Classic Synchronization Problems and Monitors

Operating Systems and Multiprogramming

Lecturer: Prof. Eric Jul

29-Feb-2016

Present by: **Marjan Mansourvar**

Contents

Short recap on last lecture

- The Critical-Section Problem
- Solutions to the critical-section problems
- Requirements

Software solutions to the critical section problem

- Peterson's Solution
- Semaphores

Classic Synchronization Problems

- The Dining- Philosophers problems
- The Readers- Writers Problem
- The bounded-Buffer Problem

Contents

- **Monitors**
  - Concept and Invention
  - How to: Signal
  - Monitoring the Philosophers
  - Monitors in Java
- **Synchronization Examples**
  - Transactional Memory
- **THE multiprogramming system**

The Critical-Section Problem

Critical Section:

- A system consisting of n processes (p0,p1,…pn-1) , each process has a segment code called critical section

Critical section problem

- is to design a protocol that processes can use to cooperate

Do{

Entry section

Critical section

Exit section

Remainder section

}

General structure of a process *Pi*

The Critical-Section Problem

3 Requirements :

- **Mutual exclusion** (Protection of critical sections)
- If process Pi is executing in its critical section then no other process can be executing in their critical section .

- **Progress**
- The decision as to who can enter the critical section must not be postponed indefinitely.
- (when no process is in its critical section and several processes are requesting to enter their critical section, who should decide who will enter the critical section?)

- **Bounded waiting**
- Exists a bound or limit on the number of times that the process are allowed to enter their CSs

Peterson's Solutions

- Restricted to 2 processes
- Requires 2 processes to share 2 data items:

int turn;

Boolean flag [2];

Turn==i; means Pi is allowed to execute in its CS

Flag[i]==true**;** means pi is ready to enter its CS

- Mutual Exclusion is preserved (Since the value of turn can be either 0 or 1)
- The progress requirement is satisfied
- The bounded-waiting is met

P0: do {
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)do skip;

// critical section
...
// end of critical section

flag[0] = false;

Reminder section

P1: do {
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0) do skip;
{

// critical section
...
// end of critical section

flag[1] = false;

Reminder section

Semaphores (Dijkstra, 1965)

- A high-level method for processes to synchronize activities
- A semaphore is an integer variable
- is accessed only through 2 standard operation : **wait() and signal ()** (P or V)

```
Wait (s) {
while (s <= 0)
; // busy
s--;
}
```

```
Signal (s) {
S++;
}
```

Going to the second parts …….

Classic Problems in Synchronization

- The Dining Philosophers
- The Readers-Writers Problems
- The Bounded-Buffer Problem

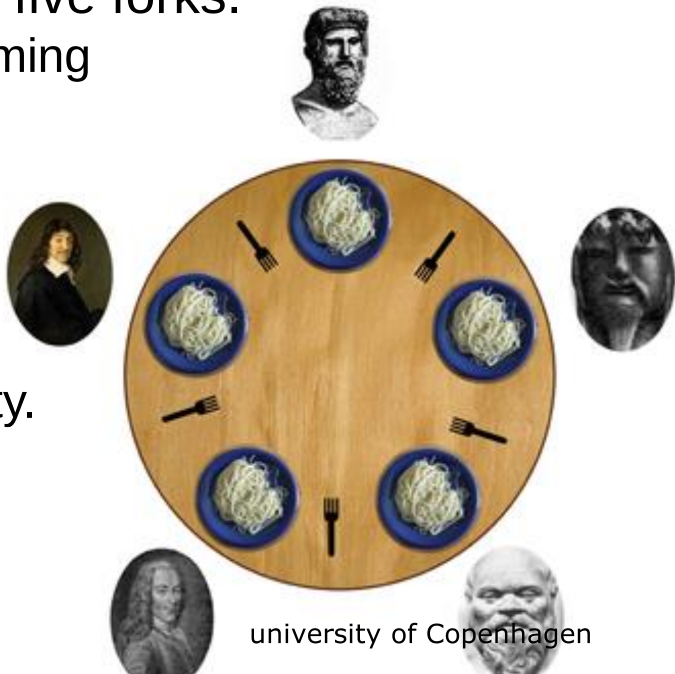
The Dining –Philosophers Problem

- It was originally formulated in 1965 by Edsger Dijkstra

A simple example showing the need to allocate several resources among several processes without deadlocks or starvation.

Problem Statement

- Five philosophers, sitting around a table, sharing five forks.
- The philosophers alternates between thinking, becoming
- hungry and eating.



- When hungry, they try to pick up a fork on his left and right.

- Then he goes back to solving the problems of humanity.

Feeding the Philosophers with Semaphores

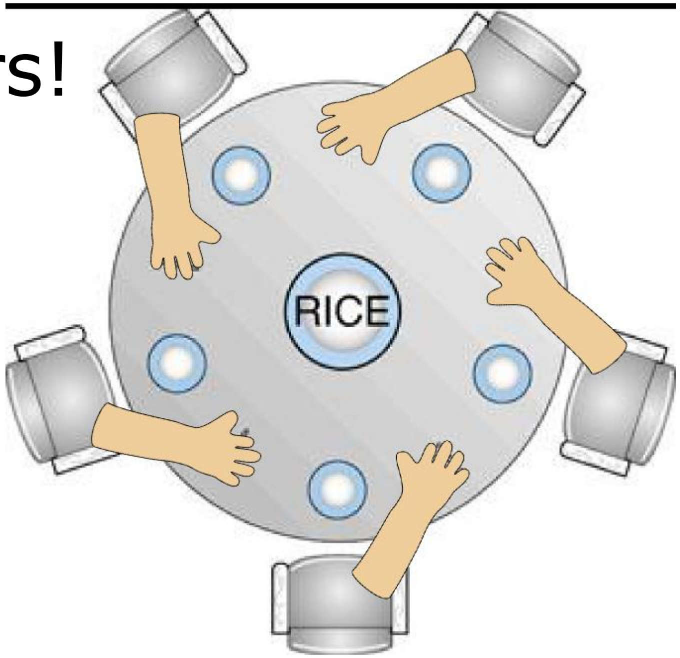
Represent each fork with a semaphore

- Try to grab a fork by execution a wait() operation on the semaphore
- Release her fork by exciting the signal ()

```
Semaphore fork [5];
do {
    wait (fork [i]);
  wait (fork [(i+1) %5]);
    …….
  / eat for awhile/
    …….
  signal (fork[i]);
signal (fork [(i+1) %5 ]);
  …….
/ think for awhile/
} while (true);
```

Feeding the Philosophers with Semaphores

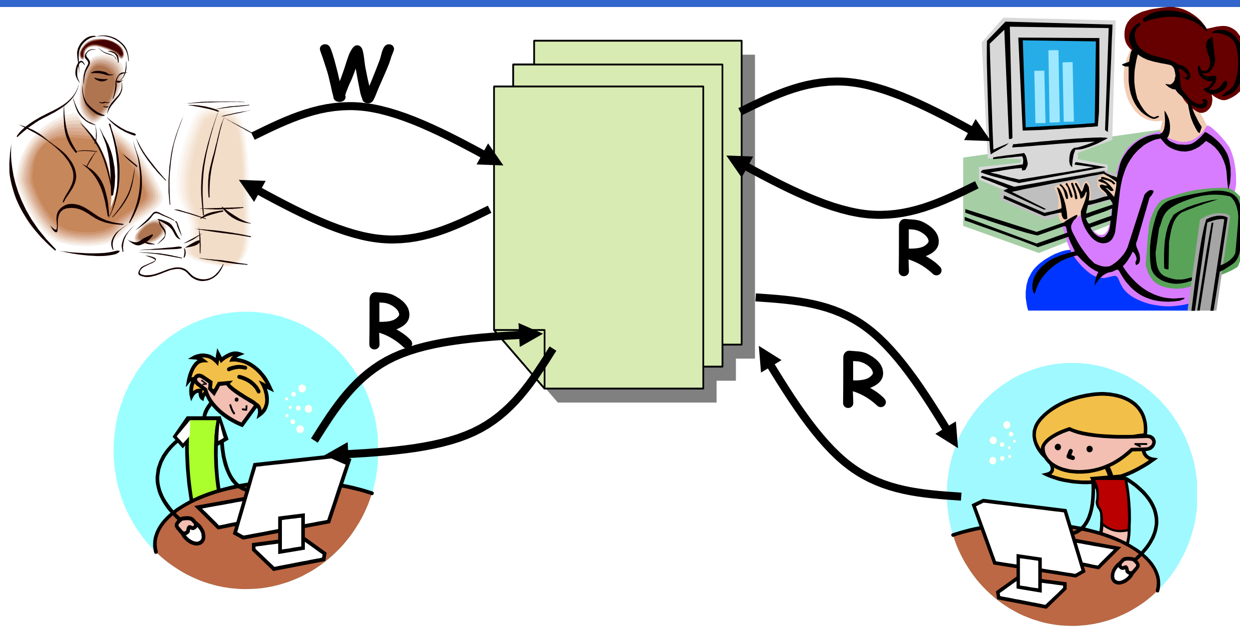
- Does this work?
- No. Risk of deadlock
- The Starving Philosophers!



Solutions to the Starving Philosophers

- Allow at most four philosophers to be siting simultaneously at the table .
- Allow a philosopher to pick up his fork only if both forks are available.
- Use an asymmetric solution(odd-number picks up first left and then right fork, while even-numbered picks up right fork and then left fork)

The Readers-Writers Problems



- A database is to be shared among several concurrent processes.
- Some of those want only to read (radars are allowed to read concurrently.)
- Some of them want to update. (writers must acquire exclusive access.
- Has several Variations , all involving priority

The Simple Semaphore Solution

- Count the number concurrent readers (counter protected by binary semaphore).
- First reader blocks future writers.
- Last reader unblocks writers.
- Ensure mutual exclusion amongst writers using a **write** semaphore.

Readers process share data structure

Initialization

Semaphore rw_mutex =1;

Semaphore mutex = 1;

int read_count = 0;

The Readers-Writers Problems

Reader

```
Do {
Wait (mutex);
Read_count++;
If (read_count == 1)
        Wait(rw_mutux);
Signal(mutex);
………
/*reading is performed*/
……….
Wait(mutex);
Read_count--;
If(read_count== o)
        Signal(rw_mutex);
Signal(mutex);
} while (true);
```

Initialization

```
Semaphore rw_mutex =1;
Semaphore  mutex = 1;
int read_count = 0;
```

Writer

```
do {
Wait (rw_mutex);
…….
/* writing is performed */
…….
Signal (rw_mutex)
} while ( true) ;
```

The Bounded-Buffer Problem

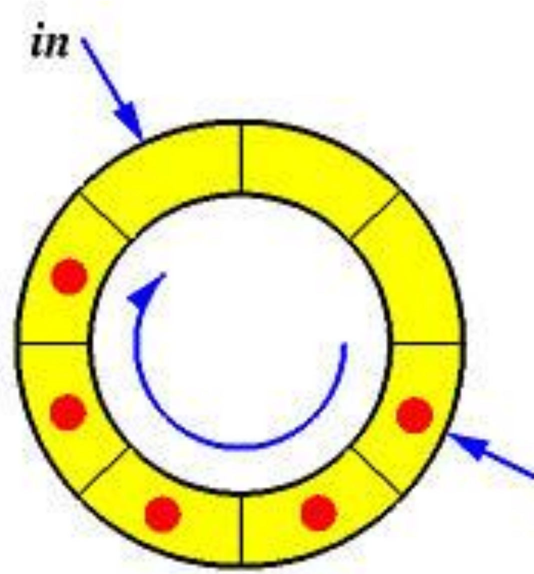
- The producer–consumer problem also known as the bounded-buffer problem

- Suppose a circular buffer with two pointers ***in*** and ***out*** to indicate the next available position for depositing data and the position that contains the next data to be retrieved.

- Bounded buffer: size 'N'
  (Access entry 0… N-1, then "wrap around" to 0 again)

- Producer process writes data to buffer
  (Must not write more than 'N' items more than consumer "ate")

- Consumer process reads data from buffer
  (Should not try to consume if there is no data)



Solving with semaphores

int n
semaphore mutex = 1; /* for mutual exclusion*/
semaphore empty = N; /* number empty buffer entries */
semaphore full = 0; /* number full buffer entries */

Producer

```
do {
……..
/*produce an item in next produced*/
 wait(empty);
 wait(mutex);
…….
/*add next produced to the buffer*/
……….
signal(mutex);
 signal(full);
} while (true);
```

Consumer

```
do {
wait(full);
wait(mutex);
………
//remove an item from buffer*/
……
signal(mutex);
signal(empty);
………
/
* consume the item in next-consumed*/
……………….
} while (true);
```

What's wrong with Semaphores?

- They are essentially shared global variables.
- Access to semaphores can come from anywhere in a program.
- They serve multiple purposes (mutual exclusion , scheduling constraint, …)
- There is no control or guarantee of proper usage.

- **Solution: use a higher level primitive called monitors**

Hoare Monitors

A synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) on certain conditions.

Invented by C. A. R. Hoare (aside: read up on CSP; it's awesome) in 1974. First implemented by the late Danish-American Per Brinch Hansen in 1993 as part of his Concurrent Pascal language.

Basically a mutex and a number of condition variables.

In terms of object-oriented programming (where we most often see monitors), it is a thread-safe class, object or module that uses implicit mutual exclusion to allow access to a method or variable by concurrent threads.

Hoare Monitors

- ❑ Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
- If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks

- ❑ The only operations can be invoked on a condition are wait () and signal ().

❑ X.wait ();

- The process invoking it is suspended until another process invoke

❑ X. signal () ;

- This process resume exactly one suspended process.

Monitors: Key Features

Higher level construct than semaphores**.**

A **package** of grouped procedures, variables and data.

Processes can only call procedures within a monitor

Can be built into programming languages.

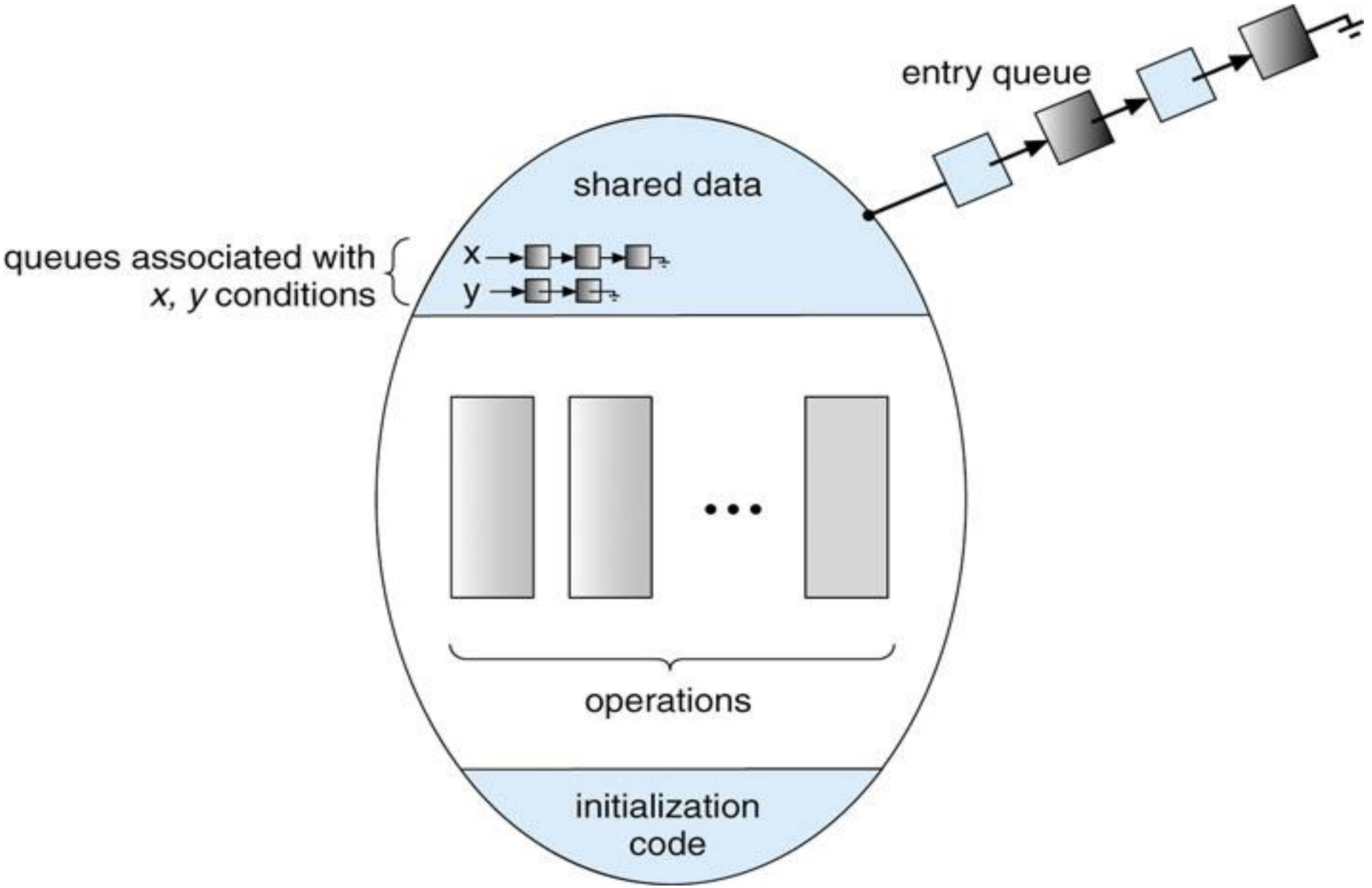
Synchronization is enforced by the compiler.

Only **one process** allowed inside the monitor at any time.

Wait and signal on condition variables.

Perfect

View of a Monitor



Are you ready to continue ????

Monitoring the Dining-Philisophers

- Monitor concepts by presenting a deadlock-free solution

- Imposes the restriction that a philosopher may pick up the forks only if both of them are available.

- Data structure :
- Enum{THINKING, HUNGRY, EATING} state [5]

- Philosopher I can set the variable state[i]= EATING only if 2 neighbors are not eating: (state[(i+4)%5] != EATING) and (state[(i+1)%5] != EATING).

Monitoring the Dining-Philisophers

```
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(i));
17 test(RIGHT(i));
}

void test(int i) {
if ((state[LEFT(i)] != EATING) &&
(state[i] == HUNGRY) &&
(state[RIGHT(i)] != EATING)) {
state[i] = EATING;
}
```

```
Initialization_code ()
{ for (int i = 0; i < 5; i++)
state[i] = THINKING; }
LEFT(i) = ((i+1) % 5)
define RIGHT(i)= ((i+4) % 5)
```

Resuming Processes within a Monitor

If several processes are suspended on condition X, and X.signal() operation is executed, how we determine which process should be resumed?

- Use a first-come, first-served (FCFS) ordering
 (The process has been waiting the longest is resumed first .

- **Conditional-wait** construct can be used.

x.wait(c);

c is an integer expression called a **priority number**

Monitors in Java

An example of a language that includes monitor support directly:

- Each object has an associated lock.
- Declare methods synchronized to enable mutual exclusion.
- Condition variables replaced by wait() and notify() methods.
- Java API supports semaphores, condition variable, and mutex lock in the **Java.util.concurrent** package .

```
Public class SimpleClass {
  ……..
Public synchronized void
safeMethod (){
….
/* Implamnetion of safeMethod() */
……
}
}
```

Transactional Memory

The main concept of **transaction Memory** originated in database theory

Provide a strategy for process synchronization .

A **transaction Memory** is a sequence of memory read-write operations that are atomic.

If all operations in a transaction are completed the memory transaction is committed.
Otherwise the operations must be rolled-back.

The benefits of transaction Memory can be obtained through features added to a programing language.

An example :

```
Void update()
{
Acquire ();
/* modify share data*/
Release();
}
```

Problems of deadlock

We add the construct atomi{s} (ensures that the operations in S execute as a transaction.

```
Void update ()
{
atomic {
/* modify share data*/
}
}
```

Advantages:

-The transactional memory system- not the developer- is responsible for guaranteeing

- deadlock is not possible
- Identify concurrent read access to the share variable

Be Patient ……Peak is near

THE multiprogramming system

THE multiprogramming system was a computer **operating system** designed by a team led by **Edsger W. Dijkstra**, described in monographs published in 1968.

The THE system was primarily a batch system that supported multitasking;

The THE system apparently introduced the first forms of software-based memory segmentation

freeing programmers from being forced to use actual physical locations on the drum memory.

THE multiprogramming system

using a modified ALGOL compiler (the only programming language supported by Dijkstra's system) to "automatically generate calls to system routines.

The code of the system was written in assembly language for the Dutch Electrologica X8 computer.

This system, THE, sported many important features found in modern operating systems such as multiprocessing, virtual memory, resource sharing, and standardized buffered I/O.

The design of the THE multiprogramming system is significant for its use of a layered structure, in which "higher" layers only depend on "lower" layers:

Design

Layer 0 was responsible for the multiprogramming aspects of the operating system. It decided which process was allocated to the CPU, and accounted for processes that were blocked on semaphores.

Layer 1 was concerned with allocating memory to processes.

Layer 2 dealt with communication between the operating system and the console.

Layer 3 managed all I/O between the devices attached to the computer. This included buffering information from the various devices.

Layer 4 consisted of user programs.

Layer 5 was the user (as Dijkstra notes, "not implemented by us").

Summary

- A look on classical synchronization problems
  - The Dining- Philosophers problems
  - The Readers- Writers Problem
  - The bounded-Buffer Problem
- The monitor concepts
- THE multiprogramming system

Thank You !

Computer science

That's all for today