Chapter 7

Concurrent Programming (2)

References :

[1] รังสิพรรณ มฤคทัต, กระบวนทัศน์ในการเขียนโปรแกรม (บทที่ 6)

[2] Tucker & Noonan, Programming Languages: Principles and Paradigms (Chapter 17)

[3] Sebesta, Concepts of Programming Languages (Chapter 13)

[4] Carver & Tai, Modern Multithreading (Chapters 3-4)

[5] Oracle, Java Documentation

Chapter Objectives

At the end of this chapter, you should be able to:

- Explain competition synchronization (semaphore, monitor)
- Explain cooperation synchronization (wait/notify, barrier, join, exchanger)
- Hand trace Java programs with threads & synchronization
- Choose appropriate synchronization methods for given problems
- Write Java programs with threads & synchronization

Interaction between Threads

- Communication
  - In Java, threads can send values/objects to each other via method call and argument passing
    - E.g. T1.setValue(v) ➔ sending v to T1
- Synchronization
  - Arrangement of multiple thread execution
  - Competition synchronization (compete for resource)
    - Threads try to update the same variable
    - Threads try to use System.out at the same time
  - Cooperation synchronization
    - Threads wait for each other upon some conditions

Issues with Concurrency

- Required properties of concurrent program
  - Safety ➔ yield the same effect as sequential program
  - Liveness ➔ able to continue, eventually leading to completion
- Problems
  - Race condition, deadlock, livelock, starvation
- Solutions
  - Critical section / mutual exclusion handling
  - Semaphore, Monitor

Race Condition

- Inconsistent update, whose effect depends on the order in which concurrent threads execute

Thread A	Thread B	
pv = **sh**;	pv = **sh**;	**sh** = shared variable = 10
pv++;	pv++;	pv = private variable
sh = pv;	**sh** = pv;	

- If one thread completes 3 lines before the other starts
  - A pv = sh = 10; increase pv to 11; sh = pv = **11**
  - B pv = sh = 11; increase pv to 12; sh = pv = **12**
- If both threads have equal speed
  - A pv = sh = 10; increase pv to 11; sh = pv = **11**
  - B pv = sh = 10; increase pv to 11; sh = pv = **11**

R. Marukatat

- Race condition occurs as a result of non-deterministic execution
  - Deterministic execution : the same program executing the same data always yields the same execution order
  - Non-deterministic execution : execution order may differ in each run, depending on context switching between threads, processor speed, etc.

Example : race condition

```
class MyThread extends Thread  {
      private  Buffer  buffer;
      …
      public void run( )    {  buffer.add( );  }
}

class Buffer  {
      private int sum = 0;
      public void add( )  {
          for (int i=0; i < 5; i++)  {
              sum++;
              try { Thread.sleep(100); }  catch(InterruptedException e) { }
          }
      }
};
```

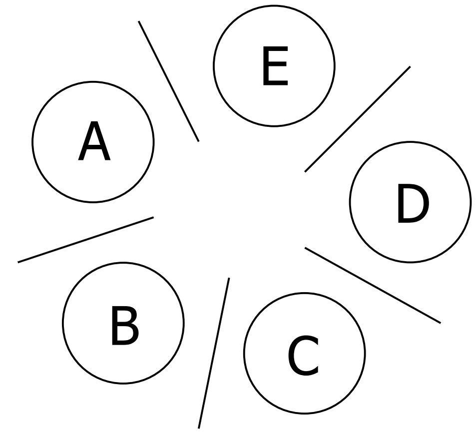
If 3 threads access the same buffer

T1	5
T2	10
T3	15

in any order

R. Marukatat

Deadlock



- The Dining Philosophers
  - To eat : use both chopsticks
  - To think : put down both chopsticks

- Deadlock occurs because all threads are waiting but none is able to proceed
  - for (;;) {

    pick left chopstick

    pick right chopstick

    eat

    release both chopsticks

    think

    }

Conditions for Deadlock

- All of these must be true for a deadlock to exist
  - Mutual exclusion : threads require exclusive right to resources
  - Wait and hold : threads hold some resources while waiting for others – since resources are acquired in pieces, not as a whole bunch
  - No preemption : resources cannot be released just because waiting threads need them
  - Circular wait : a circular chain of threads exists – each holds resource required by the next thread in the chain

Livelock / Starvation

- Livelock
  - No deadlock but no thread makes any progress
  - If left chopstick is picked but right chopstick is unavailable, one must release left chopstick
  - Livelock : everybody keeps picking & releasing a chopstick but nobody is able to eat
- Starvation
  - Occur when threads have different priorities / speeds
  - Thread waits for something from another thread, but what it waits for never arrives
  - A always gets both chopsticks, B never has a chance to eat

R. Marukatat

Critical Section

- A section of code that must be treated as an atomic event. For example, thread’s code that
  - Compete for shared resource
  - Read / write shared variable
- Mutual exclusion property
  - Critical sections are not overlapped → if thread A is in its critical section, the others cannot be in theirs
- Mutual exclusion handling
  - Semaphore : low-level mechanism
  - Monitor : high-level mechanism

Semaphore

- Integer variable (**S**), with thread queuing mechanism and 2 atomic test-and-set operations
  - **Acquire(S)**

    if (S > 0) S--;

    else wait until S > 0; // waiting threads may be enqueued
  - **Release(S)**

    if (threads are waiting for positive S) wake up 1 thread;

    else S++;

Thread A	Thread B
Acquire(S)	Acquire(S)
Critical Section	Critical Section
Release(S)	Release(S)

R. Marukatat

Synchronization steps

1. Semaphore S = 1				
2. Thread A	acquire(S) →	decrease S to 0		
3. Thread A	critical section			
4. Thread B	acquire(S) →	S = 0	→	wait
5. Thread A	release(S) →	B is waiting	→	wake up B
6. Thread B	critical section			
7. Thread B	release(S) →	none is waiting	→	increase S to 1
8. Thread C	acquire(S) →	decrease S to 0		
9. Thread C	critical section			
10. Thread C	release(S) →	none is waiting	→	increase S to 1

R. Marukatat

- Binary semaphore
- Counting semaphore
- Limitation
  - Semaphore is a shared variable & can be updated by one thread at a time ➔ managing semaphore itself requires proper synchronization
  - Acquire(S)/Release(S) : thread may be blocked & nobody unblocks it
  - Language support : require atomic test-and-set operation
  - Too low-level mechanism for software development

Class java.util.concurrent.Semaphore

- public **Semaphore** (int permits)
- public **Semaphore** (int permits, boolean fair)
  - Permits = #of permitted threads
  - If fair = true → FIFO for waiting threads
- public void **aquire** ( ) throws InterruptedException
  - Threads waiting to pass the semaphore are BLOCKED
- public void **release** ( )

Example : using semaphore

```
import java.util.concurrent.*;
class MyThread extends Thread  { … }              // from slide 7
class Buffer extends Semaphore  {
    private int sum = 0;
    private int N = 1;
    public Buffer( )       { super(N, true); }
    public void add( )  {
        try { acquire( ); }  catch(InterruptedException e) { }
        for (int i=0; i < 5; i++)  {
            sum++;
            try { Thread.sleep(100); }  catch(InterruptedException e) { }
        }
        release( );
    }
};
```

Semaphore's construtor true = FIFO for waiting threads

- *Succeed if semaphore's value > 0*
- *N threads are permitted to pass at a time*

Monitor

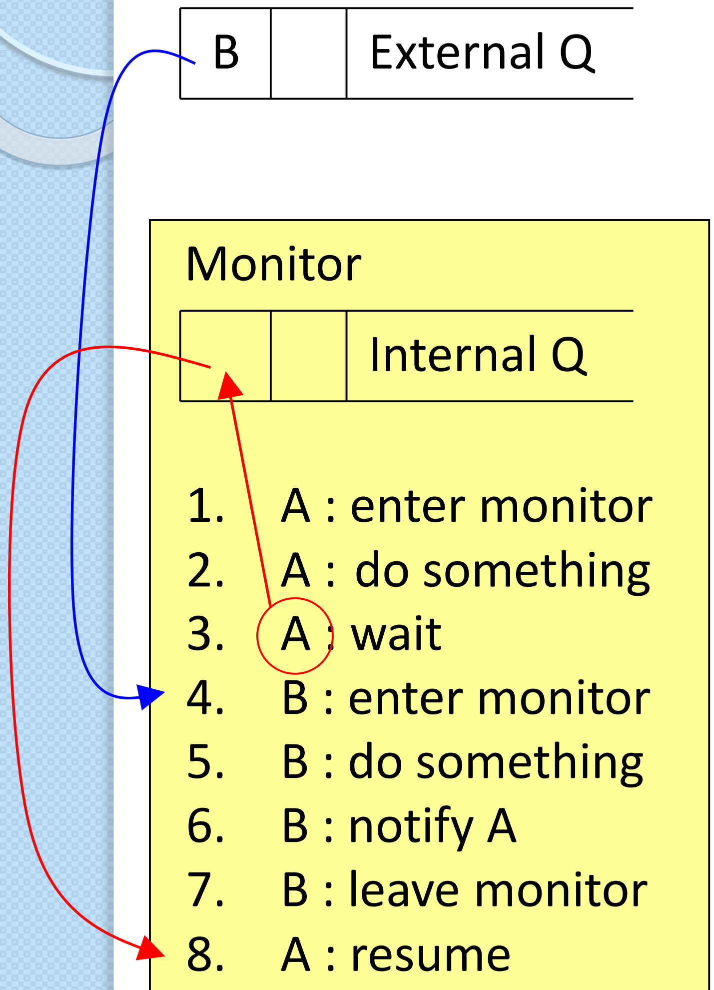
- A data structure that encapsulates shared data and mutually exclusive methods
  - Only 1 thread is allowed in a monitor ➔ synchronization is managed by runtime environment
  - Other threads wait in a queue, until the monitor is unoccupied
- Wait/notify
  - Synchronize threads inside & outside the monitor

Monitor (data structure)

Shared variable 1
Shared variable 2
…
Mutually exclusive method 1
Mutually exclusive method 2
…

R. Marukatat

Monitoring Mechanism



- External Queue : threads wait to enter the monitor
- Internal Queue : thread working inside monitor waits for some conditions
  - New thread (from external Q) enters unoccupied monitor
  - Upon notification, thread in internal Q re-enters once monitor is unoccupied
- Monitor : competition synch
- Wait/Notify : cooperation synch

Monitor in Java

- Class with at least 1 synchronized method acts as a monitor (built-in mechanism is added & handled by JVM)
- Object Monitor : synchronized method is non-static

```
class Buffer {
        ... // class variables ...
        synchronized public void put (...)          { ... }
        synchronized public int get (...)           { ... }
        ... // other non-synchronized methods
}
```

- Class Monitor : synchronized method is static

```
synchronized public static void sput (...)          { ... }
synchronized public static void sget (...)          { ... }
```

R. Marukatat

Object Monitor

- Every Java object has 1 lock
- Compiler adds object.lock + object.unlock at the beginning & the end of each synchronized method
  - Only 1 synchronized method can be executed at a time
  - Threads calling other synchronized methods (in the same object) are BLOCKED & wait in external queue

object buffer_1	**object buffer_2**
object_lock	object_lock
├ synch put ( )	├ synch put ( )
└ synch get ( )	└ synch get ( )

Thread A : buffer_1.put( )

Thread B : buffer_2.put( )

Thread C : buffer_1.get( )

Which ones are blocked ?

Example : using monitor

```
class MyThread extends Thread  { … }                // from slide 7
class Buffer  {
        private int sum = 0;

        // this method is automatically put in an object monitor
        synchronized public void add()
        {
           // object.lock is automatically added by Java
           for (int i=0; i < 5; i++)  {
                sum++;
                try { Thread.sleep(100); }
                     catch(InterruptedException e) { }
           }
           // object.unlock is automatically added by Java
        }
};
```

R. Marukatat

Class Monitor

- Every Java class also has 1 lock
- Compiler adds class.lock + class.unlock at the beginning & the end of each synchronized **static** method

class Buffer

class_lock

- synch static sput ( )
- synch static sget ( )

- Only 1 synchronized **static** method can be executed at a time

object buffer_1

object_lock

- synch put ( )
- synch get ( )

object buffer_2

object_lock

- synch put ( )
- synch get ( )

Thread A : buffer_1.sput( )

Thread B : buffer_2.sget( )

Thread C : buffer_1.get( )

Which ones are blocked ?

Example : object *vs.* class monitors

```
class Buffer
{
        synchronized public void put()
        {
           // object.lock is automatically added by Java
           …
           // object.unlock is automatically added by Java
        }

        synchronized public static void sput()
        {
           // class.lock is automatically added by Java
           …
           // class.unlock is automatically added by Java
        }
};
```

R. Marukatat

Synchronized Block

- Block of code (not whole method) can also be put in object monitor

```
Buffer b = new Buffer( );

public void f( ) {
    // non-critical code
    synchronized (b) {
        // critical code
    }
}
public void g( ) {
    synchronized (b) {
        // critical code
    }
    // non-critical code
}
```

object b

object_lock
- synch put ( )
- synch get ( )
- synch f's block
- synch g's block

- Thread A calls f( )
- Thread B calls g( )
- Thread C calls b.put( )

All of them compete to enter the same object monitor

Handling Wait / Notify

- Thread A **waits** in internal queue
- Thread B **notifies** thread A

- Issues
  - Does thread B have to leave the monitor immediately ?
  - When can thread A leave internal Q & enter the monitor ?
  - Is the condition for which A is waiting still true when it re-enters ?
    - Can be solved by waiting inside a loop
      while (!condition) { A waits }

  - Some languages allow multiple internal queues
    Hence, more issues about queue management

Strategy 1 : signal & exit

- When B notifies A → B leaves monitor & A re-enters monitor immediately ∴ notify is B's last command inside monitor
- Work with single internal queue only

Strategy 2 : signal & continue

- When B notifies A → B continues its execution inside monitor
- A leaves internal queue & enters external queue (A memorizes its resumption point). It has to compete with other threads to re-enter monitor & resume its execution

Strategy 3 : signal & wait

- Swap role after notification → B enters internal queue while A gets out of the queue to use monitor
- Difficult to implement & control

Wait / Notify in Java

- Java uses signal & continue strategy, with only 1 internal queue inside each monitor
- Class Object
  - public final void **wait** ( ) throws InterruptedException
  - public final void **wait** (long) throws InterruptedException
  - public final void **wait** (long, int) throws InterruptedException
- Waiting thread gets out of internal queue when
  - It is notified
  - Waiting time is over
  - It is interrupted while waiting

R. Marukatat

- Notify threads waiting in internal queue
  - These methods are member of class Object
  - public final void **notify** ( ) // OS chooses which thread // to be notified
  - public final void **notifyAll** ( ) // notify all threads

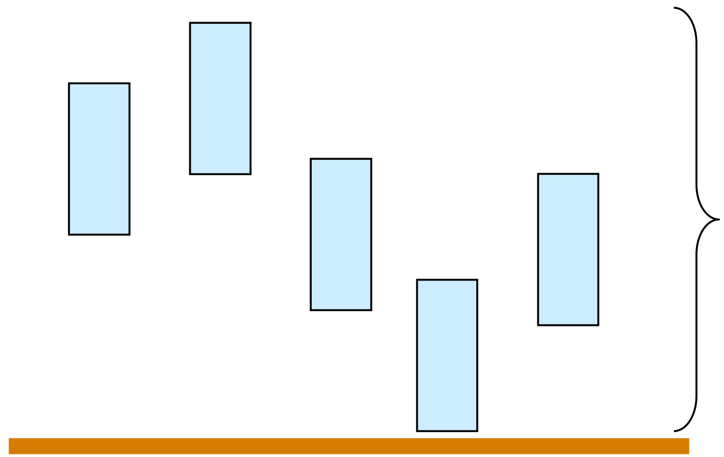
- Wait/notify must be used inside monitor
- Otherwise, IllegalMonitorStateException (runtime exception) will be thrown

share = -10

Thread A	Thread B	Thread C
local_w = 0	local_w = 10	local_w = -100
local_n = 10	local_n = 100	local_n = 0
0	10	-100
while (share < local_w)	while (share < local_w)	while (share < local_w)
{	{	{
wait until share >= 0	*wait until share >= 10*	*bypass this loop*
try { **wait( );** **}**	**try {** **wait( );** **}**	**try {** **wait( );** **}**
catch (…) { }	catch (…) { }	catch (…) { }
}	}	}
… update share to 15	… update share	… update share to 5
10	100	0
if (share >= local_n)	if (share >= local_n)	if (share >= local_n)
notifyAll( );	**notifyAll( );**	**notifyAll( );**
release B		*release A and B*

Barrier

- Synchronization between multiple threads
- Everybody waits until everybody else reaches certain point



When a thread reaches the barrier, it is blocked untill all the others also reach this point

Barrier

- Can be implemented by using wait / notifyAll
- Or use java.util.concurrent.CyclicBarrier

Class java.util.concurrent.CyclicBarrier

- public **CyclicBarrier** (int parties)
- public int **await** ( ) throws InterruptedException, BrokenBarrierException
  - Return #of threads still to arrive (so return 0 if it is the last thread to arrive)

```
try {  barrier.await();  }
catch(InterruptedException e)       { ... by interrupted thread ... }
catch(BrokenBarrierException e)   { ... by other threads at barrier ... }
```

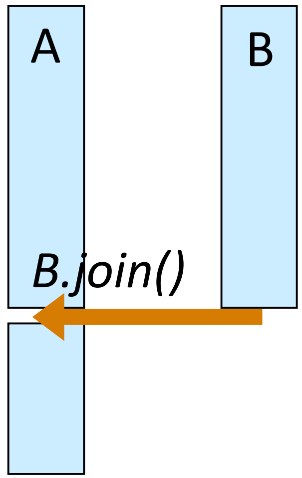
- public int **getParties** ( )   // #required
- public int **getNumberWaiting** ( )   // #waiting at barrier
- public void **reset** ( )

R. Marukatat

Join

- Thread A waits for thread B to complete its execution or die (before A can continue) by calling B.join( )

- Class Thread
  - public final synchronized void **join** ( )
    throws InterruptedException
  - public final synchronized void **join** (long)
    throws InterruptedException
  - public final synchronized void **join** (long, int)
    throws InterruptedException



Exchanger

- Synchronized communication between 2 threads
  - Threads wait for each other at certain point in the program to exchange objects
- Class java.util.concurrent.**Exchanger<V>**
  - V = type of object to be exchanged
  - public **Exchanger<V>** ( )
  - public V **exchange** (V mine) throws InterruptedException
    - Returned value = object from the other thread
    - Exchanger also acts as a barrier ➔ both threads must arrive at the exchanger before exchanging objects

R. Marukatat

Thread A	**Thread B**
Create myValue (=10) … *wait for B at exchanger* yourValue = **exchanger.exchange**(myValue) Result = myValue/2 + yourValue *205 5 200*	Create myValue (=200) … *wait for A at exchanger* yourValue = **exchanger.exchange**(myValue) Result = myValue/2 + yourValue *110 100 10*

R. Marukatat