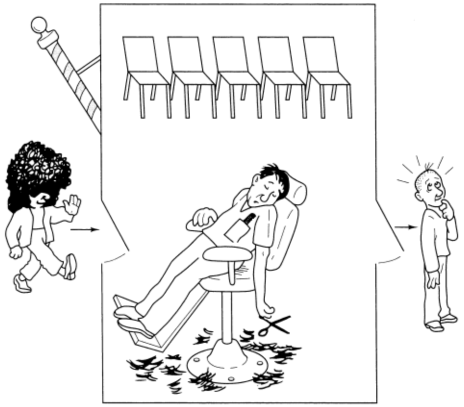


Figure 1: A bridge (Direction for top is north)

CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-buffer problem
  - (有限缓存问题)
- Readers-writers problem
  - (读者-著者问题)
- Dining-philosophers problem
  - (哲学家就餐问题)
- Barbershop problem
  - (理发师问题)

Your turn to read and understand these classic CS problems!



PPTs from others\OS PPT in English\ch07.ppt

Be careful: Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of waiting processes.
  - Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	*wait*(Q);
wait(Q);	*wait*(S);
⋮	⋮
signal(S);	*signal*(Q);
signal(Q)	*signal*(S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue (say, if LIFO) in which it is suspended.
- **Priority Inversion** – scheduling problem when lower-priority process holds a lock needed by higher-priority process

Monitors [管程] - Programming Language solutions

- Monitor: Hide Mutual Exclusion
  - No need for users to explicitly call the related functions
    - Declare a monitor which hide the details of synchronization, and provide friendly interface
  - Only one process may be active within the monitor at a time.
- Found in many concurrent programming languages:
  - Concurrent Pascal, Modula-3, C++, Java...
- Can also be implemented by semaphores.

```
procedure Producer
begin
   while true do
   begin
      produce an item
      ProdCons.Enter();
   end;
end;

procedure Consumer
begin
   while true do
   begin
      ProdCons.Remove();
      consume an item;
   end;
end;
```

```
monitor ProdCons
   condition full, empty;

   procedure Enter;
   begin
      if (buffer is full)
         wait(full);
      put item into buffer;
      if (only one item)
         signal(empty);
   end;

   procedure Remove;
   begin
      if (buffer is empty)
         wait(empty);
      remove an item;
      if (buffer was full)
         signal(full);
   end;
```

Condition variable != Semaphore

- Condition variables provide a mechanism to wait for events (a "rendezvous point")
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - Wait – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - Signal – wakeup one waiting thread
  - Broadcast – wakeup all waiting threads
- Note: Condition variables are not boolean objects
  - "if (condition_variable) then" … does not make sense
  - "if (num_resources == 0) then wait(resources_available)" does

Monitors in Java

- Every object of a class that has a ***synchronized*** method has a monitor associated with it
  - Any such method is guaranteed by the Java Virtual Machine execution model to execute mutually exclusively from any other synchronized methods for that object
- Access to individual objects such as arrays can also be synchronized
  - also complete class definitions
- *One* condition variable per monitor
  - ***wait()*** releases a lock, i.e, enters holding area
  - ***notify()*** signals a process to be allowed to continue
  - ***notifyAll()*** allows all waiting processes to continue

Monitors in Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object's monitor
  - A thread enters an object's monitor by
    - » Executing a method declared "synchronized"
      - Can mix synchronized/unsynchronized methods in same class
    - » Executing the body of a "synchronized" statement
      - Supports finer-grained locking than an entire procedure
      - Identical to the Modula-2 "LOCK (m) DO" construct
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()

- example:

```
class DataBase {
    public synchronized void write ( . . . ) { . . . }
    public synchronized read ( . . . ) { . . . }
    public void getVersion() { . . . }
}
```

  - once a thread enters either of the *read* or *write* methods, JVM ensures that the other is not concurrently entered for the same object
  - *getVersion* could be entered by another thread since not synchronized
  - code could still access a database safely by locking the call rather than by using synchronized methods:

```
DataBase db = new DataBase();
  synchronized(db) { db.write( . . . ); }
```

Monitor as a Mini-OS

- The concept of a monitor is very similar to an operating system.
  - One can consider the initialization as those data that are initialized when the system is booted up, the private data and code as the internal data structures and functions of an operating system, and the monitor procedures as the **system call**s.
  - User programs are, of course, threads that make service requests.
  - Therefore, a monitor can be considered a mini-OS with limited services.