We also call the mechanism to overcome this risk as synchronization

Operating sys

Part V: Synchronization risk [1]

Data Inconsistency

By KONG LingBo (孔令波)

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

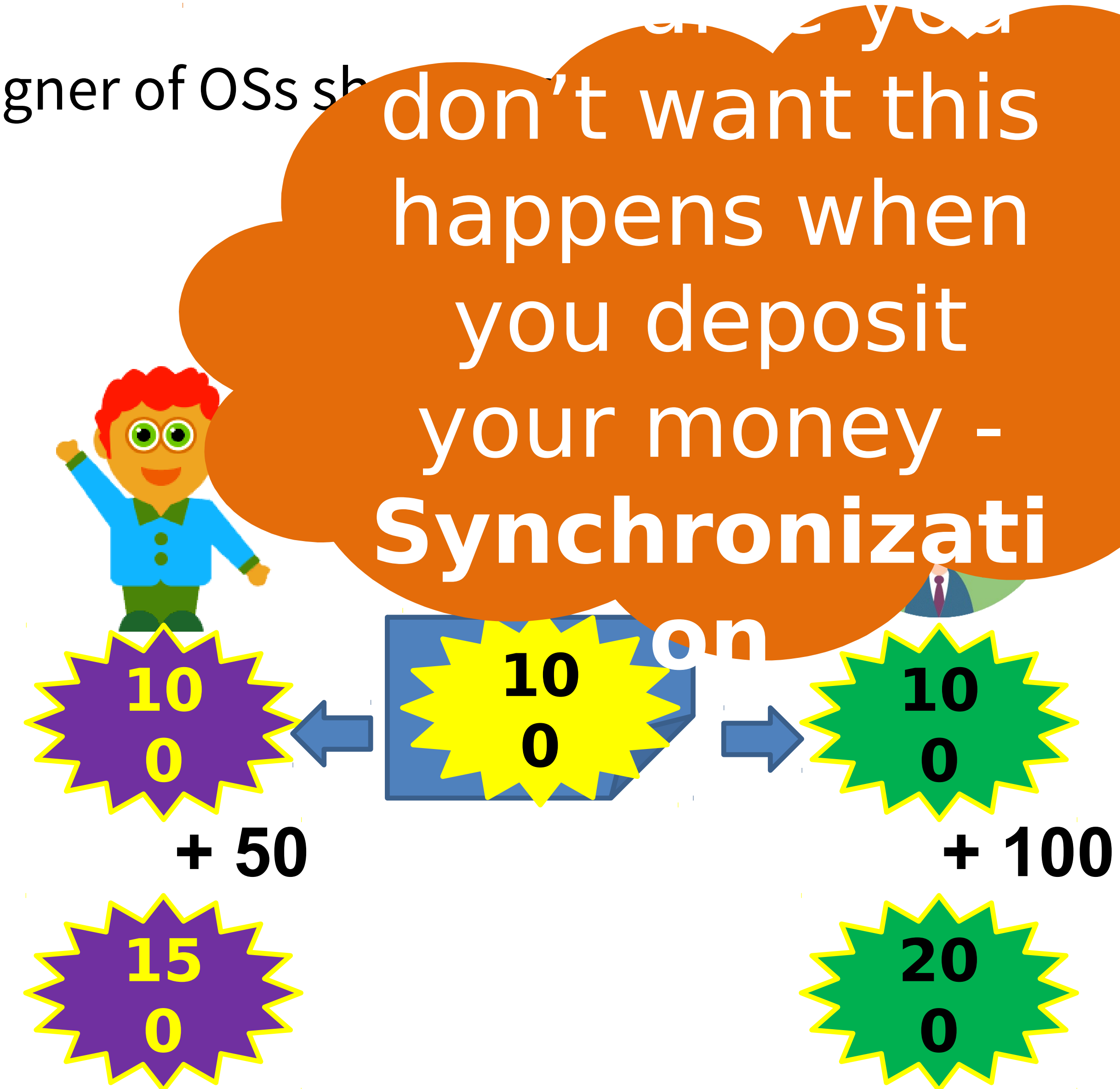
- Concurrent processes (or threads) often need to **share data** (maintained either in shared memory or files) an d **resources**
  - If there is no proper policy to assign resources among processes, it may result in that all the processes get bl ocked ☾ **Deadlock** [死锁]
- If there is **no controlled access** to shared data, e xecution of the processes on th... leave **Cooperation**
  - **The results will then depend ... data were modified ☾ Data In...**
    - i.e. the results are non-deterministic

It always implies "Share resources" at the same time

PPTs from others\flame.cs.dal.ca_~hawkey_3120\May21ProcessSynchronization.ppt

Some functions the designer of OSs sh

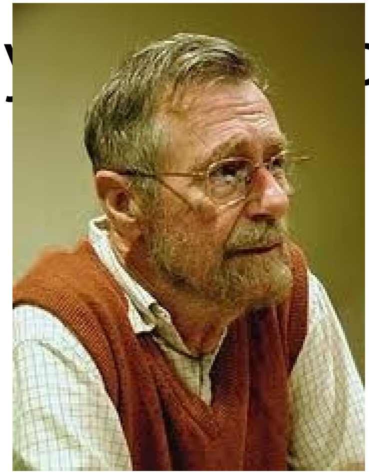
- Resource competition
  - **Data inconsistency** is another issue we should consider carefully when competing.

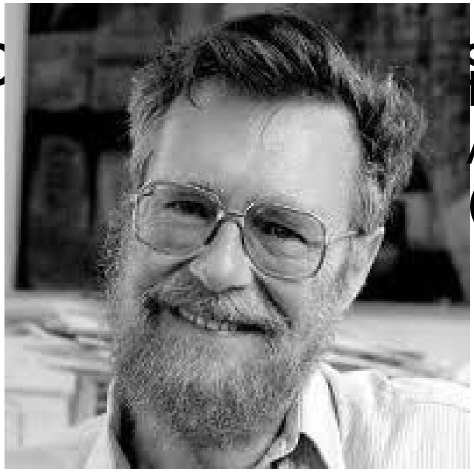


Summarized as **Mutual Exclusion** [互斥] problem

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

- 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**





stra in 19

Died at
August 6, 200
(aged 72)

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

For memorizing: **Edsger 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;
  - **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)
Load R1, balance

Add R1, amount

Store R1, balance

What is the final balance?

Time line

Thread 2:deposit(20)

Load R1, balance

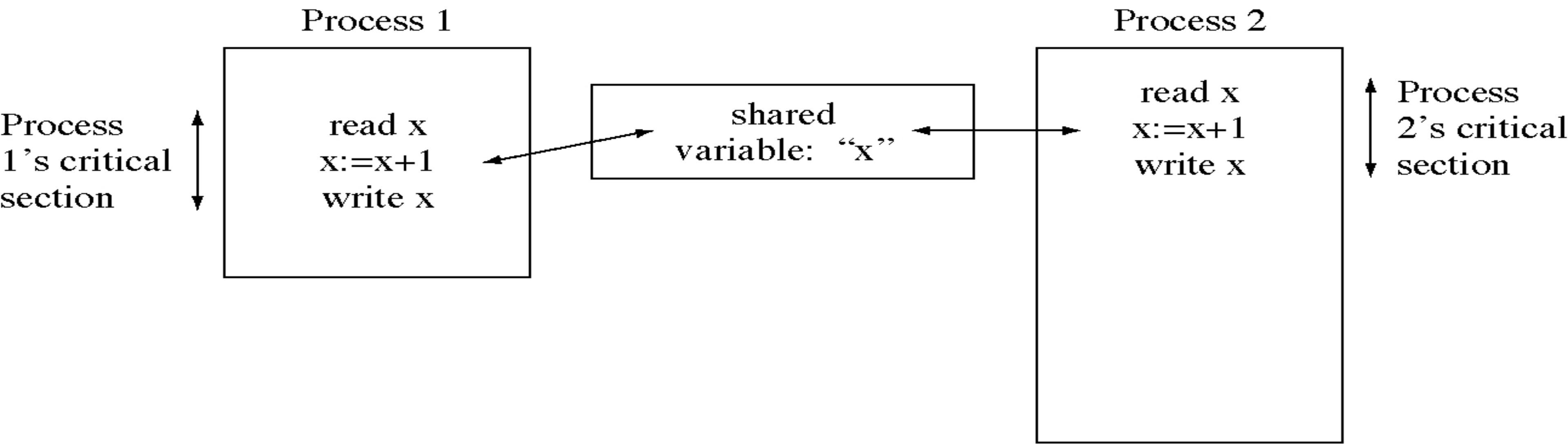
Add R1, amount

Store R1, 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



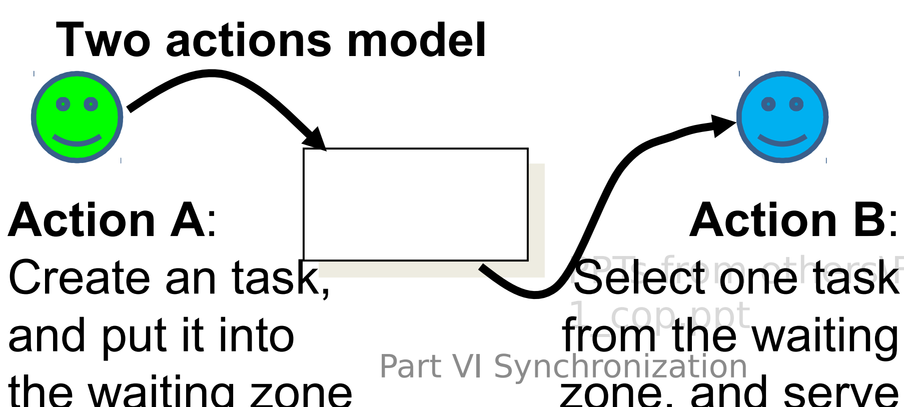
We investigate the P-C model first

CLASSIC SYNCHRONIZATION MODELS

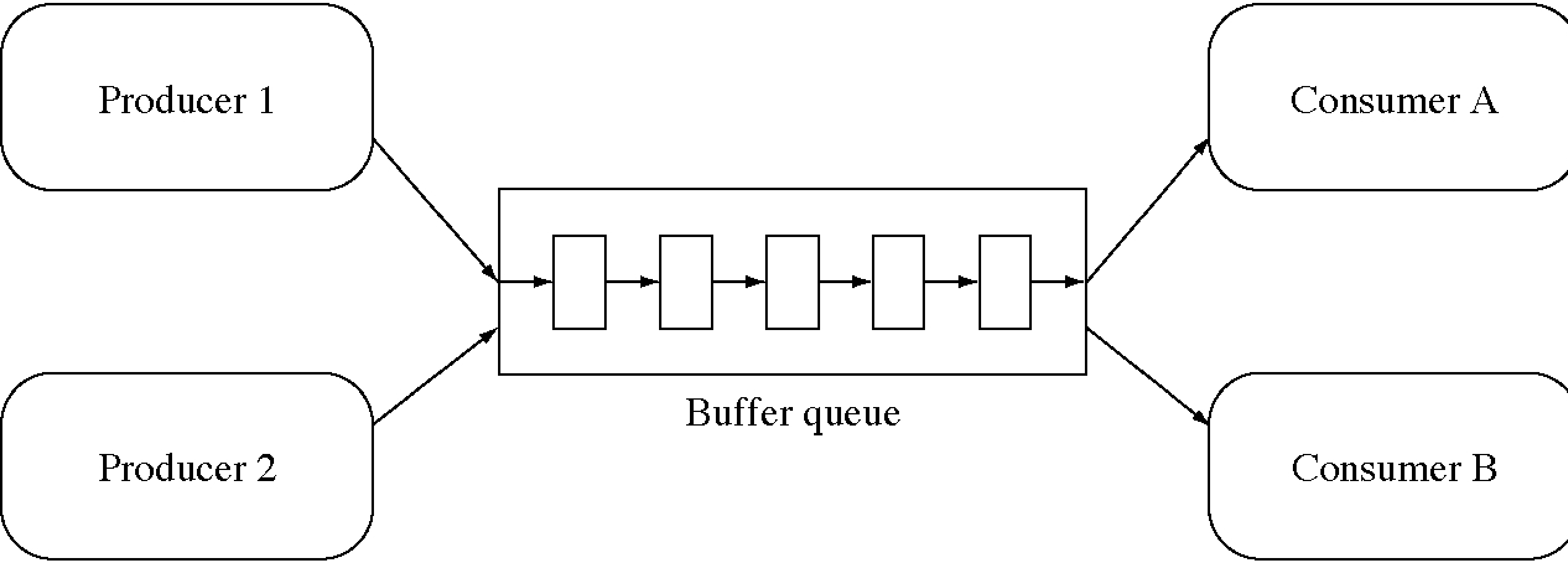
- Producer-Consumer model
- Readers-Writers Problem
- The Barbershop Problem
- Dining philosopher problem

Producer/Consumer (P/C) Problem

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



Multiple Producers and Consumers



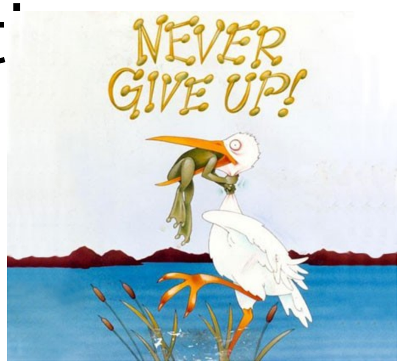
The key of synchronization

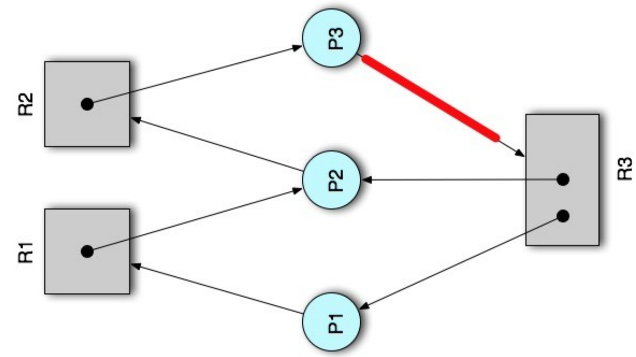
TO CONTROL THE EXECUTION OF CRITICAL SECTIONS AMONG CONCURRENT PROCESSE S/THREADS

Ensuring <u>**Mutual Exclusio**</u>

But we also should avoid For **EFFICIENCY**

- **Deadlock**
  - None of the involved processes could move fur
    t





- **Starvation**
  - Some process is never executed!
  - Process/ th nitely

Rules for robust synchronization

- Of course, Mutual exclusion should be guaranteed (**consistency**) [互斥]
  - Only one 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 DERIVED FROM **LOCK MECHANISM**!

Solution ... blem using Locks

- T ... ution is:

do {

acquire **lo...**

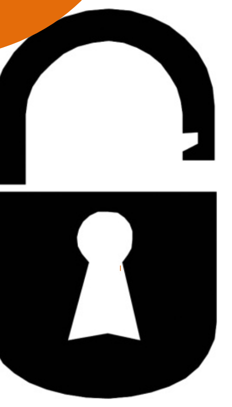
critical section

release **lock**

remainder section

} while (TRUE);

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



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).

The Critical-Section Problem: 2-Processes: Algorithm N (Pet[illegible] 1981)

> Interrupt could happen between any two near statements

- Process P_i

```
do {
    flag [i] = true;
    turn = j;
    while (flag [j] and turn = j) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

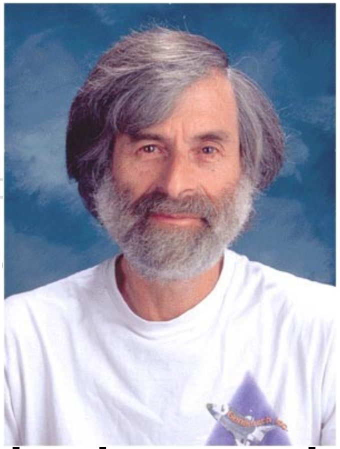
- Meets all three requirements; solves the critical-section problem for two processes.

Peterson’ s Algorithm: Proof of Correctn ess

- 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 a nd turn=1 (at same time) ℗ this is **impossible**

How about for N-processes?

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



- Critical section for n process
  - We have **Bakery algorithm**
  - It is also known as **Lamport's** bakery algorithm. Lamport?
  - Dr. **Leslie Lamport**
    - I know him because of his wo
    - You can check some informati my homepage.

You could learn this by yourself and finish a report.

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 impl
    waiting functio
    emaphores & Mo

In DBMS, the
SELF-SPIN LOCK
[自旋锁]
Is a kind of
BUSY WAITING
lock

Hardware solutions

- Many systems provide hardware support fo r critical section code.
- Uniprocessors – could **disable interrupts**:
  - Currently running code would execute without preemption.
  - Generally too inefficient on multiprocessor syst ems.
- 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 C S, nothing else can run
  - because preemption impossible
- On a multiprocessor: mutual exclusion is not achieved
  - **Interrupts are "per-CPU"** (interrupts are not di sabled on other processors).
- Generally not a practical solution for user programs, but cou ld 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 r

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 represents the essence of the corresponding machine instruction.

PPTs from others\From Ariel J. Frank\OS381\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 th e corresponding machine instruction.

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

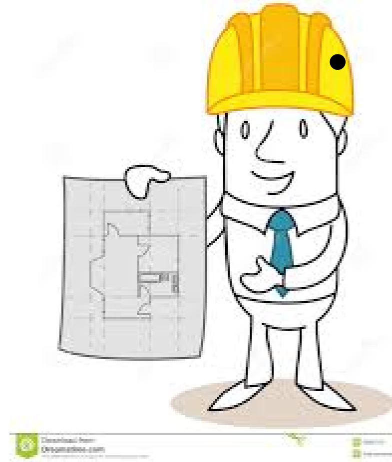
Machine Instructions for Mutual Exclusion

- 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>** cons
  - Starvation is possible
    tion and more than one proces
  - 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 ot n
Stallings\Chapter05.ppt

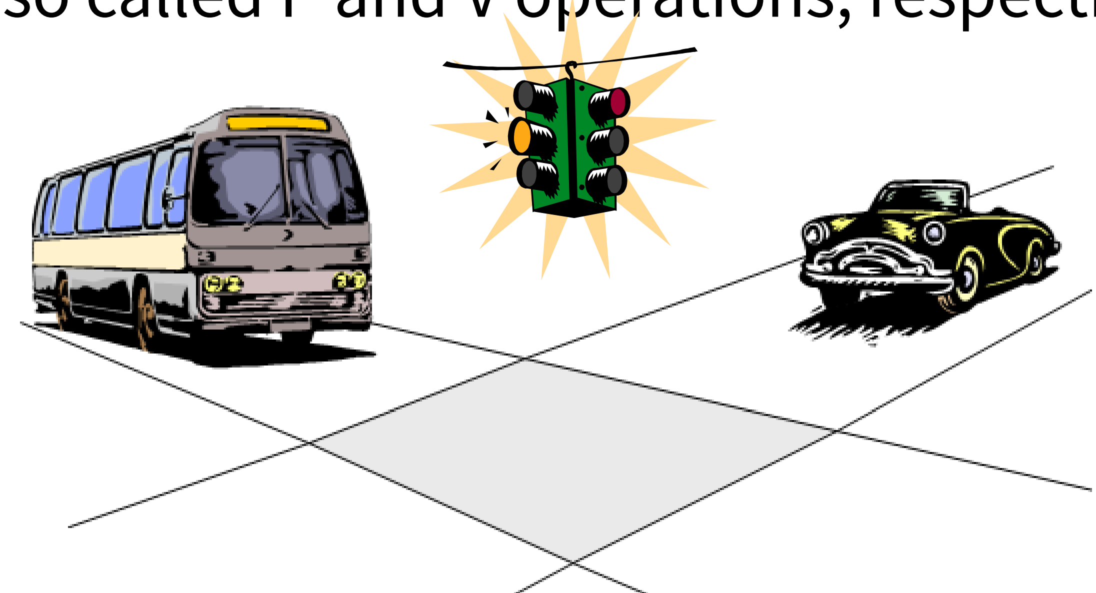
Lock + Manager + Waiting Room



- 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)

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 command s
    - 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

May 11, 1930 – August 6, 200

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

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

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

- Invented in the 1965
  - **Basis of all contemporary OS synchronization mec hanisms**
- In computer science, a semaphore is a protected variable or abstract data type that constitutes a c lassic method of controlling access by several pr ocesses to a common resource in a parallel progr amming environment.
  - A semaphore generally takes one of two forms: **binar y** and **counting**.
- Either semaphore type may be employed to prev ent a race condition.
  - On the other hand, a semaphore **is of no value in pre venting resource deadlock**, such as the dining philo sophers problem.

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 sectio
   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 proces ses (**Order Scheduling [S**

– Execute *B* in P_j only after *A*

– Use semaphore *flag* init

– Code:

P_i P_j

A ***wait(flag)***

signal(flag) ***B***

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

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

A() {              B() {
  write(x);              P(s1);
  V(s1);               read(x);
  P(s2);               write(y);
  read(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 P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

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, thread is blo... a resource becomes avai...
  - Increment the semaph... urning it to pool s...
    - If there are processes... ces, wake up one of th...

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 proc ess is no longer in the critical section

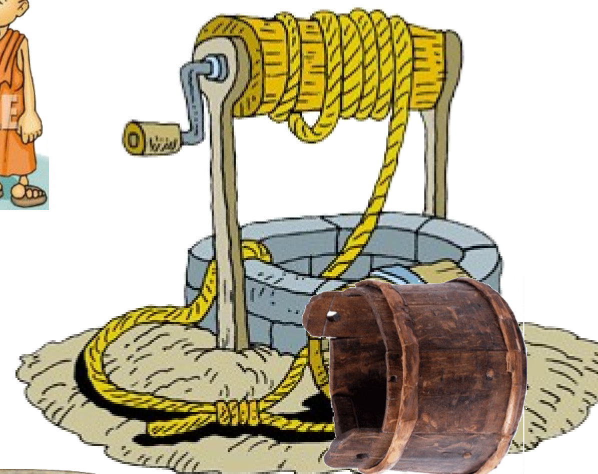
Implementation

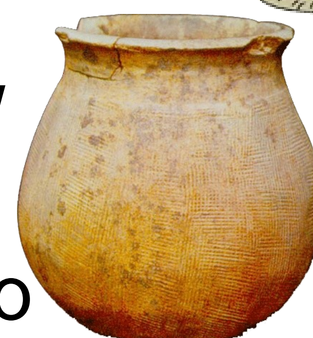
- BTW, the implementation just described is how Linux implements semaphores
- The up and down operations represent requ ire access to a critical section which is the semaphore variable
- Need hardware/OS support e.g.,
  - Signals,TSL
  - Signals allow for a "message" to be sent to proc esses

DETAILED EXAMPL
E FOR CS PROBLEM

A story: monks drink water

- Many monks
  - Some are **old**
  - Some are **young**
- One well
  - Only one bucket in well ever y time
- One Vat
  - Can contain 10 buckets of w ater
  - One bucket to put water into and fetch water from the vat
- Three buckets in total



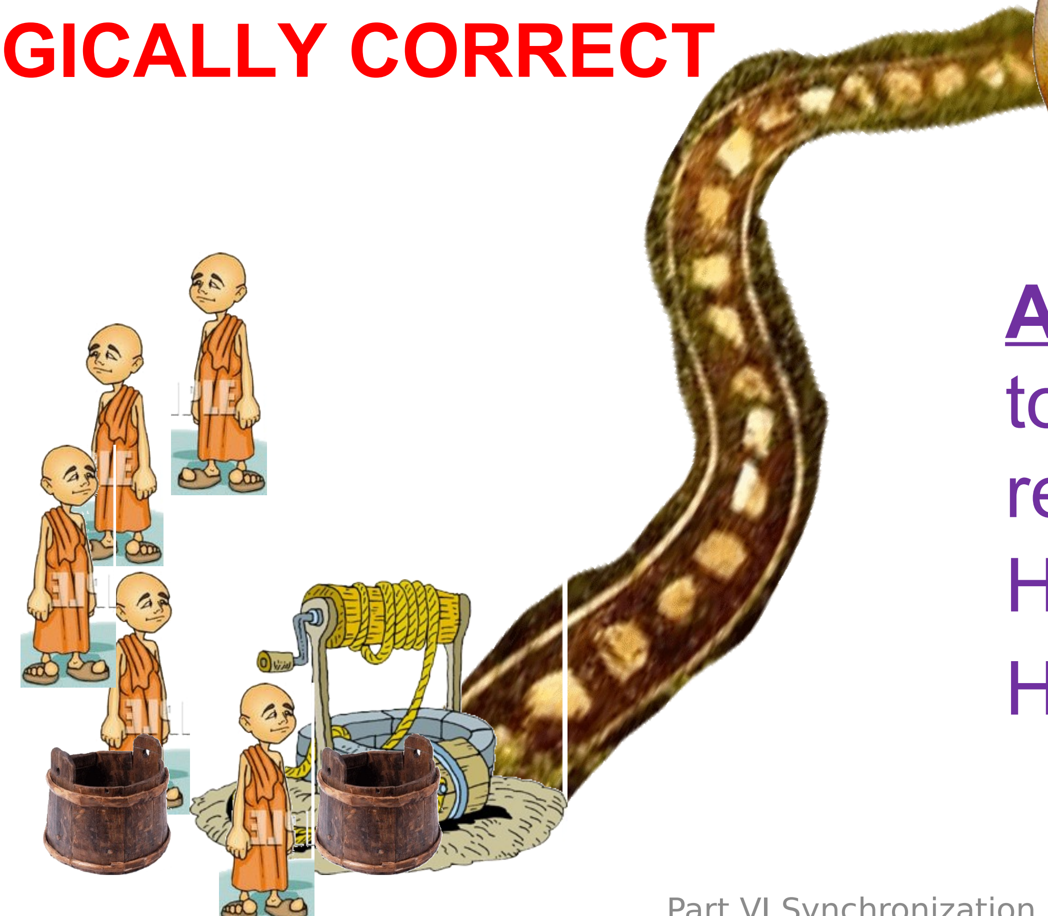


General rules to cope with CS problem using semaphores

1. Find the types of actors
   - To determine the **processes**
2. Recognize the shared resources between actors ☾ initial values of semaphores
3. Infer 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)

<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 a nd 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 sh ould get one bucket to pick water from the well or v

In our story (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) .

In our story (cont'　)

3. Infer the constraints based on the situations whe
n 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 t
he old: initially, the vat has 0 bucket of water)
       2. Compete a bucket (☾ **ME**: share buckets with othe
r monks)
       3. Try to fetch a bucket of water from the vat (**ME** of v
at:)
       4. Release the bucket
       5. Inform the young because the vat now could cont
ain one more bucket of water (**SCH**: for the youn

In our story (cont’)

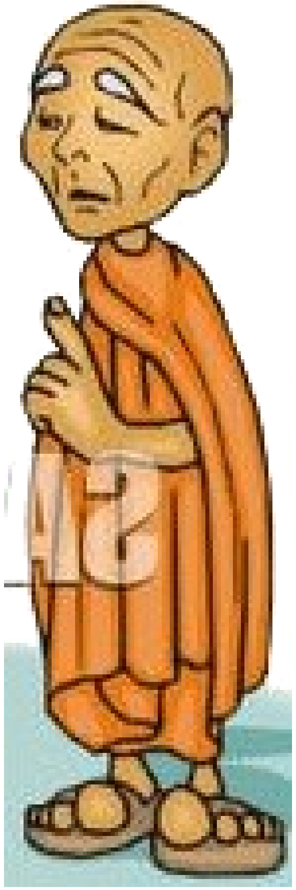
4. Use semaphores to finish those processes
   - Binary semaphores for **ME**: Well, vat
   - Counting semaphore: idleBuckets (=3) for **<u>ME</u>**, Vat CouldContain (= 10), WaterInVat (= 0) for **<u>SCH</u>**
     - For young:
       1. when VatCouldContain =0, the young should not compete th e buckets;
       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 t he 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:

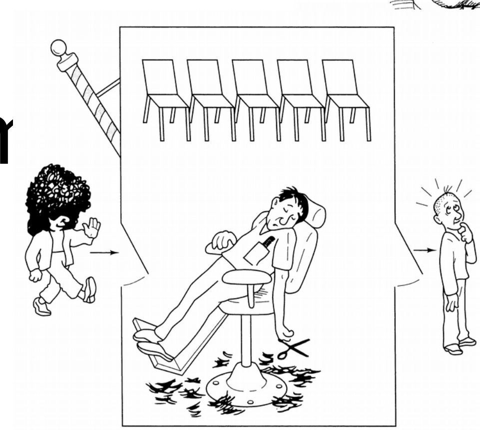
```
P(WasterInVat) // initial = 0
  P(idleBuckets); // initial = 3
    P(vat);
      // get a bucket and drink the water
    V(vat);
  V(idleBuckets);
V(VatCouldContain); // initial = 10
```



CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-b
  - （有限缓存问
- Readers-wr
  - （读者 - 著者问题）
- Dining-philosophers proble
  - （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.p

Be careful: Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitel y for an event that can be caused by only one of waiting p rocesses.
  - 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 whic h it is suspended.
- **Priority Inversion** – scheduling problem when lower-prio rity process holds a lock needed by higher-priority proces s

Programming Language solutions: **Monitors**

- 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 != Semaph ore

- 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*** meth
od has a monitor associated with it
  - Any such method is guaranteed by the Java Virtual Machin
e execution model to execute mutually exclusively from a
ny other synchronized methods for that object
- Access to individual objects such as arrays can also b
e 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 a n operating system.
  - One can consider the initialization as those data that are initialized when the system is booted u p, 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.