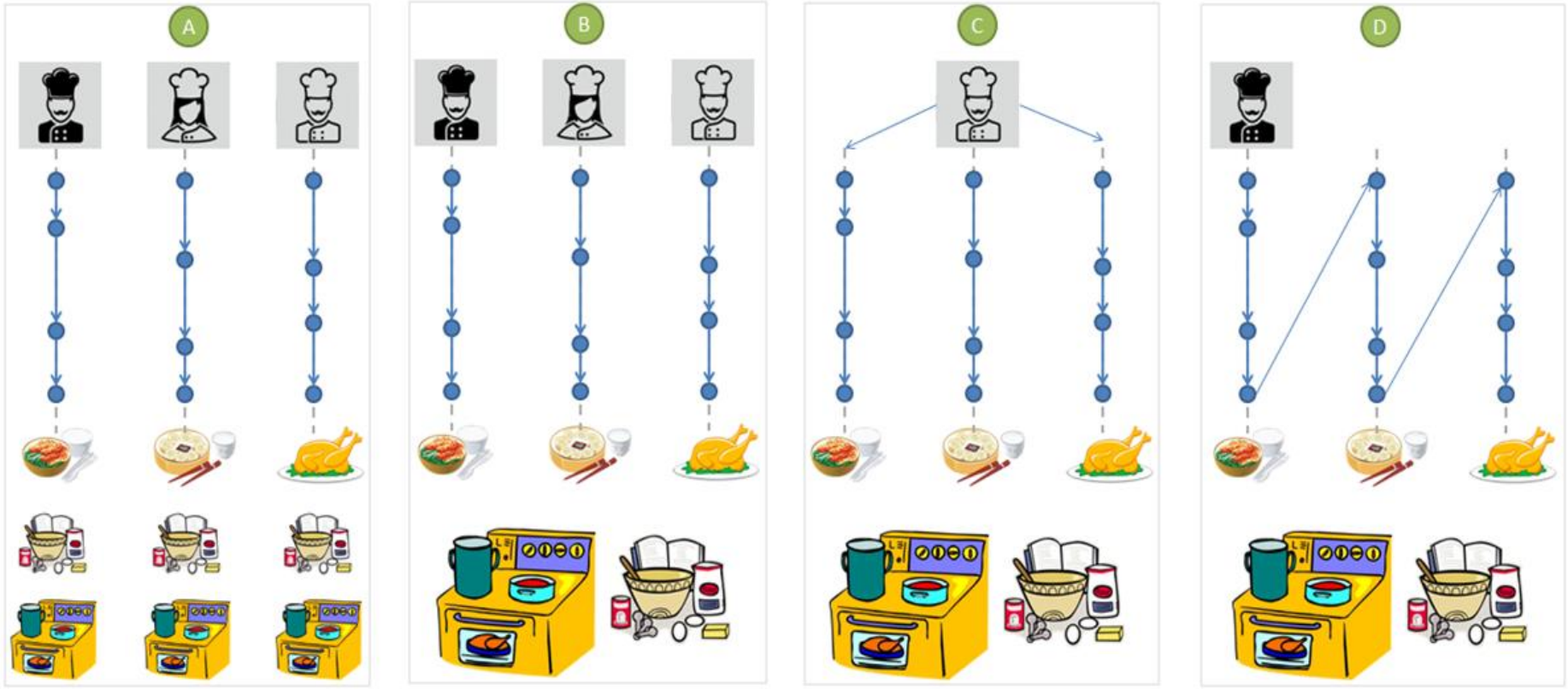
Chương 4
Lập trình xử lý đồng thời (Concurrency)

Nội dung

- Processes và Multi-Processing System
- Threads và Multi-threading
- Cơ bản về Thread trong Java
- Monitors, Waiting và Notifying
- Deadlock

Tại sao cần xử lý đồng thời?



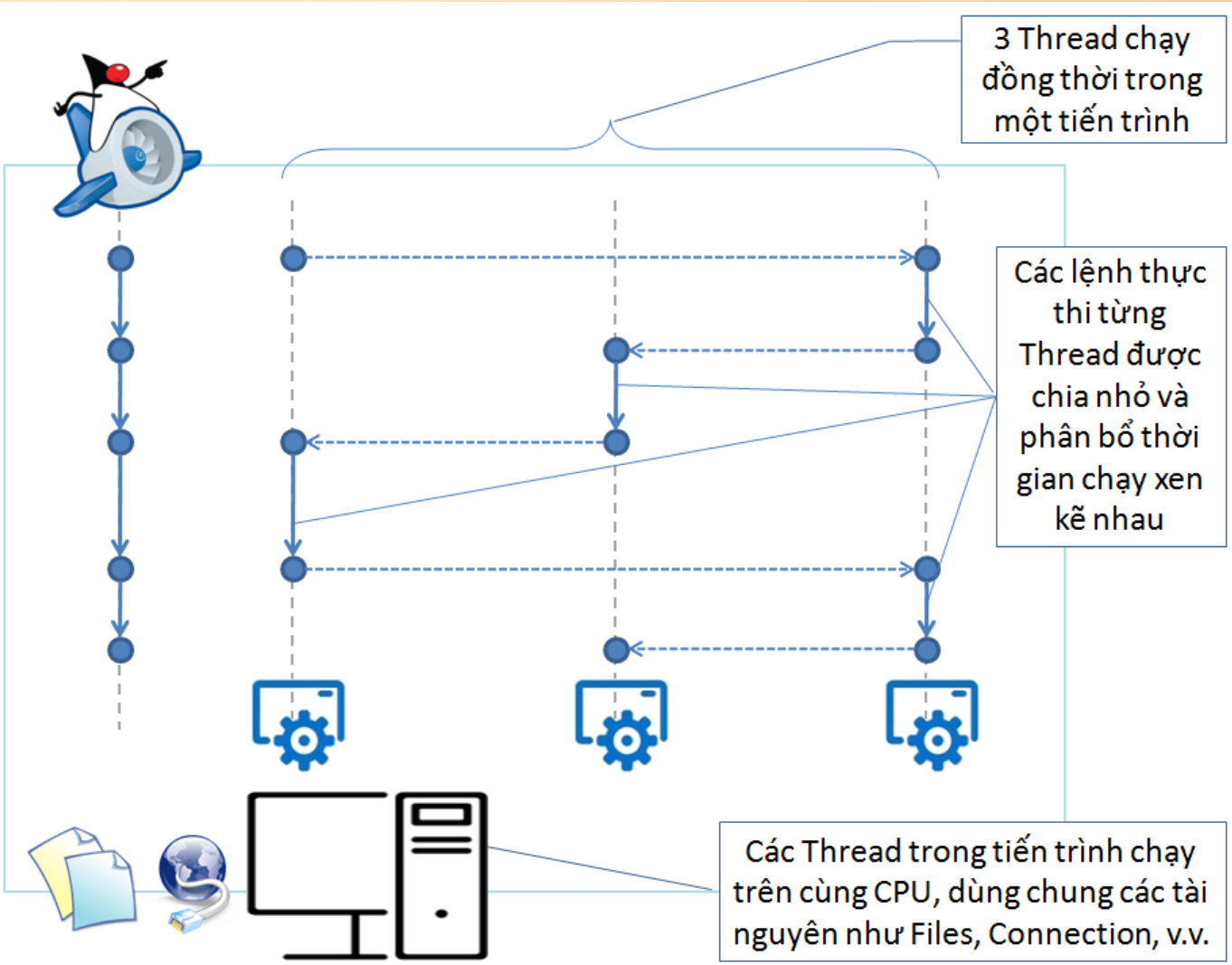
Xử lý đồng thời trong Java

Nấu ba món ăn được diễn ra "đồng thời"

Thao tác nấu được chia nhỏ và phân bổ xen kẽ giữa các món ăn

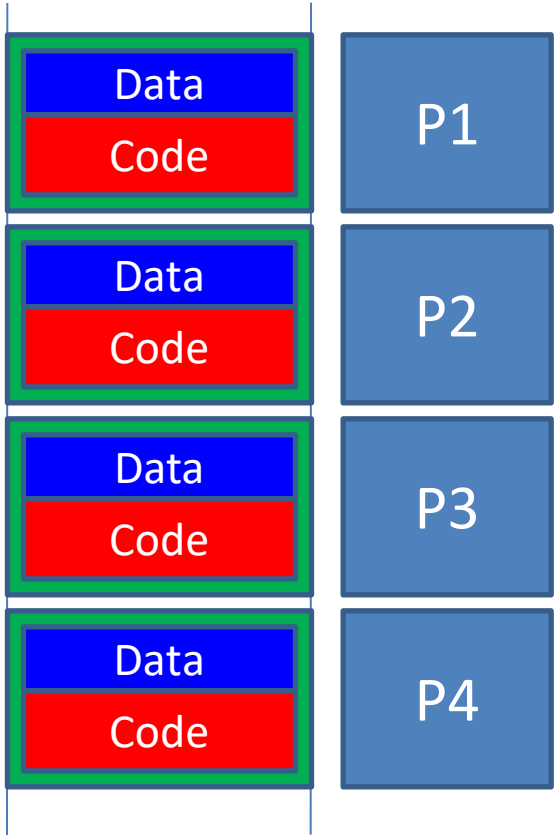
Bếp, nguyên liệu/gia vị được sử dụng chung cho 3 món ăn

Xử lý đồng thời trong Java

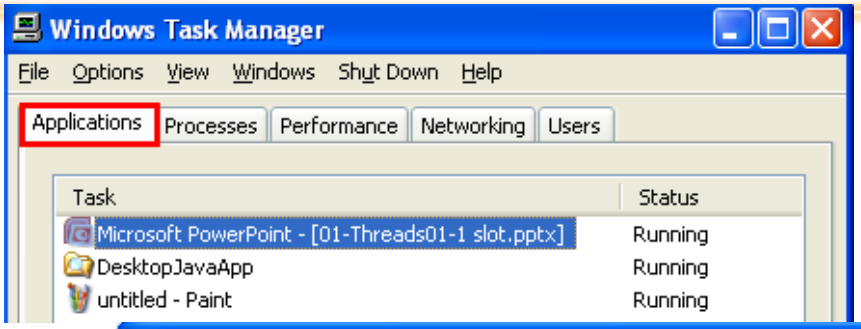


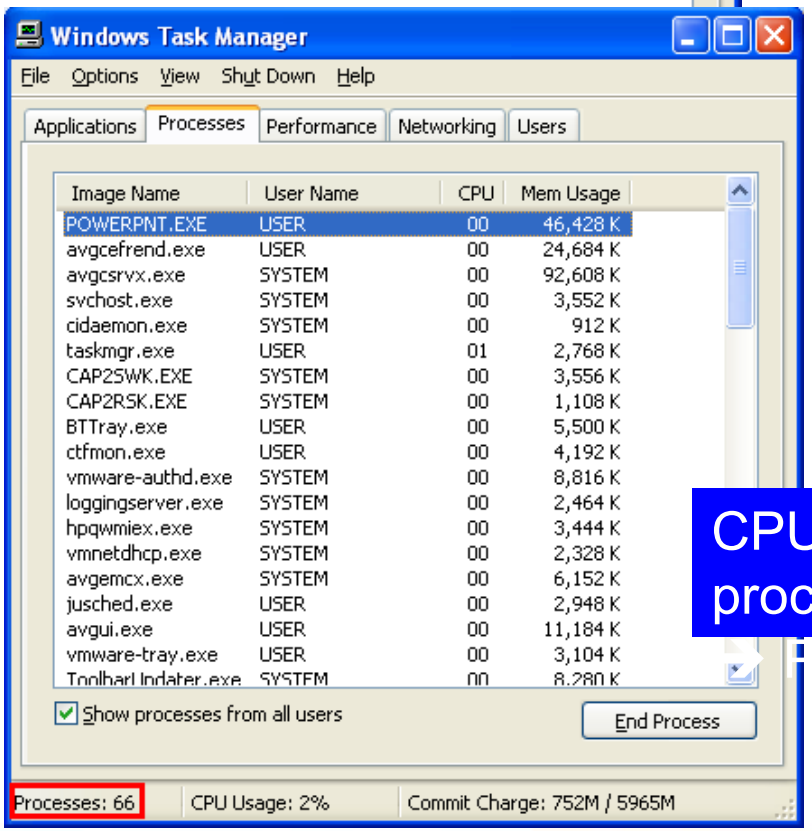
1. Processes và Multi-Processing System

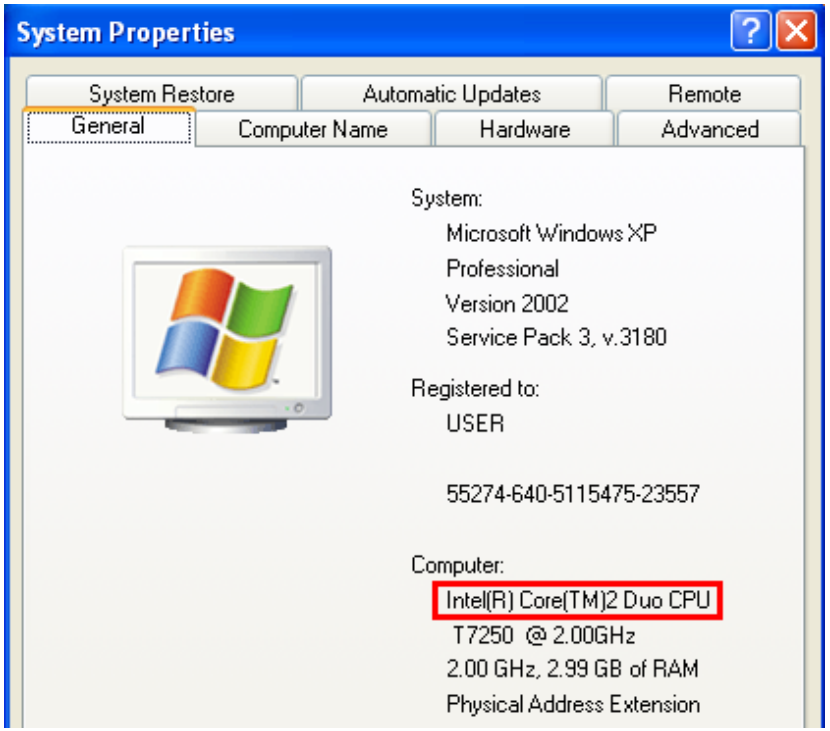
- Program: một file có thể thực thi (data + code) được lưu trữ trong bộ nhớ ngoài (disk).
- Process: Một chương trình đang chạy. Data và code được nạp vào RAM.
- Multi-Processing System: Hệ điều hành cho phép nhiều process thực thi đồng thời.



1. Processes and Multi-Processing System



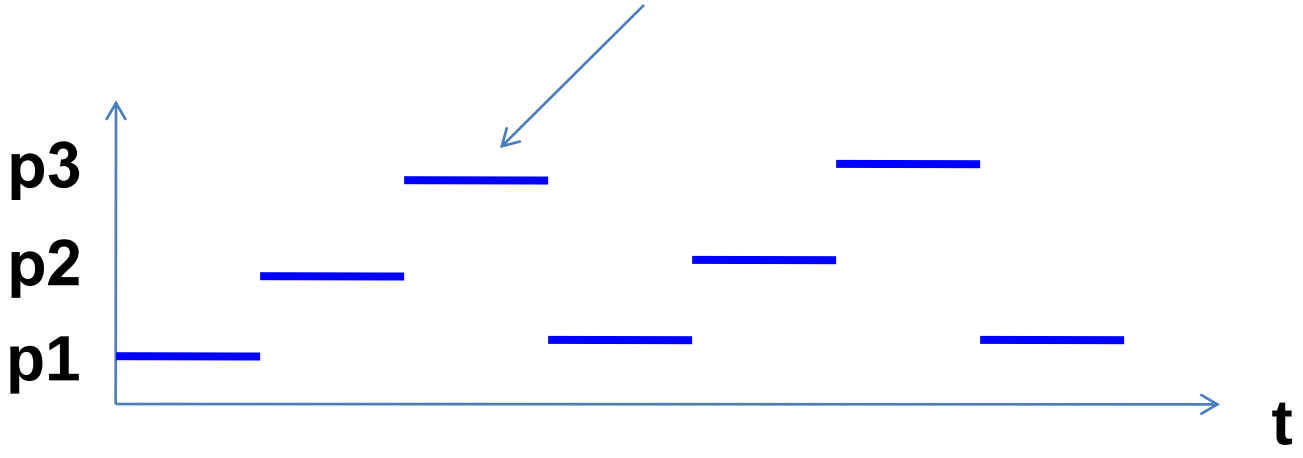




CPU 2 cores → 1 core runs 33 (66/2) processes

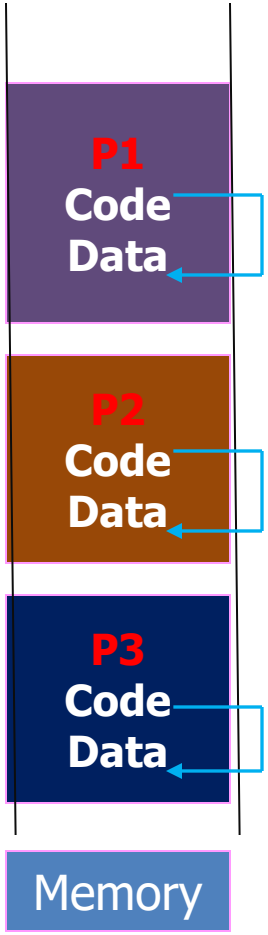
1. Processes và Multi-Processing System

- HĐH quản lý các process trên 1 CPU như thế nào?
  - Cơ chế time-sharing: HĐH cho phép mỗi process thực thi trong một khoảng thời gian (time slice, quantum, khoảng 50 to 70 ms). Khi hết thời gian, process khác sẽ được chọn thực thi → Scheduler (một thành phần của HĐH) sẽ thực hiện chọn process hiện hành (current).



1. Processes và Multi-Processing System

➢ HĐH quản lý các process trên 1 CPU như thế nào?



Process Table

App	Code Addr	Duration (mili sec)	CPU	…
P1	30320	50	1	
P2	20154	60	2	
P3	10166	70	1	
…	…	…	…	
…	…	…	…	

Cơ chế Time-slicing. Mỗi process được cấp phát resources (CPU, …) và thực thi trong một khoảng thời gian (VD 50 milliseconds). Khi hết thời gian, process này sẽ tạm dừng để nhường resource cho process kế tiếp (được chọn bởi scheduler của HĐH).

2. Threads và Multi-threading

- Thread, còn gọi là *lightweight processes,* là một đơn vị code đang thực thi, thực hiện một nhiệm vụ cụ thể.
- Threads tồn tại trong process — **mỗi process có ít nhất một thread (main thread).** Threads chia sẻ resource của process bao gồm bộ nhớ và các file → *vấn đề phát sinh???*
- Cả process và thread đều được cung cấp một môi trường để thực thi, nhưng việc tạo một thread mới đòi hỏi ít tài nguyên hơn việc tạo một process mới.
- Thực thi multi-thread là một đặc điểm cơ bản của Java Flatform. Các thread trong một program được quản lý bởi JVM. Scheduler trong JVM sẽ chọn thread hiện hành.

2. Threads và Multi-threading (tt)

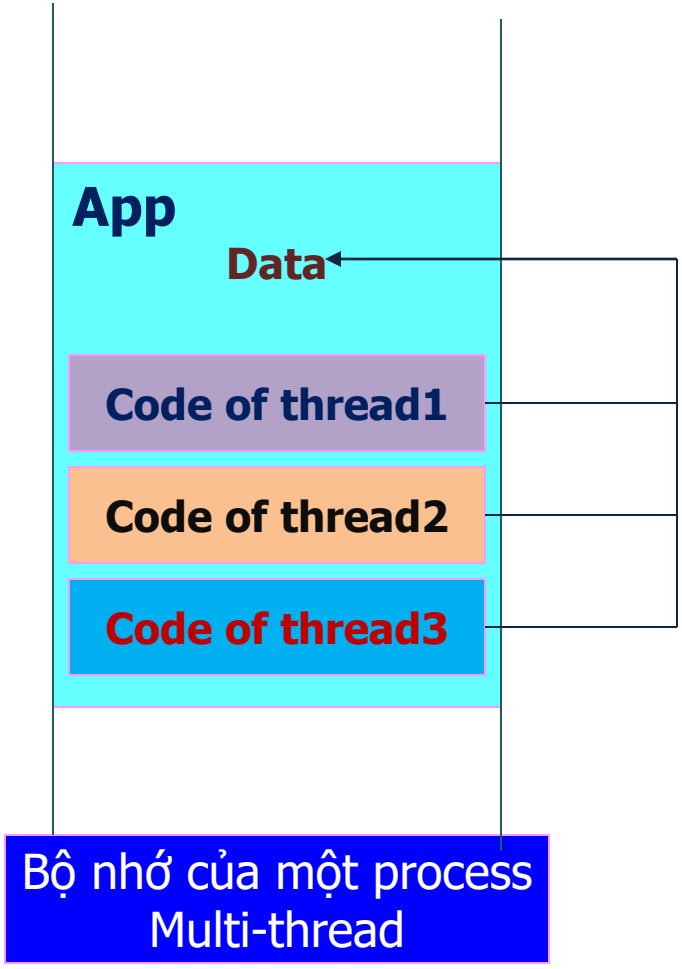
JVM quản lý threads như thế nào?

Một chương trình có thể có một số threads, các threads này được thực thi đồng thời.

Thread Table

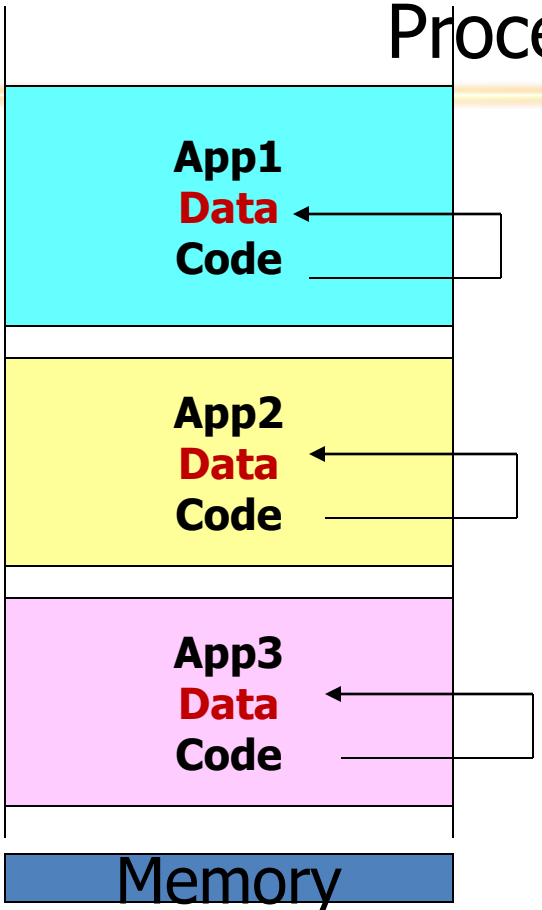
Thread	Code Addr	Duration (mili sec)	CPU	State
Thread 1	10320	15	1	ready
Thread 2	40154	17	2	ready
Thread 3	80166	22	1	sleep
…	…	…	…	….

Sử dụng cơ chế Time-sharing.

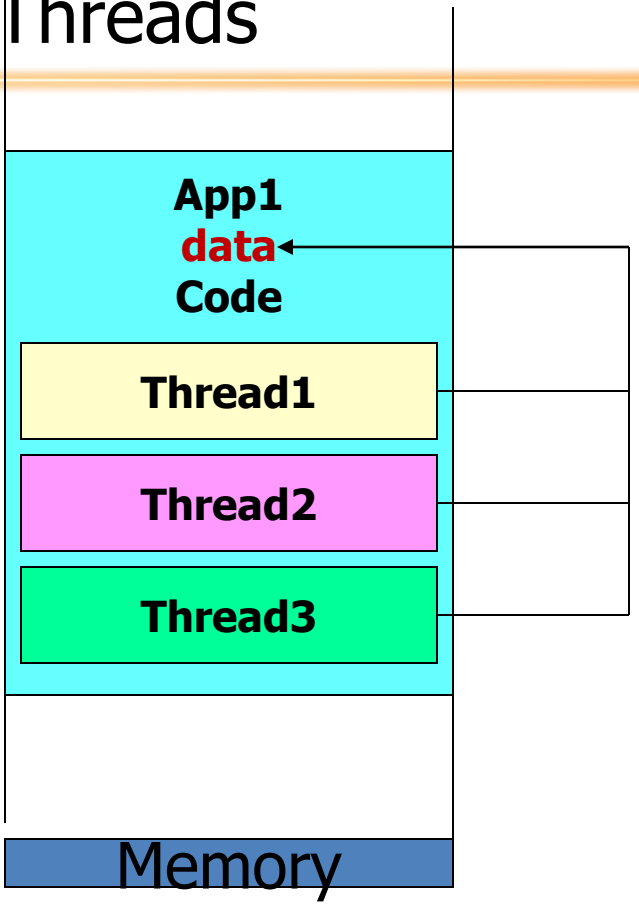


2. Threads và Multi-threading (tt)

Processes VS Threads



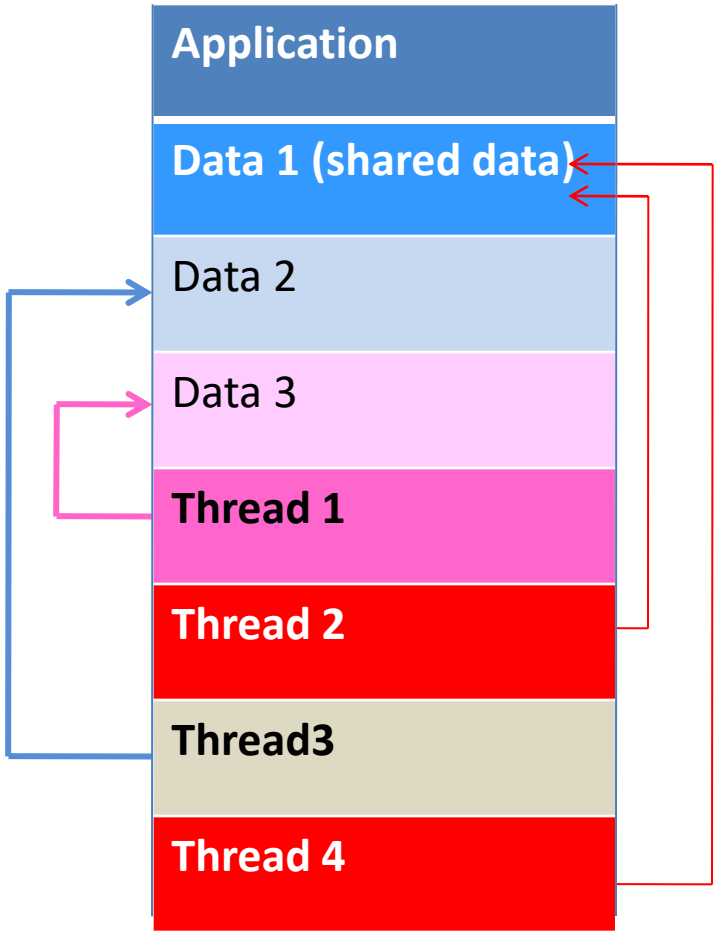
Cơ chế trong HĐH không cho phép process này truy cập bộ nhớ của process khác



Các threads trong một process có thể truy cập dữ liệu chung của process

2. Threads và Multi-threading (tt)

Race Conditions



Giả sử rằng thread2 và thread4 thực thi đồng thời

Time	Thread 2	Thread4
1	**Data1=10**	…
2	….	…
3	….	**Data1= 100**
4	….	…
5	…	…
6	**Y= 2*Data1;**	…
7		…

Y=?

Race xuất hiện

Cần phải Synchronization

3. Cơ bản về Thread trong Java

- **Thread trong Java:**
  - **Class java.lang.Thread**
  - **Class java.lang.Object**
  - **Ngôn ngữ Java và JVM (Java Virtual Machine)**
- **Làm thế nào để tạo Thread trong Java?**

(1) Create một subclass của class **java.lang.Thread** và override phương thức **run()**

(2) Create một class implementing interface **Runnable** và override phương thức **run()**.

Tạo một subclass của class Thread

Thread1.java * | Thread1Using.java

```
1  /* Thread1.java */
2    public class Thread1 extends Thread {
3
4        public Thread1() {
5            super ();
6        }
7        public void run() {
8            System.out.println("I'm a child thread");
9        }
10   }
```

Thread1.java * | Thread1Using.java

```
1  public class Thread1Using {
2      public static void main(String args[])
3      {   Thread1 t= new Thread1();
4          System.out.println("I'm the main thread.");
5          t.start();
6          System.out.println("Hello");
7      }
8  }
```

Thread table

Thread	Code Addr	State	…
main	8000	run	…
t	10320	run	

Mỗi process có ít nhất một thread (main thread).

Output - ThreadDemo (run-single)

```
compile:
run-single:
I'm the main thread.
Hello
I'm a child thread
```

Class implements interface Runnable

RunnableThread.java * | RunnableClassUsing.java

```
1  public class RunnableThread implements Runnable {
2
3      public RunnableThread() {
4      }
5      public void run() {
6          System.out.println("I'm a child thread");
7      }
8  }
```

Thread	Code Addr	State	…
main	8000	run	…
t	10320	run	

RunnableThread.java * | RunnableClassUsing.java *

```
1  public class RunnableClassUsing {
2
3      public static void main(String args[]) {
4          RunnableThread obj= new RunnableThread();
5          Thread t= new Thread (obj);
6          t.start();
7          System.out.println("I'm the main thread.");
8          System.out.println("Hello");
9      }
10 }
```

Output - ThreadDemo (run-single)

```
compile:
run-single:
I'm the main thread.
Hello
I'm a child thread
```

Class java.lang.Thread

Declaration
public class **Thread** extends **Object** implements **Runnable**

set/get-is

Properties
id
name
state
threadGroup
daemon
priority

Common Methods
start()
join ()
sleep (milisec)
yield()
notify()
notifyAll()
wait()

Constructor
Thread()
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, String name)

static int	MAX_PRIORITY	10
static int	MIN_PRIORITY	1
static int	NORM_PRIORITY	5

17

Class java.lang.Thread (tt)

Phương thức	Ý nghĩa
final String getName()	Lấy tên của thread
final int getPriority()	Lấy thứ tự ưu tiên của thread
final Boolean isAlive()	Kiểm tra một thread có còn chạy hay không
final void join()	Chờ đến khi một thread ngừng hoạt động
void run()	Chạy một thread
static void sleep(long milliseconds)	Tạm ngừng hoạt động một thread trong khoảng thời gian milliseconds
void start()	Bắt đầu một thread bằng cách gọi run

Sử dụng một số method của class Thread

ThreadProperties.java

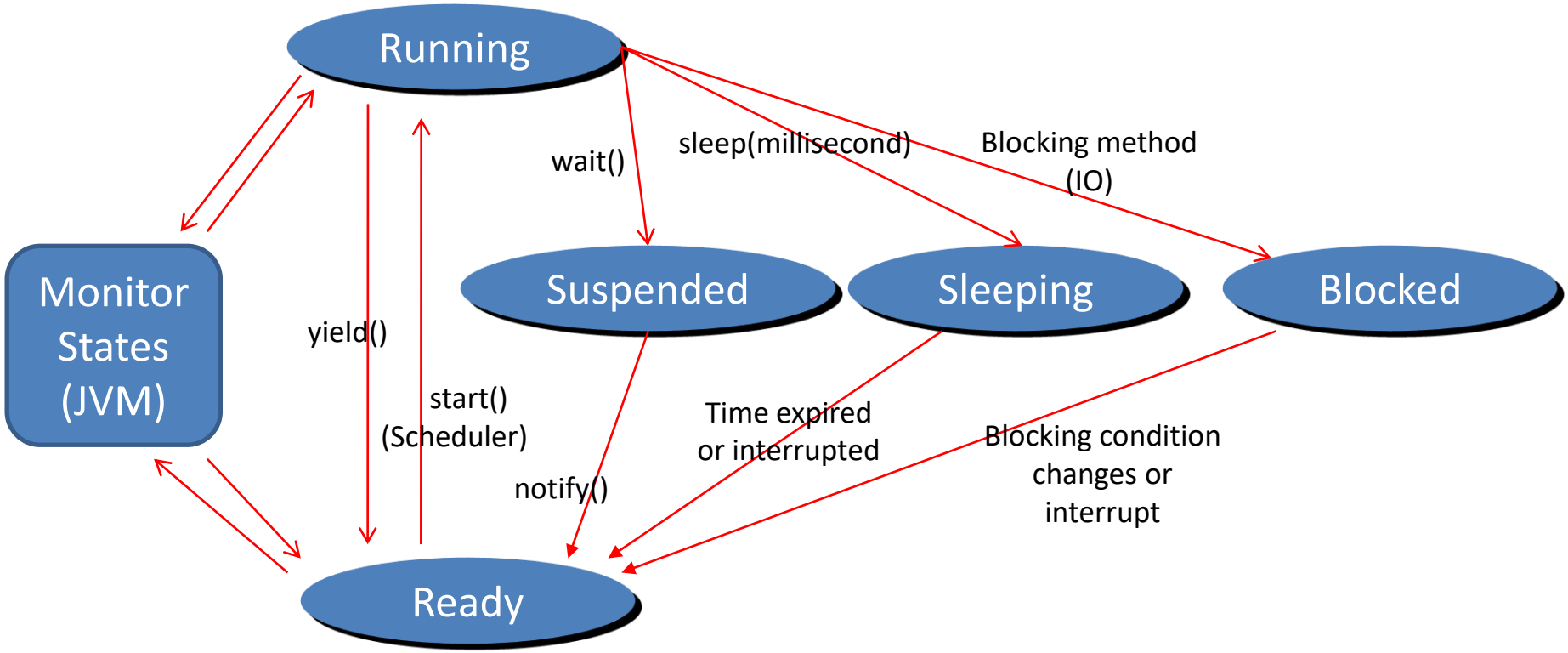
```
1  public class ThreadProperties extends Thread{
2      public ThreadProperties(String threadName) {
3        super(threadName);
4        this.start();
5      }
6      public static void showProperties(Thread t){
7        System.out.println("I'm the " + t.getName() + " thread");
8        System.out.println("--My ID:" + t.getId());
9        System.out.println("--My name:" + t.getName());
10       System.out.println("--My priority:" + t.getPriority());
11       System.out.println("--My state:" + t.getState());
12       System.out.println("--I'm a deamon:" + t.isDaemon());
13       System.out.println("--I'm alive:" + t.isAlive());
14     }
15     public void run(){
16       showProperties(this);
17     }
18     public static void main (String args[]){
19       System.out.println("Thread count:" + Thread.activeCount());
20       Thread t= Thread.currentThread();
21       showProperties(t);
22       ThreadProperties t1= new ThreadProperties("Son1");
23       ThreadProperties t2= new ThreadProperties("Son2");
24       System.out.println("Thread count:" + Thread.activeCount());
25     }
26 }
```

Output

Debugger Console | ThreadD

```
run-single:
Thread count:1
I'm the main thread
--My ID:1
--My name:main
--My priority:5
--My state:RUNNABLE
--I'm a deamon:false
--I'm alive:true
Thread count:3
I'm the Son2 thread
--My ID:9
--My name:Son2
--My priority:5
--My state:RUNNABLE
--I'm a deamon:false
--I'm alive:true
I'm the Son1 thread
--My ID:8
--My name:Son1
--My priority:5
--My state:RUNNABLE
--I'm a deamon:false
--I'm alive:true
```

Các trạng thái của Thread

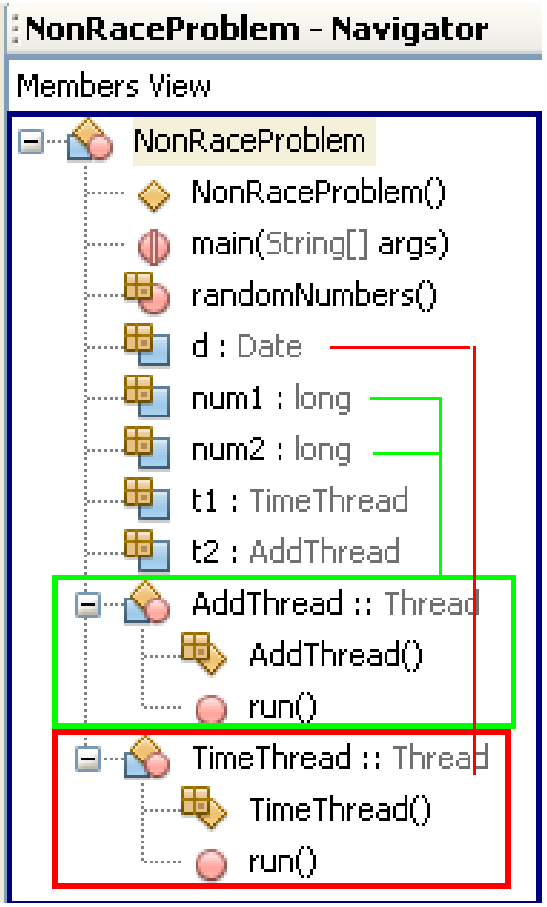


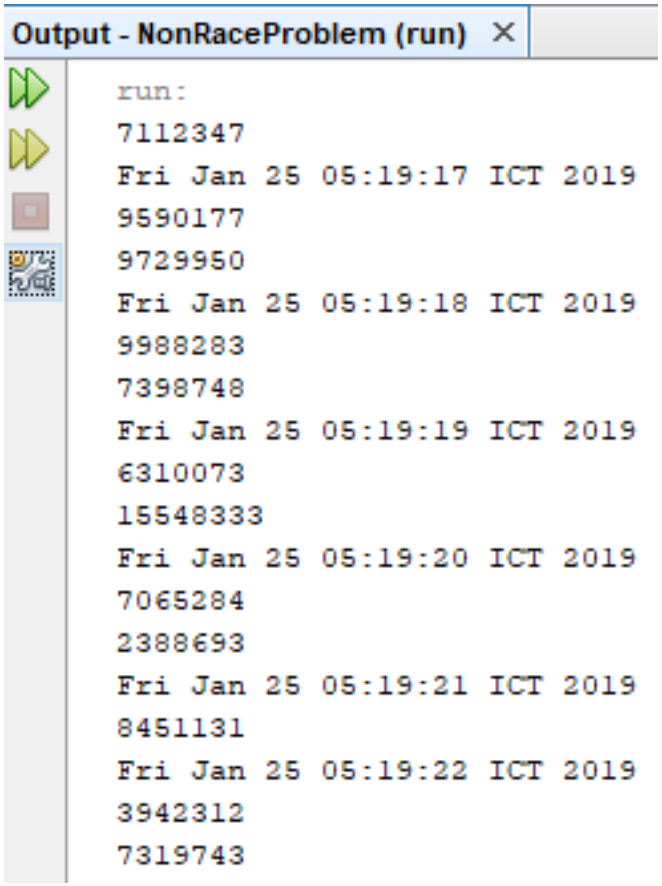
Mỗi thời điểm chỉ có một thread ready được chọn bởi JVM scheduler.

Minh họa Non-race

Chương trình gồm 2 threads:

- Thread thứ nhất sẽ xuất ra thời gian của hệ thống mỗi giây một lần.
- Thread thứ hai sẽ xuất ra tổng của 2 số nguyên ngẫu nhiên mỗi ½ giây một lần.





Minh họa Non-race (tt)

```
1  import java.util.Date;
2  public class NonRaceProblem {
3      Date d=null;
4      long num1=0, num2=0;
5      // 2 threads of inner Threads, declared belo
6      TimeThread t1 = new TimeThread ();
7      AddThread t2 = new AddThread ();
8
9      public NonRaceProblem() {
10         d=new Date(System.currentTimeMillis());
11         randomNumbers(); t1.start(); t2.start();
12     }
13     void randomNumbers(){
14         num1= Math.round(Math.random()*10000000);
15         num2= Math.round(Math.random()*1000000
16     }
```

```
17     // Inner class 1: Thread for printing out the time
18     class TimeThread extends Thread{
19         TimeThread() { super(); }
20         public void run() {
21             while (true){
22                 try {
23                     System.out.println(d);
24                     this.sleep(1000);
25                     d=new Date(System.currentTimeMillis());
26                 }
27                 catch(java.lang.InterruptedException e){
28                     e.printStackTrace();
29                 }
30             }
31         }
```

```
33 // Inner class 2: Thread for printing out sum of 2 numbers
34 class AddThread extends Thread{
35     AddThread() { super(); }
36     public void run() {
37         while (true){
38             try { System.out.println(num1+ num2);
39                   randomNumbers();
40                   this.sleep(500);
41             }
42             catch(java.lang.InterruptedException e){
43                 e.printStackTrace();
44             }
45         }
46     }
47 }
```

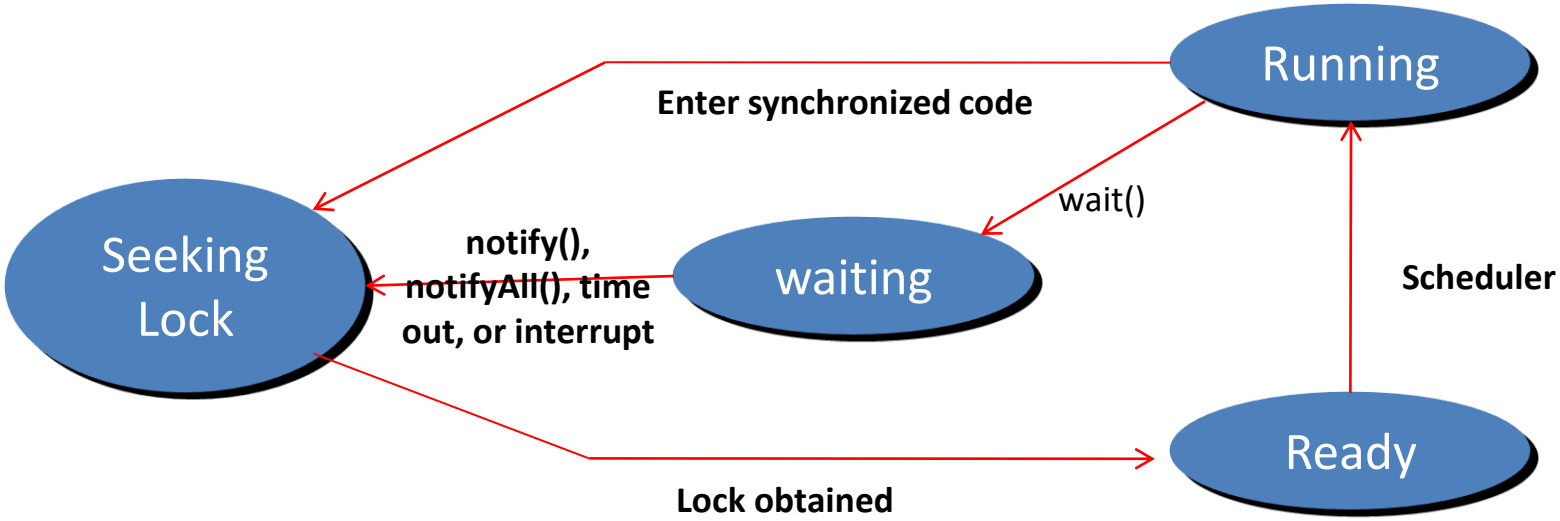
Output - NonRaceProblem (run)

```
run:
7112347
Fri Jan 25 05:19:17 ICT 2019
9590177
9729950
Fri Jan 25 05:19:18 ICT 2019
9988283
7398748
Fri Jan 25 05:19:19 ICT 2019
6310073
15548333
```

4. Monitors, Waiting và Notifying

- Một số thread có thể truy cập đồng thời tài nguyên dùng chung. Phải đồng bộ hóa việc truy cập các tài nguyên dùng chung ➔ Mỗi đối tượng có một khóa (lock).
- *Lock*: một biến được compiler thêm vào để giám sát trạng thái của tài nguyên dùng chung. Trước khi một thread truy cập tài nguyên dùng chung, lock sẽ được kiểm tra.
- Sau khi một thread đã có lock (được phép truy cập), nó có thể truy cập tài nguyên dùng chung. Khi thực hiện xong, thread phải thông báo (notify) cho các thread khác (cơ chế wait-notify).

4. Monitors, Waiting and Notifying (tt)



4. Monitors, Waiting and Notifying (tt)

- *Hai cách để đánh dấu code là đồng bộ hóa*
  - Đồng bộ hóa toàn bộ phương thức:

```
synchronized  Type Method(args){
    <code>
}
```

  - Đồng bộ hóa một số phương thức của một đối tượng

```
Type Method ( args){
    ……
    synchronized ( object_var){
      object_var.method1(args);
      object_var.method2(args);
    }
}
```

Ví dụ đồng bộ hóa thread

```
public class SynchronizedMethod implements Runnable{
    public void run(){
        for (int i = 0; i < 5; i++){
            System.out.println(Thread.cur
            try{
                Thread.sleep(200);
            }
            catch(Exception ex){
                System.out.println();
            }
        }
        System.out.println("Finished " + 
    }
}
public static void main(String[] args) {
    SynchronizedMethod m1 = new Synchroniz
    Thread t1 = new Thread(m1);
    Thread t2 = new Thread(m1);
    Thread t3 = new Thread(m1);
    t1.setName("Thread 1"); t2.setName("Thread 2");t3.setName("Thread 3");
    t1.start(); t2.start();t3.start();
}
```

Output - SynchronizedDemo (run)

```
run:
Thread 1: 0
Thread 3: 0
Thread 2: 0
Thread 2: 1
Thread 3: 1
Thread 1: 1
Thread 1: 2
Thread 3: 2
Thread 2: 2
Thread 2: 3
Thread 1: 3
Thread 3: 3
Thread 1: 4
Thread 2: 4
Thread 3: 4
Finished Thread 3
Finished Thread 1
Finished Thread 2
BUILD SUCCESSFUL (total time: 1 second)
```

Ví dụ đồng bộ hóa thread (tt)

```
public class SynchronizedMethod implements Runnable{
    public synchronized void run(){
        for (int i = 0; i < 5; i++){
            System.out.println(Thread.currentThrea
            try{
                Thread.sleep(200);
            }
            catch(Exception ex){
                System.out.println();
            }
        }
        System.out.println("Finished " + Thread.cu
    }
}
```

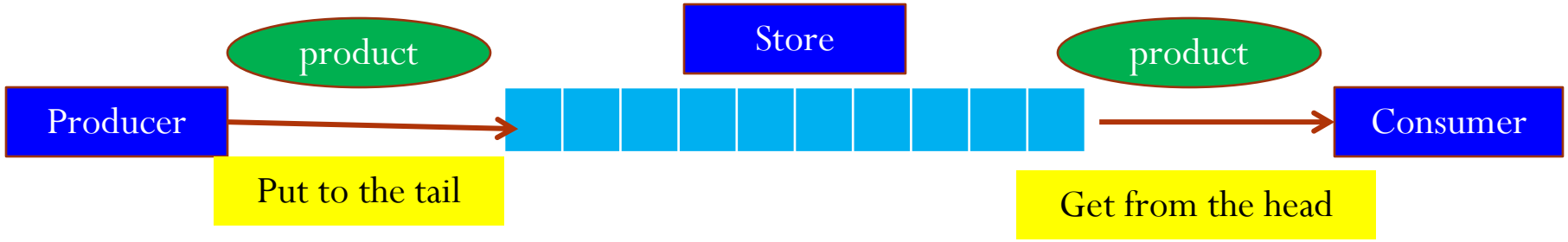
```
 public static void main(String[] args) {
     SynchronizedMethod m1 = new SynchronizedMetho
     Thread t1 = new Thread(m1);
     Thread t2 = new Thread(m1);
     Thread t3 = new Thread(m1);
     t1.setName("Thread 1"); t2.setName("Thread 2");t3.setName("Thread 3");
     t1.start(); t2.start();t3.start();
 }
```

Output - SynchronizedDemo (run)

```
run:
Thread 2: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
Finished Thread 2
Thread 3: 0
Thread 3: 1
Thread 3: 2
Thread 3: 3
Thread 3: 4
Finished Thread 3
Thread 1: 0
Thread 1: 1
Thread 1: 2
Thread 1: 3
Thread 1: 4
Finished Thread 1
```

The Producer-Consumer Problem

- Producer makes a product then puts it to a store.
- Consumer buys a product from a store.
- Selling Procedure: First In First Out



Attention!:
Store is common resource of 2 threads: producer and consumer.
➔ Store is a monitor and it's activities needs synchronization

Synchronizing:
* After a thread accessed common resource, it should sleep a moment or it must notify to the next thread (or all thread) in the thread-pool to awake and execute.
* Use the **synchronized** keyword to declare a method that will access common resource.

The Producer-Consumer Problem (tt)

```
public class Store {
    int maxN=50;  // maximum number of products can be contains int the store
    long [] a;    // product list
    int n;        // current number of products
    public Store() {  n=0; a=new long[maxN];  }
    private boolean empty() { return n==0; }
    private boolean full() { return n==maxN; }
    public /* synchronized */ boolean put(long x){
       if (full()) return false;
       a[n++]=x;
       try  { Thread.sleep(500); }
       catch (Exception e){  }
       return true;
    }
    public /* synchronized */ long get(){
       long result=0;
       if (!empty()) {
           result=a[0]; // get the product at the front of line
           for (int i=0;i<n-1;i++) a[i]=a[i+1]; // shift products up.
           n--;
       }
       try  {  Thread.sleep(500);  }
       catch (Exception e){       }
       return result;
    }
}
```

A product is simulated as a number.

/* synchronized */
No synchronization

The Producer-Consumer Problem (tt)

```
public class Producer extends Thread{
    Store store=null;
    long index=1;     // index of product that will be made
    public Producer(Store s) {
        store=s;
    }
    public void run(){
        while (true){
            try {
                    boolean result= store.put(index);
                    if (result==true) System.out.println("** Product " + (index++)+ " is made.");
                    else System.out.println("Store is full!");
             }
            catch (Exception e) {
            }
        }
    }
}
```

The Producer-Consumer Problem (tt)

```
public class Consumer extends Thread {
    Store store=null;
    public Consumer(Store s) {
        store=s;
    }
    public void run(){
        while (true){
            try {
                long x= store.get();
                if (x>0) System.out.println("-- Product " + x + " is bought.");
                else System.out.println("Consumer is waiting for new product.");
            }
            catch (Exception e) {
            }
        }
    }
}
```

```
public class ProducerConsumerProblem {
    Store store;
    Producer pro;
    Consumer con;
    public ProducerConsumerProblem() {
        store= new Store();  pro= new Producer(store); con= new Consumer(store);
        pro.start(); con.start();
    }
    public static void main (String args[]){
        ProducerConsumerProblem obj=new ProducerConsumerProblem();
    }
}
```

The Producer-Consumer Problem (tt)

Synchronization is not used

Output - ThreadDemo (run-single) #2

```
compile:
run-single:
** Product 1 is made.
-- Product 1 is bought.
-- Product 2 is bought.
** Product 2 is made.
Consumer is waiting for new product.
** Product 3 is made.
** Product 4 is made.
-- Product 3 is bought.
** Product 5 is made.
-- Product 4 is bought.
```

Synchronization is used:

Output - ThreadDemo (run-single)

```
run-single:
** Product 1 is made.
** Product 2 is made.
-- Product 1 is bought.
-- Product 2 is bought.
Consumer is waiting for new product.
Consumer is waiting for new product.
Consumer is waiting for new product.
Consumer is waiting for new product.
** Product 3 is made.
** Product 4 is made.
-- Product 3 is bought.
-- Product 4 is bought.
Consumer is waiting for new product.
Consumer is waiting for new product.
```

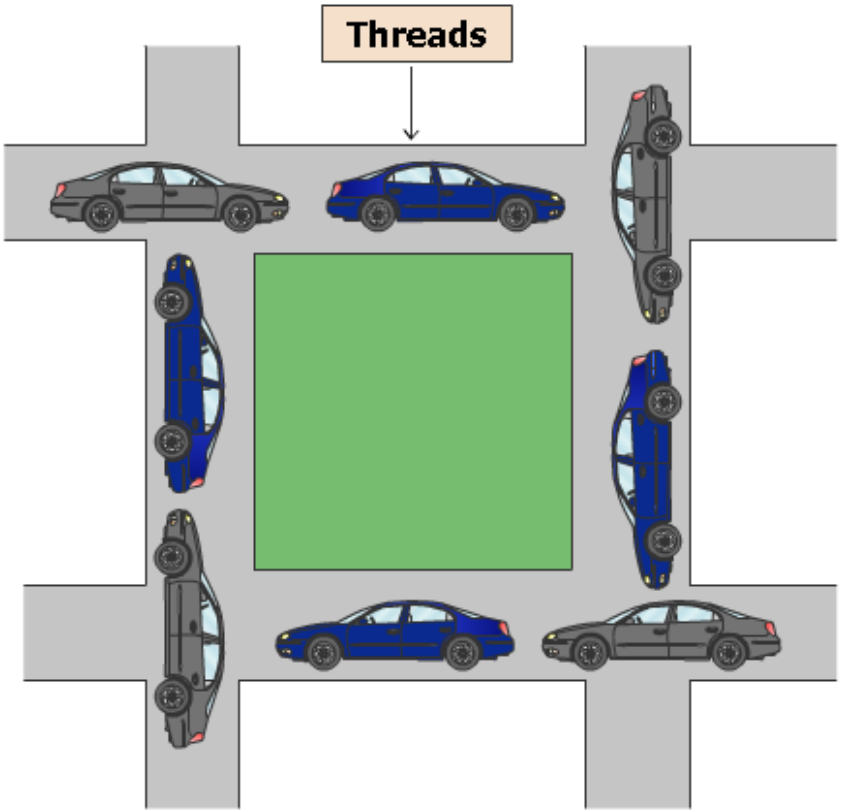
5. Deadlock

Deadlock là gì?

Deadlock mô tả một tình huống mà ở đó hai hay nhiều thread bị block vĩnh viễn, đợi lẫn nhau.

Deadlock xuất hiện khi nào?

Tồn tại một vòng chờ khóa được giữ bởi các thread khác nhau.



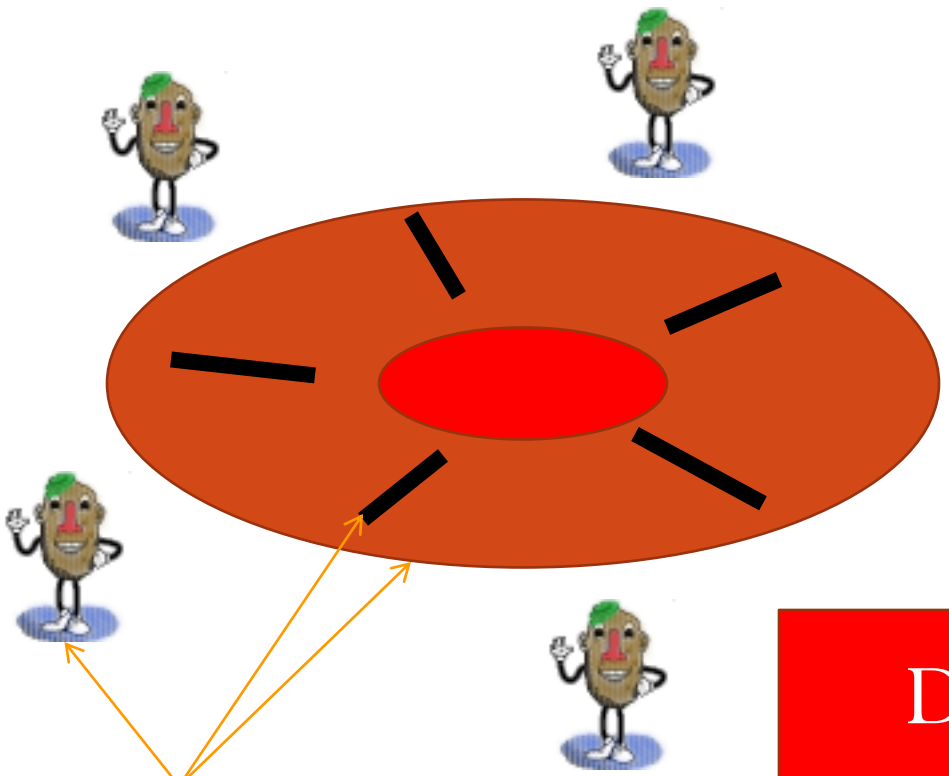
Deadlock (tt)

```
public class DeadLockDemo implements Runnable  {
   DeadLockDemo assistance=null; // giám đốc có trợ lý
   int a=100, b=200;
   public synchronized void changeValues(){
    try{ Thread.sleep(500); a++; b++;  }
    catch(Exception e) { }
   }
   public synchronized void run(){
    while (true)
    {  try { System.out.println(Thread.currentThread().getName());
          System.out.println("a=" + a);
          System.out.println("b=" + b);
          Thread.sleep(500);
       }
       catch(Exception e) { }
       assistance.changeValues();
     }
   }
```

```
Output - ThreadDemo (run-single) #2
init:
deps-jar:
compile-single:
run-single:
Thread-2
a=100
b=200
Thread-1
a=100
b=200
```

```
   public static void main(String args[]) {
      DeadLockDemo person1= new DeadLockDemo();
      DeadLockDemo person2= new DeadLockDemo();
      person1.assistance= person2;  // hai giám đốc
      person2.assistance= person1;  // lại là trợ lý của nhau
      Thread t1= new Thread(person1,"Thread-1");
      Thread t2= new Thread(person2,"Thread-2");
      t1.start();
      t2.start();
      try {
           t1.join();// t1 will be executed to the end
           t2.join();// t2 will be executed to the end
       }
       catch(Exception e) { }
   }
}
```

The Philosophers Problem



Cơ chế Wait-notify, một cách giúp ngăn chặn deadlock

Deadlock!

3 classes

The Philosophers Problem

```
package threadpkg;
public class ChopStick {
    boolean ready;
    ChopStick(){
        ready=true;
    }
    public synchronized void getUp()
      { while (!ready)
          { try {
              System.out.println("A philosopher is waiting for a chopstick.");
               wait();
             }
            catch (InterruptedException e){
               System.out.println ("An error occured!");
             }
          }
        ready=false;
      }
    public synchronized void getDown
    { ready= true;
      notify();
    }
}
```

Thread table

Thread	Code Addr	Duration (mili sec)	CPU	State
Thread 1	10320	15	1	Suspended →Ready
Thread 2	40154	17	2	Suspended
Thread 3	80166	22	1	Suspended
…	…	…	…	….

The Philosophers Problem

```
package threadpkg;
public class Philosopher extends Thread{
   ChopStick leftStick, rightStick;  // He/she needs 2 chop sticks
   int position;                     // His/her position at the dinner table
   Philosopher(int pos, ChopStick lStick, ChopStick rStick)
   { position=pos ; leftStick=lStick; rightStick=rStick;
   }
   public void eat()
   { leftStick.getUp();  rightStick.getUp();
     System.out.println("The " + position +"(th) philosopher is eating");
   }
   public void think()
   { leftStick.getDown(); rightStick.getDown();
     System.out.println("The " + position +"(th) philosopher is thinking\"");
   }
```

The Philosophers Problem

Philosopher.java

```
    public void run()
    { while (true)
      { eat();
        try   {  sleep(1000); }
        catch (InterruptedException e)
        { System.out.println("An error occurred!");
        }
        think();
        try {  sleep(1000); }
        catch (InterruptedException e)
        { System.out.println("An error occurred!");
        }
      }
    }
}
```

The Philosophers Problem

```
package threadpkg;
public class DinnerTable {
  static int n;
  static ChopStick[] sticks = new ChopStick[5];
  static Philosopher[] philosopers = new Philosopher[5];

  public static void main (String args[])
  { n=5;
    int i;
    for (i=0;i<n;++i) sticks[i]=new ChopStick();
    for (i=0;i<n;++i) philosopers[i] =
          new Philosopher (i,sticks[i],sticks[(i+1)%5]);
    for (i=0;i<n;++i) philosopers[i].start();
  }
}
```