Exclusion mutuelle

Deep Philosophical Question

- The Bakery Algorithm is
  - Succinct,
  - Elegant, and
  - Fair.
- Q: So why isn't it practical?
- A: Well, you have to read N distinct variables

Shared Memory

- Shared read/write memory locations called Registers (historical reasons)
- Come in different flavors
  - Multi-Reader-Single-Writer (Flag[])
  - Multi-Reader-Multi-Writer (**Victim**[])
  - Not that interesting: SRMW and SRSW

Theorem

At least N MRSW (multi-reader/ single-writer) registers are needed to solve deadlock-free mutual exclusion.

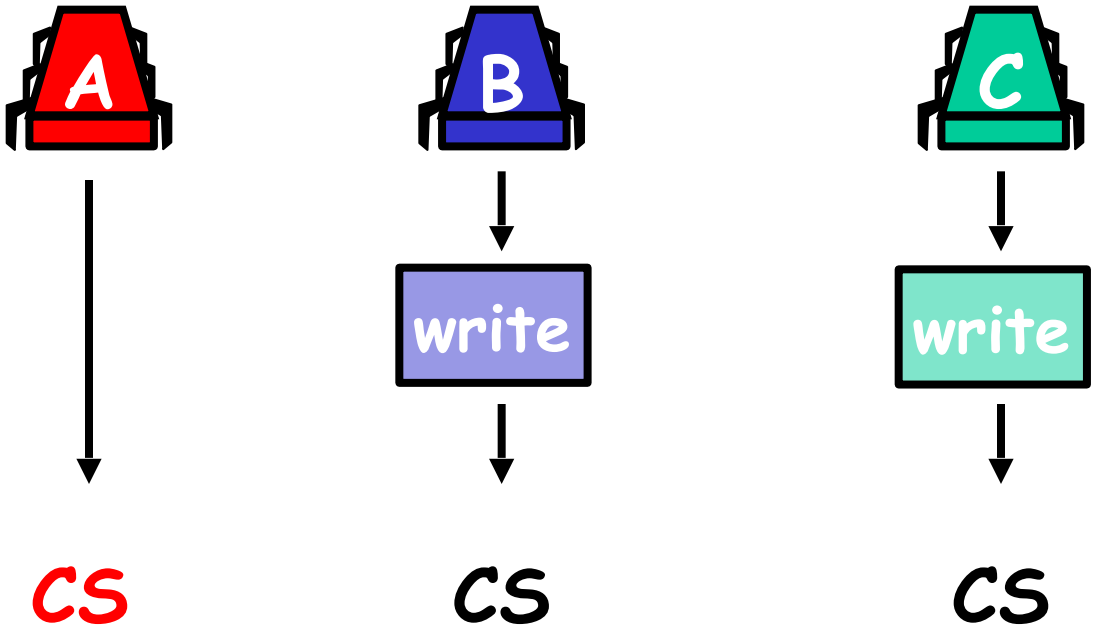
N registers like Flag[]…

Proving Algorithmic Impossibility

- To show no algorithm exists:
  - assume by way of contradiction one does,
  - show a bad execution that violates properties:
  - in our case assume an alg for deadlock free mutual exclusion using < N registers

Proof: Need N-MRSW Registers

Each thread must write to some register



…can't tell whether A is in critical section

Upper Bound

- Bakery algorithm
  - Uses 2N MRSW registers
- So the bound is (pretty) tight
- But what if we use MRMW registers?
  - Like **victim[]** ?

Bad News Theorem

At least N MRMW multi-reader/ multi-writer registers are needed to solve deadlock-free mutual exclusion.

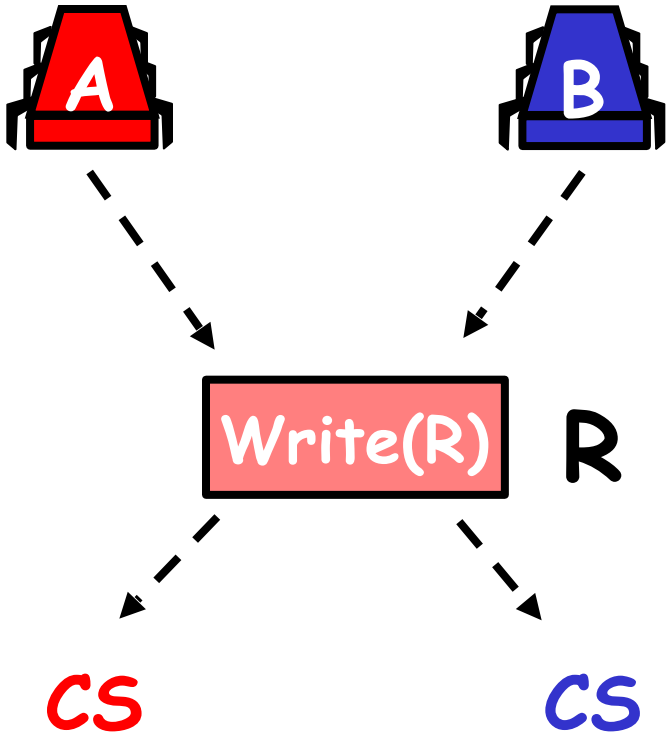
(So multiple writers don't help)

Theorem (First 2-Threads)

Theorem: Deadlock-free mutual exclusion for 2 threads requires at least 2 multi-reader multi-writer registers

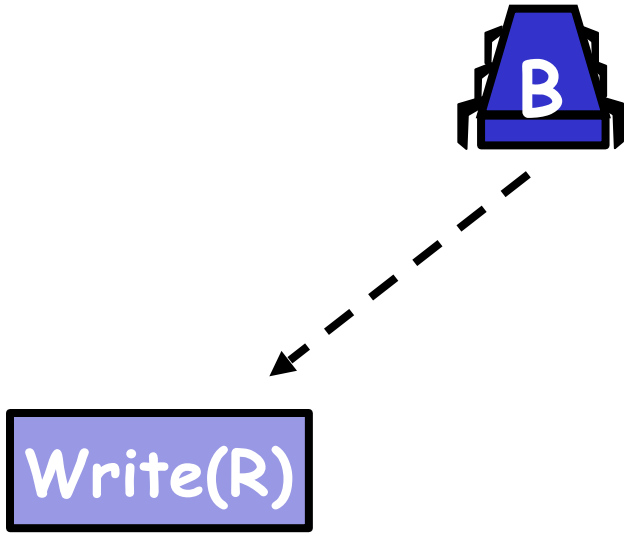
Proof: assume one register suffices and derive a contradiction

Two Thread Execution



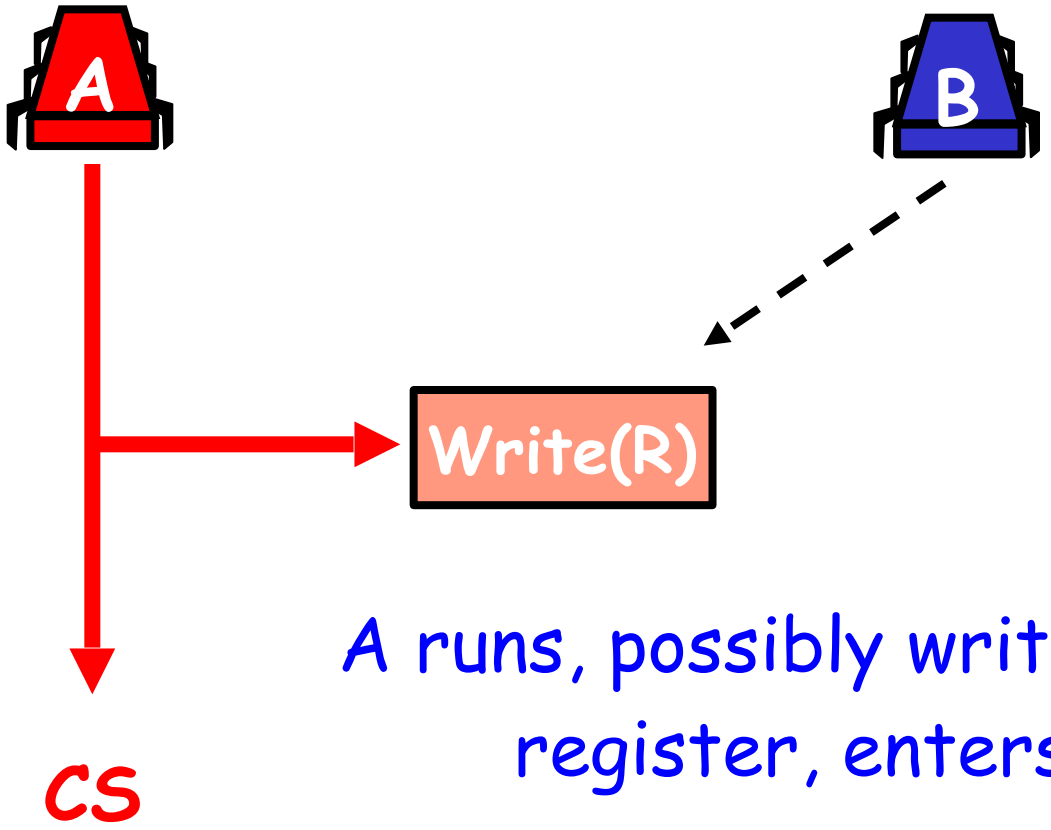
- Threads run, reading and writing R
- Deadlock free so at least one gets in

Covering State for One Register Always Exists



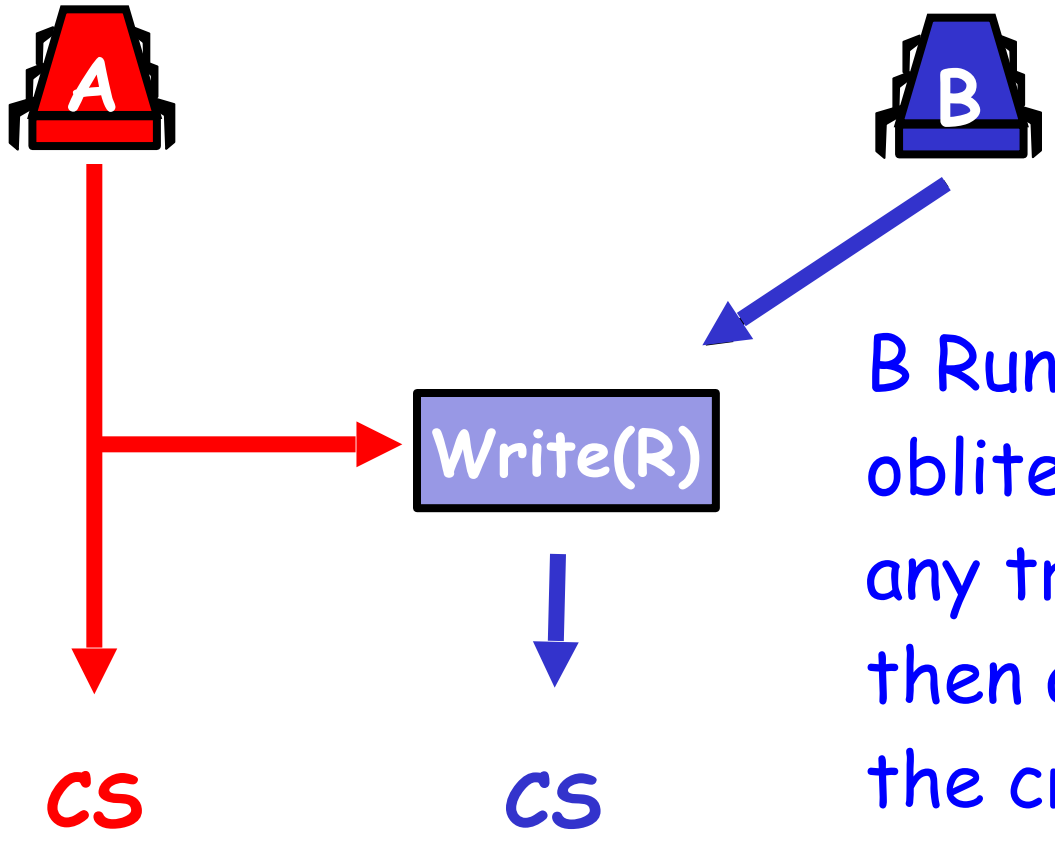
In any protocol B has to write to the register before entering CS, so stop it just before

Proof: Assume Cover of 1



A runs, possibly writes to the register, enters CS

Proof: Assume Cover of 1

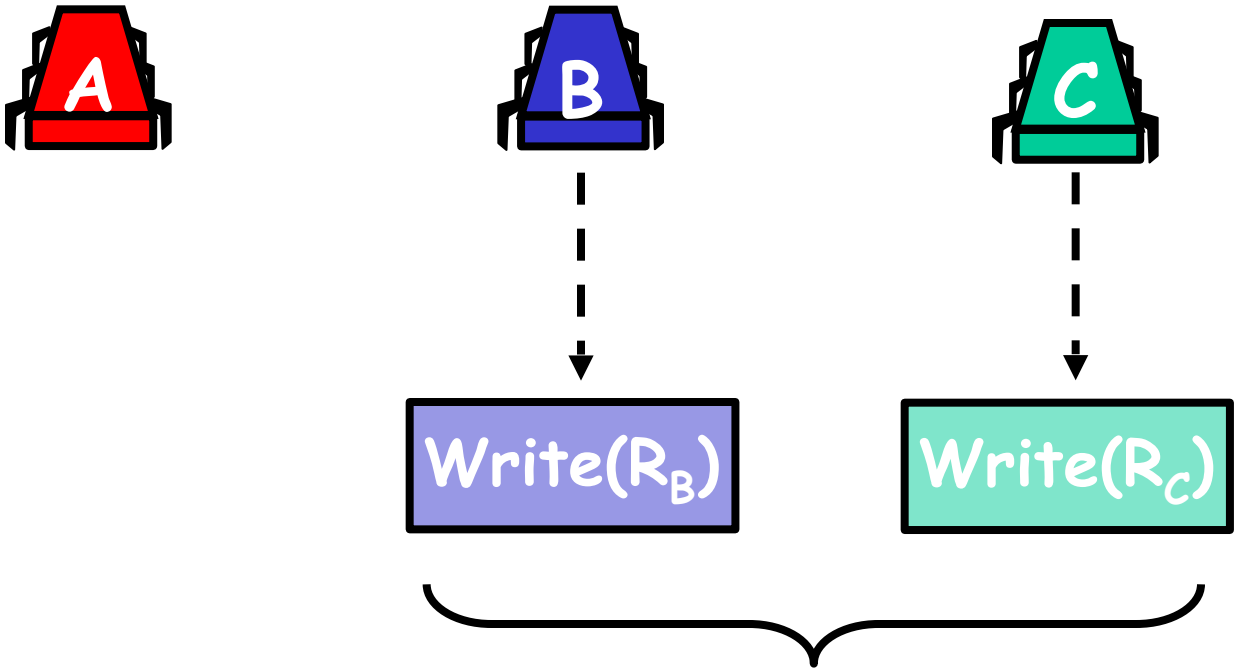


B Runs, first obliterating any trace of A, then also enters the critical section

Theorem

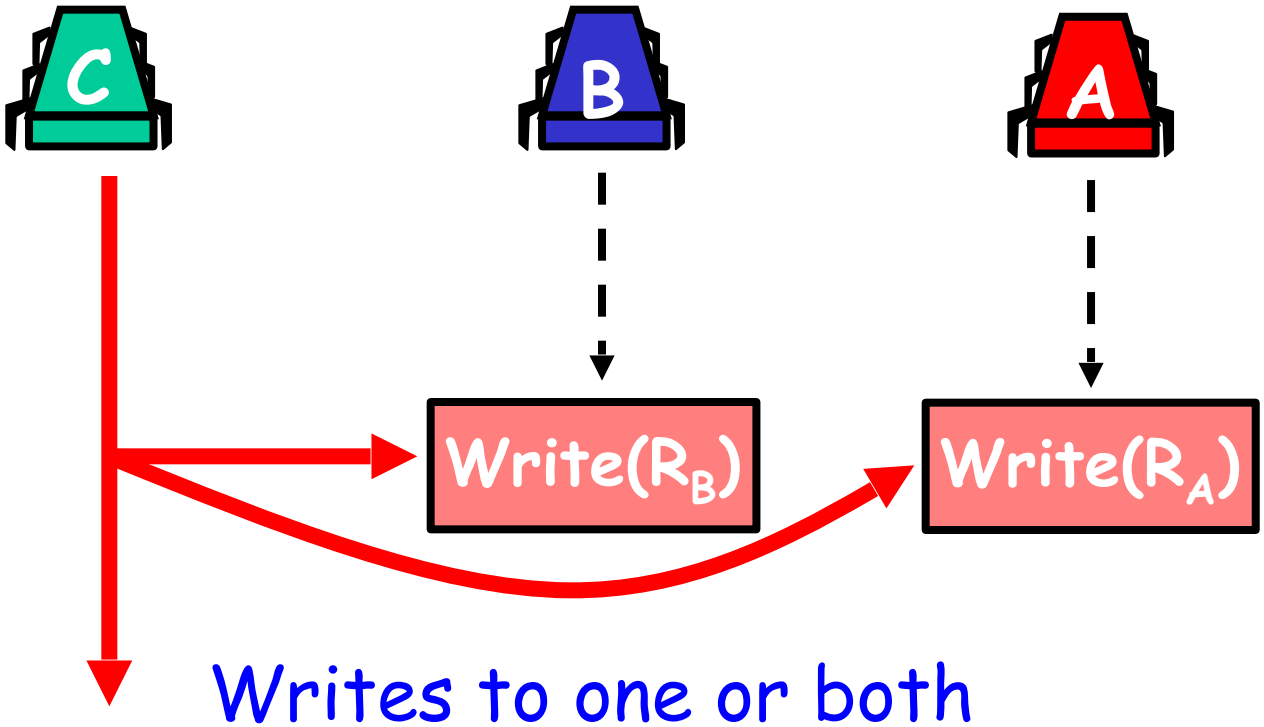
Deadlock-free mutual exclusion for 3 threads requires at least 3 multi-reader multi-writer registers

Proof: Assume Cover of 2



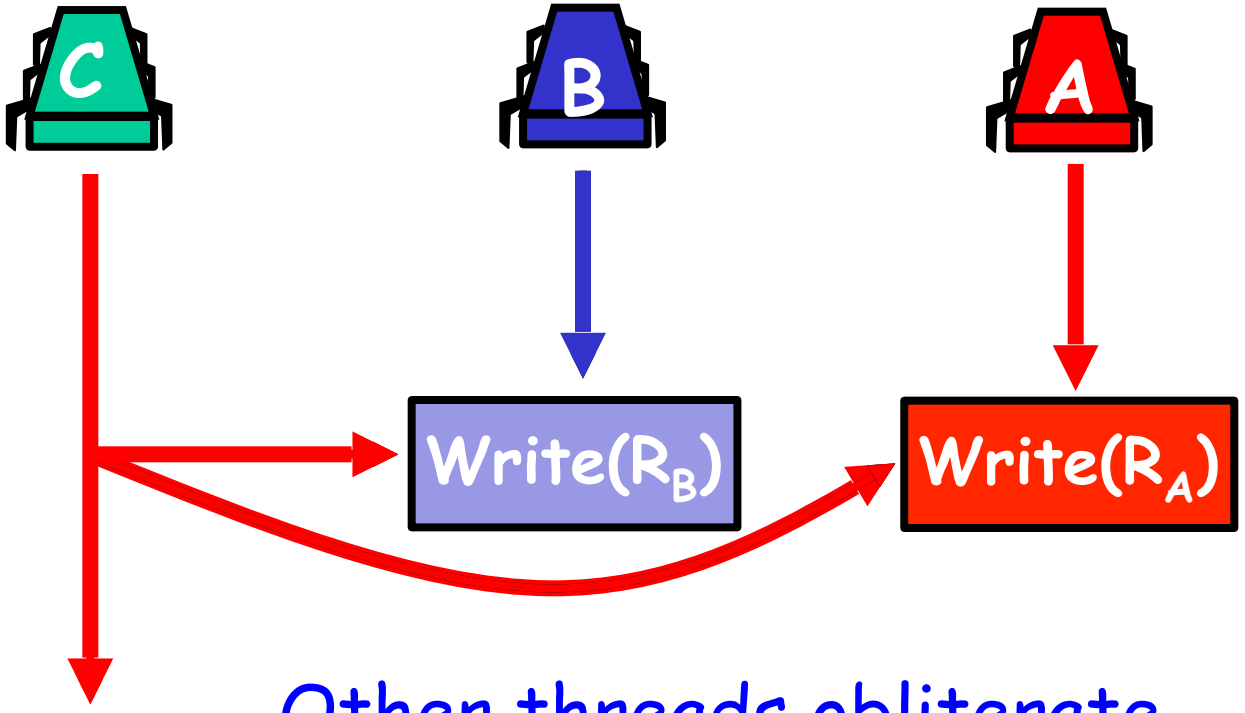
Only 2 registers

Run A Solo



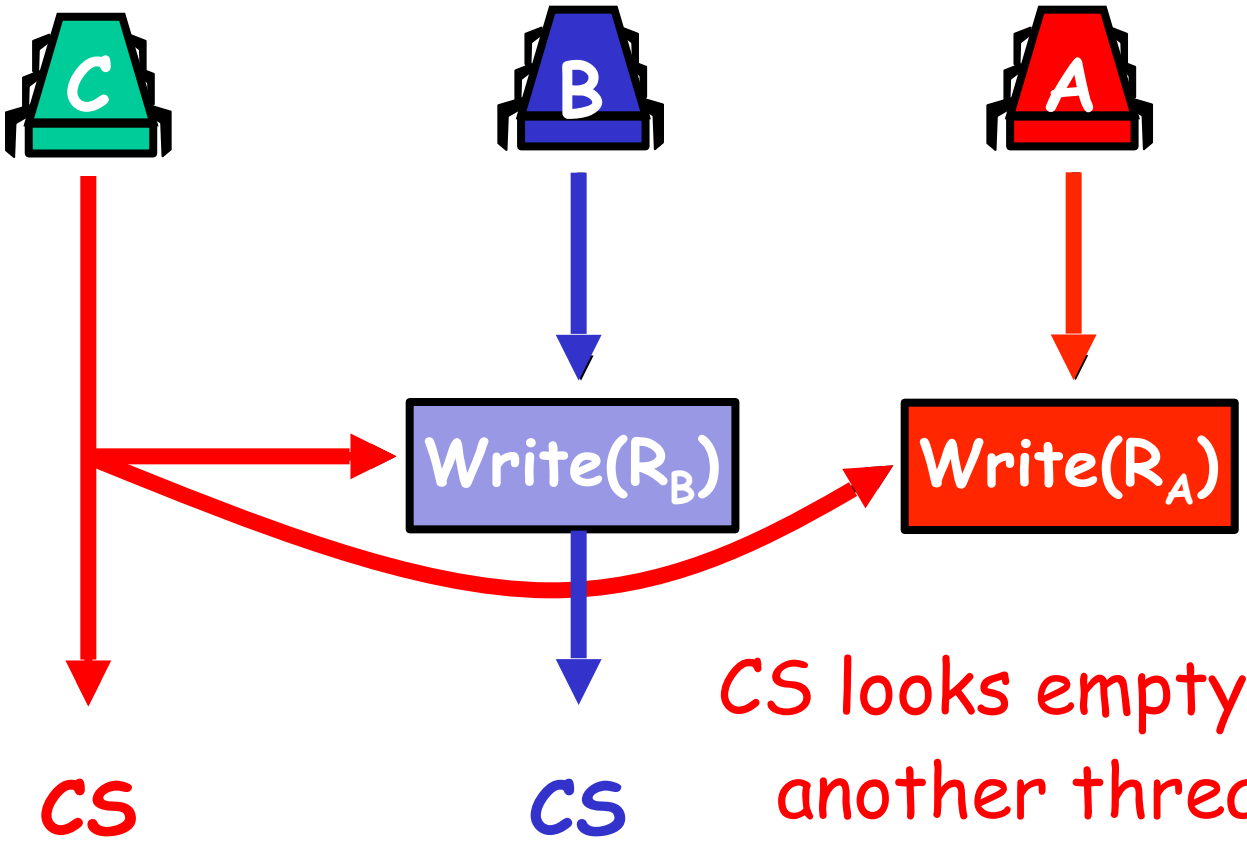
CS Writes to one or both registers, enters CS

Obliterate Traces of A



Other threads obliterate evidence that C entered CS

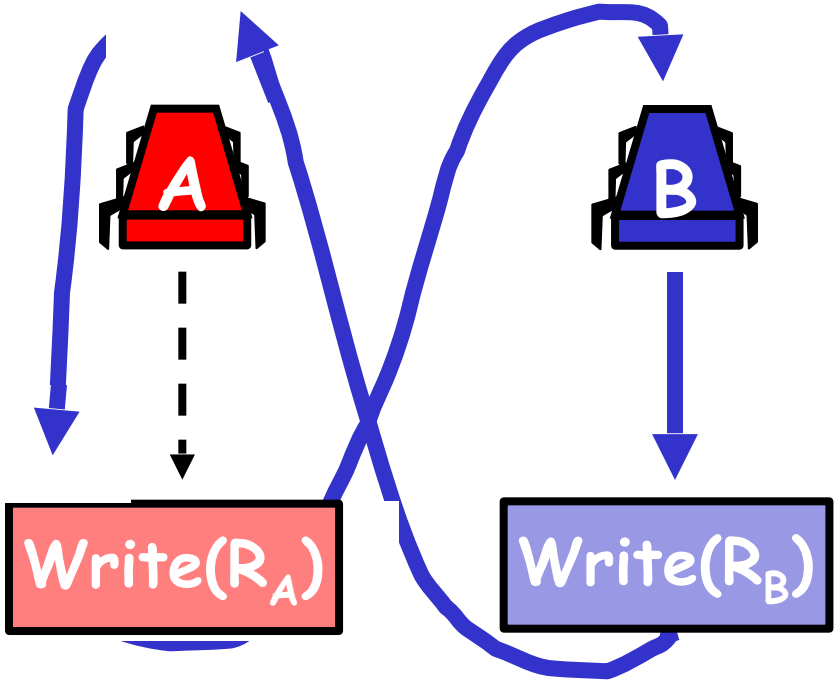
Mutual Exclusion Fails



Proof Strategy

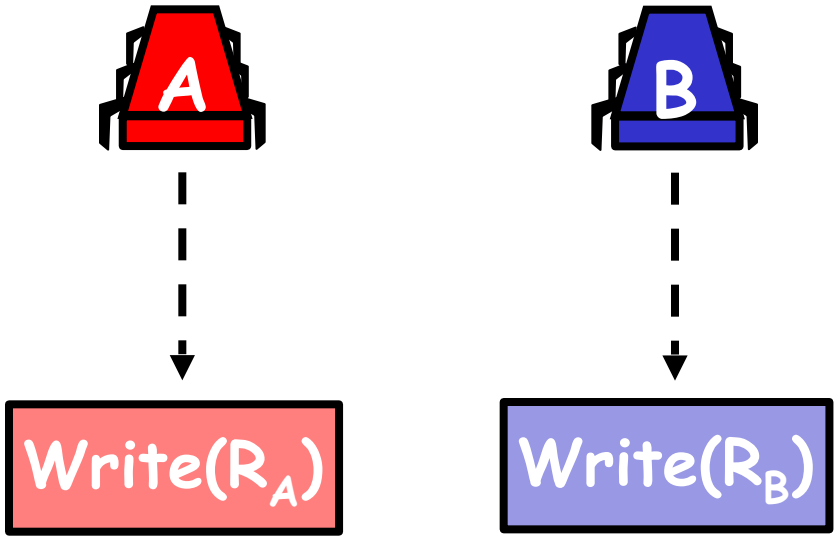
- Proved: a contradiction starting from a covering state for 2 registers
- Claim: a covering state for 2 registers is reachable from any state where CS is empty

Covering State for Two



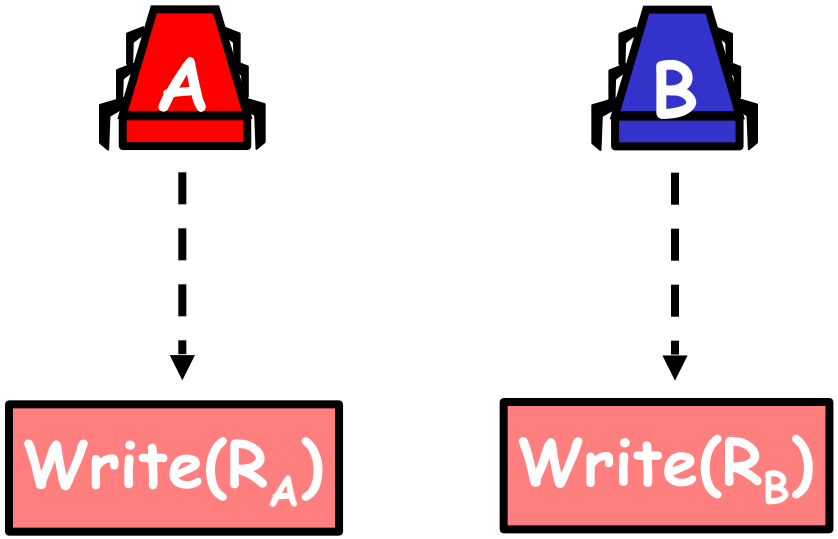
- If we run B through CS 3 times, B must return twice to cover some register, say R_B

Covering State for Two



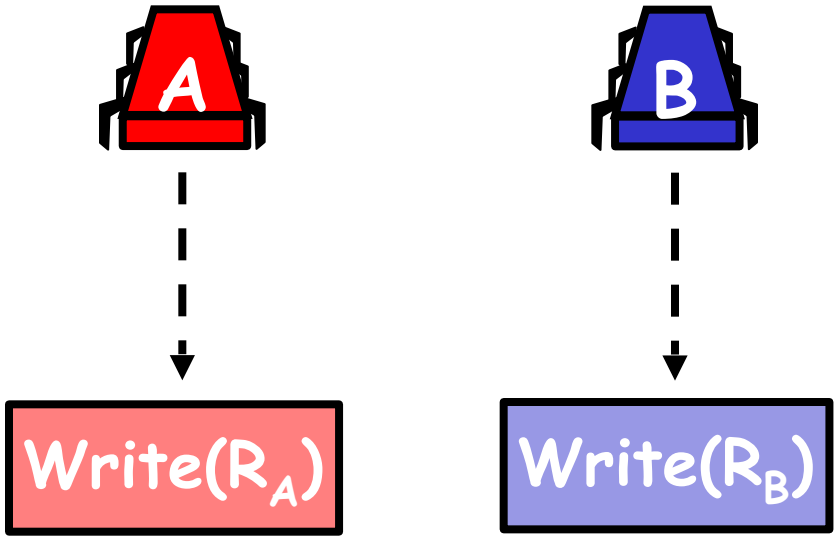
- Start with B covering register R_B for the 1st time
- Run A until it is about to write to uncovered R_A
- Are we done?

Covering State for Two



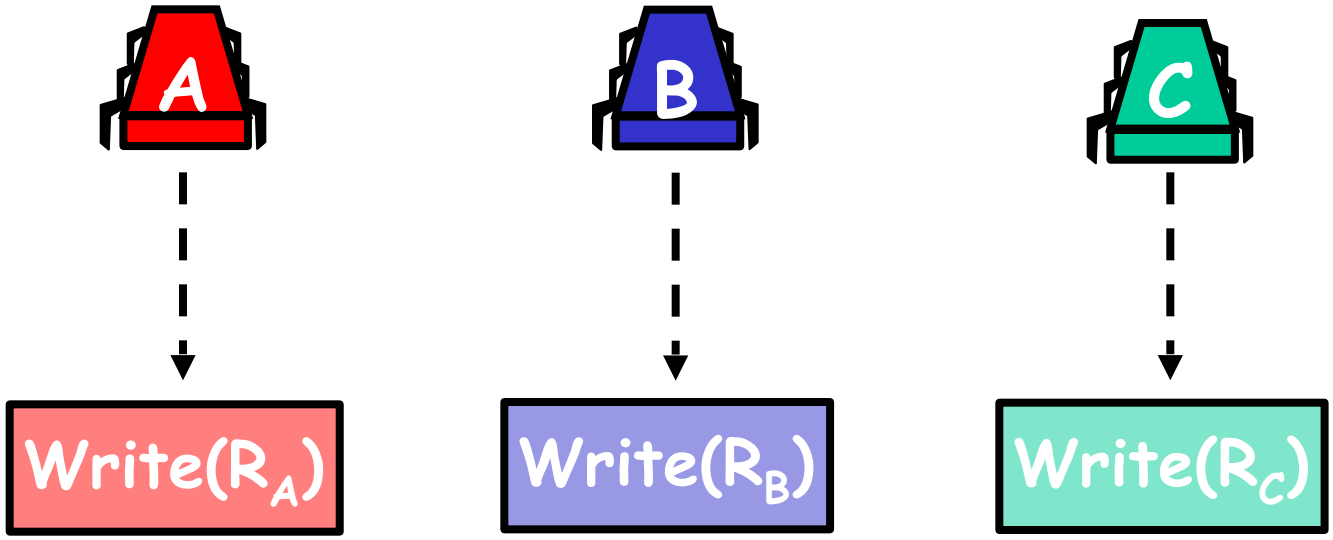
- NO! *A* could have written to R_B
- So *CS* no longer looks empty

Covering State for Two



- Run B obliterating traces of A in R_B
- Run B again until it is about to write to R_B
- Now we are done

Inductively We Can Show



- There is a covering state
  - Where k threads not in CS cover k distinct registers
  - Proof follows when k = N-1

Summary of Lecture

- In the 1960's many incorrect solutions to starvation-free mutual exclusion using RW-registers were published…
- Today we know how to solve FIFO N thread mutual exclusion using 2N RW-Registers

Summary of Lecture

- N RW-Registers inefficient
  - Because writes "cover" older writes
- Need stronger hardware operations
  - that do not have the "covering problem"
- In next lectures - understand what these operations are…

Rappel sur les Threads en java

Threads

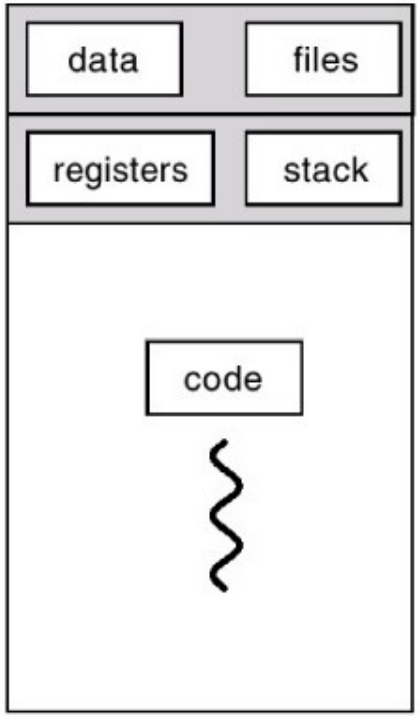
- threads: plusieurs activités qui coexistent et partagent des données
  - exemples:
    - pendant un chargement long faire autre chose
    - serveur répondre à des requêtes concurrentes
    - coopérer entre activités
    - multicoeur: plusieurs threads partagent le mulit-coeur
  - problème de l'accès aux ressources partagées
    - verrous
    - moniteur
    - synchronisation

Note

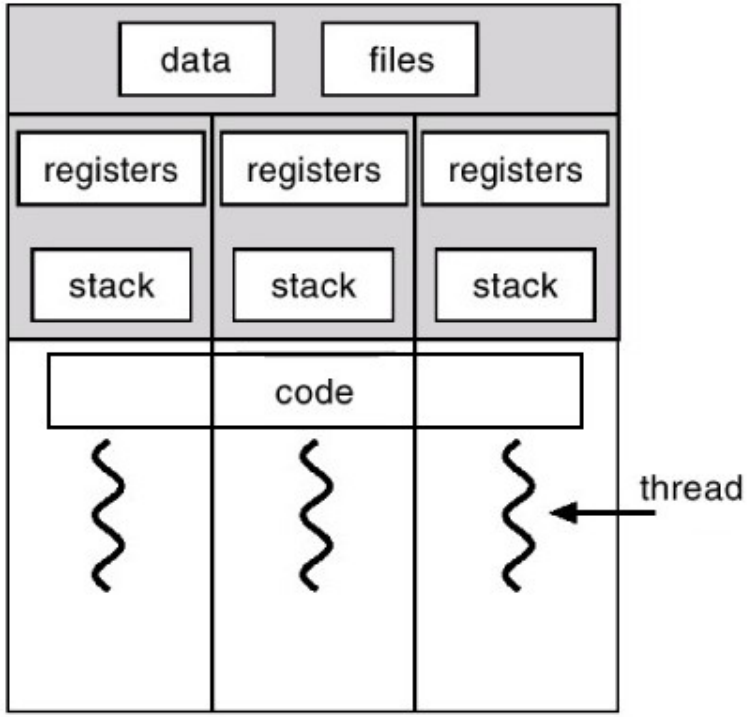
- A bas niveau
  - processus versus thread
    - processus (typiquement unix): « code » dans son propre espace mémoire, ses propres ressources + IPC pour se synchroniser et échanger avec les autres
    - thread (lightweight processes) « code » avec des resources partagées
    - chaque processus a au moins un thread, les threads partagent les ressources du processus dans lequel elles s 'exécutent

système…

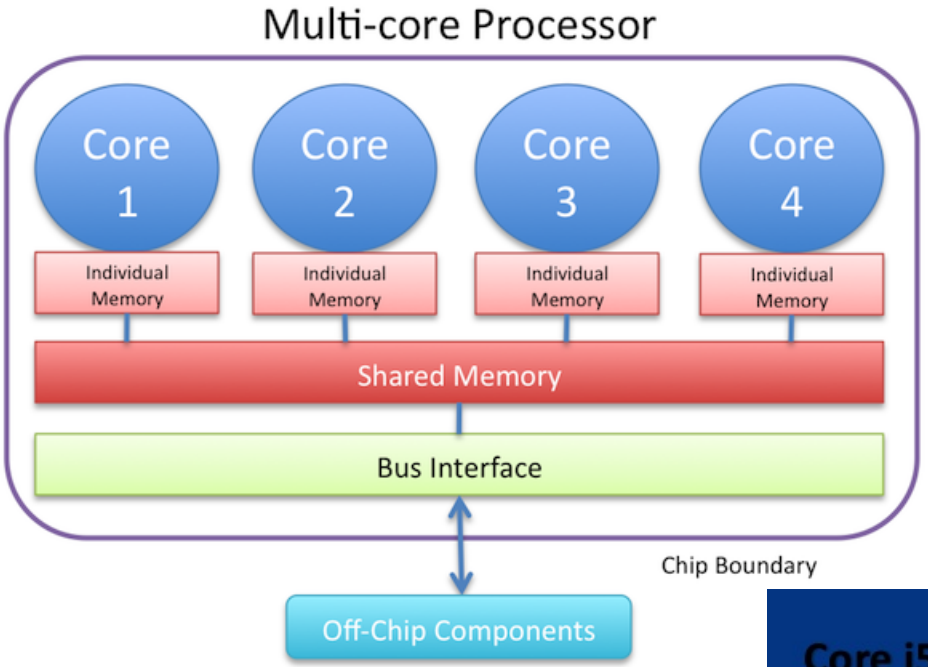
process

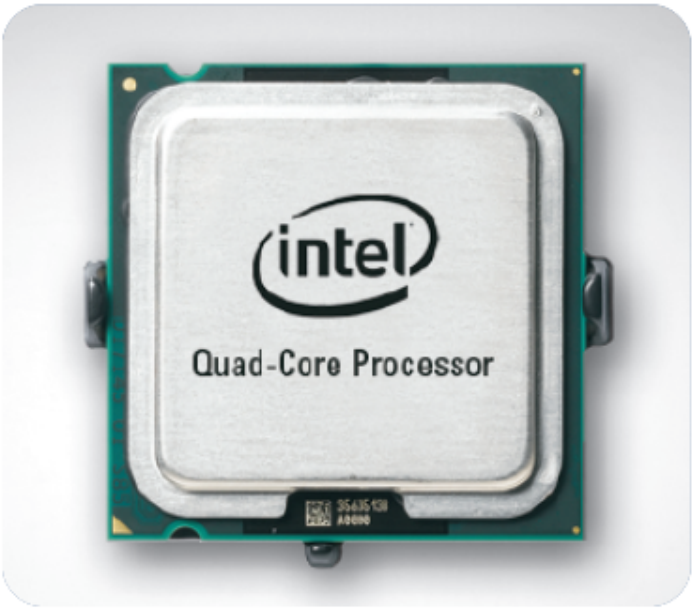


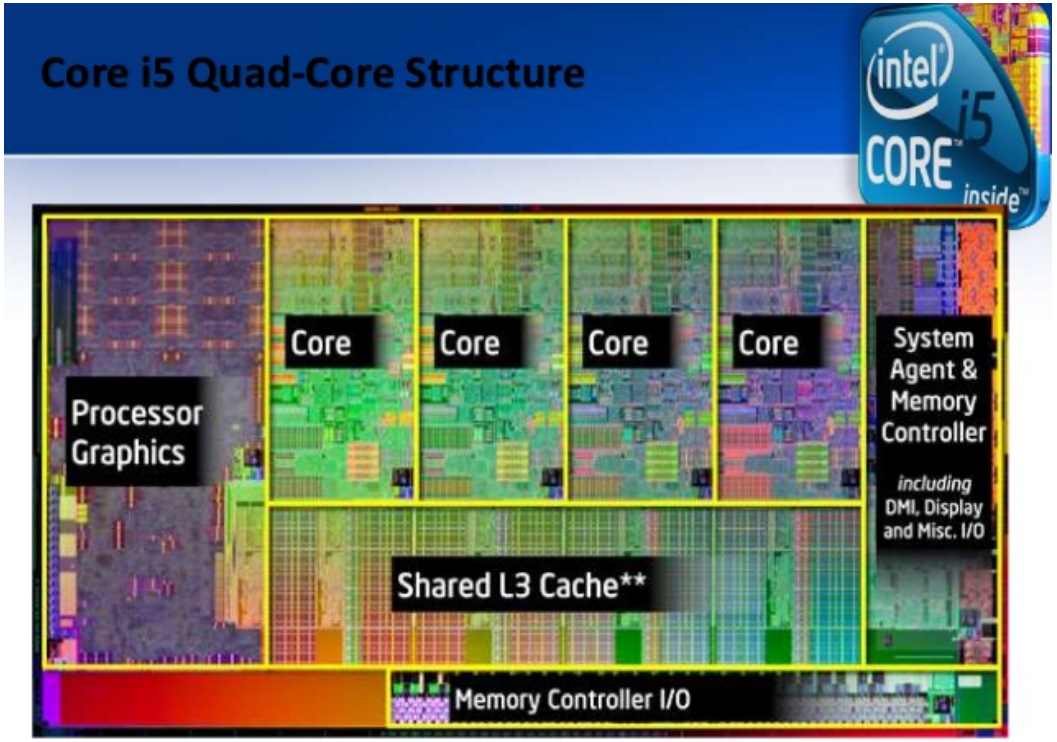
threads dans un process



threaded


Multi-core Processor
Core 1
Core 2
Core 3
Core 4
Individual Memory
Individual Memory
Individual Memory
Individual Memory
Shared Memory
Bus Interface
Chip Boundary
Off-Chip Components


intel
Quad-Core Processor


Core i5 Quad-Core Structure
intel
i5
CORE
inside
Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including DMI, Display and Misc. I/O
Shared L3 Cache**
Memory Controller I/O

Threads Java: Principes de base

- extension de la classe *Thread*
  - méthode *run* est le code qui sera exécuté.
  - la création d'un objet dont la super-classe est *Thread* crée le thread (mais ne la démarre pas)
  - la méthode *start* démarre la thread (et retourne immédiatement)
  - la méthode *join* permet d'attendre la fin du thread
  - les exécutions des threads sont asynchrones et concurrentes

Exemple

```
class ThreadAffiche extends Thread {
    private String mot;
    private int delay;
    public ThreadAffiche(String w, int duree) {
        mot = w;
        delay = duree;
    }
    public void run() {
        try {
            for (;;) {
                System.out.println(mot);
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

```
public static void main(String[] args) {
  new ThreadAffiche("PING", 1000).start();
  new ThreadAffiche("PONG", 3000).start();
  new ThreadAffiche(« splash!",1500).start();
 }
```

Alternative: Interface Runnable

- Une autre solution:
  - créer une classe qui implémente l'interface *Runnable* (cette interface contient la méthode *run*)
  - créer un *Thread* à partir du constructeur *Thread* avec un *Runnable* comme argument.

Exemple

```
class RunnableAffiche implements Runnable{
    private String mot;
    private int delay;
    public RunnableAffiche(String w,int duree){
        mot=w;
        delay=duree;
    }
    public void run(){
        try{
            for(;;){
                System.out.println(mot);
                Thread.sleep(delay);
            }
        }catch(InterruptedException e){
        }
    }
}
```

```
public static void main(String[] args) {
    Runnable ping=new RunnableAffiche("PING", 1000);
    Runnable pong=new RunnableAffiche("PONG", 500);
     new Thread(ping).start();
     new Thread(pong).start();
}
```

Thread

- création de l'objet : new ThreadAffiche
- démarrage du thread: méthode start
- un thread est lancé dans un autre thread (main thread) par la méthode start()
- exécution concurrente

attendre la fin du thread: join()

```
Thread t = new Thread() {
    public void run() {
        try {
            Thread.sleep(6000);
        } catch (InterruptedException ex) {
        }
        System.out.println("thread terminée");
    }
};
t.start();
try {
    t.join();
} catch (InterruptedException ex) {
}
System.out.println("FINI");
```

Interruption

La méthode interrupt() permet d'interrompre un thread en cours d'exécution d'une méthode sleep(), join() ou wait()

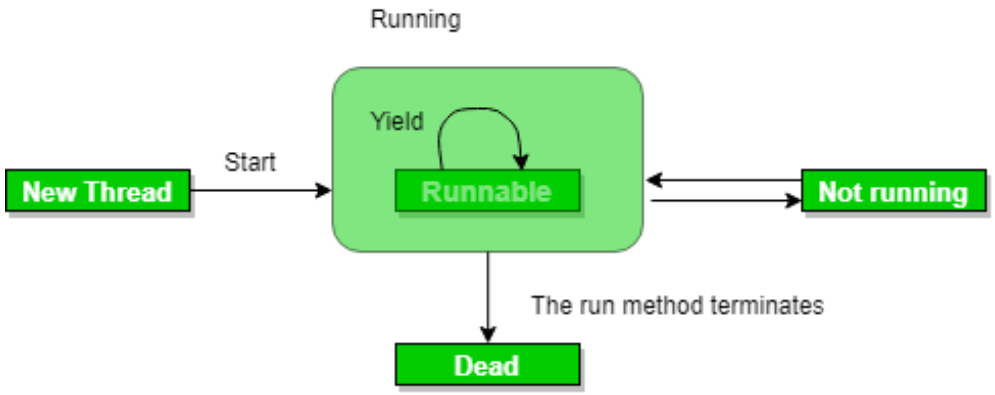
interruption

```
ipublic class EssaiThread  extends Thread {

   public void run() {

         try {
            Thread.sleep(1000);
            System.out.println("apres avoir dormi  ");
        } catch (InterruptedException ex) {
            System.out.println("Thread.sleep interrompue");
          //   java.lang.System.exit(0);
         }
         System.out.println(" a la fin  ");

   }

   public static void main(String[] args) {
      Thread t1 = new EssaiThread();
      t1.start();
      Thread t2 = new EssaiThread();
      t2.start();
      try {
         Thread.sleep(500);
         t1.interrupt();
         Thread.sleep(100);
         t2.interrupt();

      } catch (InterruptedException ex) {
          System.out.println("Thread.sleep interrompue");
      }
   }
}
```

Class Thread

quelques méthodes:

- yield()
- sleep()
- join()
- interrupt()/ isInterrupted()
- setPriority()/ getPriority()
- setName()/ getName()
- getId()
- setDaemon()/ isDaemon()



Partage…

- les threads s'exécutent concurremment et peuvent accéder concurremment aux objets dans leur portée
- quel est l'effet d'une modification d'une variable partagée entre plusieurs threads?
- (en général la question est de l'atomicité de l'accès aux variables partagées)

```
public class InVisible {

   public static boolean fait = false;
   public static int n;

   public static class Lecteur extends Thread {

       public void run() {
           while (!fait);
           System.out.println(n);

       }
   }

  public static void main(String[] args) throws InterruptedException {
       new Lecteur().start();
      // Thread.sleep(100);
       n = 150;
        fait = true;
     System.out.println("fait");

   }
}
```

Ce programme peut donner des résultats différents:

- ne pas terminer
- afficher 150

(la thread main affiche« fait »)

Lecture-écriture

- « atomicité » tout se passe comme si l'opération est d'un seul tenant sans interruption
  - lire ou écrire une « valeur simple » est indivisible
  - lire ou écrire une valeur correspondant à plusieurs mots mémoires n'est pas forcément indivisible
- optimisation de code: certaines opérations peuvent être inversées (ou supprimées): volatile

problème de la cohérence de la mémoire

En java

- **atomicité des lectures écritures:**
  - lecture et écriture pour les variables des types primitifs sauf long et double
  - lecture et écriture pour les variables déclarées comme *volatile*
  - lecture et écriture pour les references

- **happens before** (Spécifié dans le chapitre 17 du)
  - Each action in a thread happens-before every action in that thread that comes later in the program's order.
  - An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.
  - A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
  - A call to start on a thread happens-before any action in the started thread.
  - All actions in a thread happen-before any other thread successfully returns from a join on that thread.
  - Transitivity of the relation « happens before"
- (les objets **immuables** (immutable) -qui, une fois créés ne peuvent être modifiés permettent aussi d'assurer la cohérence mémoire)

```
public class InVisible {

   public static boolean fait = false;
   public static int n;
   public static Integer I=3;
   public static class Lecteur extends Thread {

       public void run() {
          while (!fait) { synchronized(I){} ;}
          System.out.println(n);

       }
   }

  public static void main(String[] args) throws InterruptedException {
       new Lecteur().start();
      // Thread.sleep(100);
       n = 150;
        synchronized(I){
       fait = true;
       }
     System.out.println("fait");

   }
}
```

Partage…

- Avec volatile on assure l'atomicité sur l'accès aux variables (simples) partagées
- Mais souvent cela ne suffit pas toujours, on veut en général avoir une atomicité pour plusieurs opérations sur ces variables:
- Avec des lectures et des écritures atomiques
  - il faut contrôler l'accès:
    - thread un lit une variable (R1) puis modifie cette variable (W1)
    - thread deux lit la même variable (R2) puis la modifie (W2)
    - R1-R2-W2-W1
    - R1-W1-R2-W2 résultat différent!

Exemple

```
class X{
    int val;
}
class Concur extends Thread{
    X x;
    int i;
    String nom;
    public Concur(String st, X x){
        nom=st;
        this.x=x;
    }
    public void run(){
        i=x.val;
        System.out.println("thread:"+nom+" valeur x="+i);
        try{
            Thread.sleep(10);
        }catch(Exception e){}
        x.val=i+1;
        System.out.println("thread:"+nom+" valeur x="+x.val);
    }
}
```

```
public static void main(String[] args) {
        X x=new X();
        Thread un=new Concur("un",x);
        Thread deux=new Concur("deux",x);
        un.start(); deux.start();
        try{
            un.join();
            deux.join();
        }catch (InterruptedException e){}
        System.out.println("X="+x.val);
    }
```

thread:deux valeur x=0
thread:un valeur x=0
thread:un valeur x=1
thread:deux valeur x=1
X=1

Deuxième exemple

```
class Y{
    int val=0;
    public int increment(){
        int tmp=val;
        tmp++;
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
class Concur1 extends Thread{
    Y y;
    String nom;
    public Concur1(String st, Y y){
        nom=st;
        this.y=y;
    }
    public void run(){
        System.out.println("thread:"+nom+" valeur="+y.increment());
    }
}
```

```
public static void main(String[] args) {
        Y y=new Y();
        Thread un=new Concur1("un",y);
        Thread deux=new Concur1("deux",y);
        un.start(); deux.start();
        try{
            un.join();
            deux.join();
        }catch (InterruptedException e){}
        System.out.println("Y="+y.getVal());
    }
----------
```

- thread:un valeur=1
- thread:deux valeur=1
- Y=1

compteur

```
class Counter1 {
    int n;
    int getAdd() {
   Thread.yield();
        return n++;
    }
}
```

```
class Counter {

    int n;

    int getAdd() {
        int temp = n;
        Thread.yield();
        n = temp + 1;
        return temp;
    }
}
```

```
new Thread() {
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(c.getAdd());
        }
    }
}.start();
new Thread() {
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(c.getAdd());
        }
    }
}.start();
System.out.println(c.getAdd());
```

Utiliser des verrous
Exclusion mutuelle

Verrous

- à chaque objet est associé un verrou (lock)
- synchronized(expr) {instructions}
  - expr doit s'évaluer comme une référence à un objet
  - verrou sur cet objet pour la durée de l'exécution de instructions
  - déclarer les méthodes comme synchronized: le thread obtient le verrou sur la méthode et le relâche quand la méthode se termine.
  - (les verrous sont ré-entrants: thread qui demande un verrou qu'il a déjà … l'obtient)

synchronized

```
class Concur extends Thread{
    X x;
    int i;
    String nom;
    public Concur(String st, X x){
        nom=st;
        this.x=x;
    }
    public void run(){
        synchronized(x){
            i=x.val;
            System.out.println("thread:"+nom+" valeur x="+i);
            try{
                Thread.sleep(10);
            }catch(Exception e){}
            x.val=i+1;
            System.out.println("thread:"+nom+" valeur x="+x.val);
        }
    }
}
```

```
class X{
    int val;
}
```

```
class Y{
    int val=0;
    public synchronized int increment(){
        int tmp=val;
        tmp++;
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
```

Méthode synchronisée

```
class Y{
    int val=0;
    public synchronized int increment(){
        int tmp=val;
        tmp++;
        try{
            Thread.currentThread().sleep(100);
        }catch(Exception e){}
        val=tmp;
        return(tmp);
    }
    int getVal(){return val;}
}
```

```
class Counter3 {
    private int n;
    synchronized int getAdd() {
        return n++;
    }
}
….
Counter3 c = new Counter3();
 new Thread() {
          public void run() {
              for (int i = 0; i < 100; i++) {
                  System.out.println(c.getAdd());
              }
          }
       }.start();
       new Thread() {
          public void run() {
              for (int i = 0; i < 100; i++) {
                  System.out.println(c.getAdd());
              }
          }
       }.start();
       System.out.println(c.getAdd());
```

Mais…

- la synchronisation par des verrous peut entraîner un blocage:
  - la thread un (XA) pose un verrou sur l'objet A et (YB) demande un verrou sur l'objet B
  - la thread deux (XB) pose un verrou sur l'objet B et (YA) demande un verrou sur l'objet A
  - si XA –XB : ni YA ni YB ne peuvent être satisfaites -> blocage
- (pour une méthode synchronisée, le verrou concerne l'objet globalement et pas seulement la méthode)

Synchronisation

- attendre qu'une condition soit réalisée pour poursuivre.

```
class Partage{
    private volatile boolean fait=false;
    Integer data=0;
    void ecrire() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {

        }
        data=153; fait=true;
    }
    void lire(){
        while (!fait){}
        System.out.println(data);
    }
}
```

```
final Partage partage = new Partage();

new Thread(new Runnable(){
    public void run(){partage.ecrire();
    }
 }).start();
 new Thread(new Runnable(){
    public void run(){partage.lire();
    }
}).start();
```

Synchronisation…

- wait, notifyAll, notify
  - attendre une condition / notifier le changement de condition:

```
synchronized void fairesurcondition(){
        while(!condition){
                wait();
        }
        faire ce qu'il faut quand la condition est vraie
----------------
synchronized void changercondition(){
        … changer quelque chose concernant la condition
        notifyAll(); // ou notify()
}
```

Class Object

void	() Wakes up a single thread that is waiting on this object's monitor.
void	() Wakes up all threads that are waiting on this object's monitor.
void	() Causes the current thread to wait until another thread invokes the method or the method for this object.
void	(long timeout) Causes the current thread to wait until either another thread invokes the method or the method for this object, or a specified amount of time has elapsed.
void	(long timeout, int nanos) Causes the current thread to wait until another thread invokes the method or the method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

mémoire locale des threads…

Dans la suite on utilisera ce programme pour nommer les threads

```
1  public class ThreadID {
2    private static volatile int nextID = 0;
3    private static class ThreadLocalID extends ThreadLocal<Integer> {
4      protected synchronized Integer initialValue() {
5        return nextID++;
6      }
7    }
8    private static ThreadLocalID threadID = new ThreadLocalID();
9    public static int get() {
10     return threadID.get();
11   }
12   public static void set(int index) {
13     threadID.set(index);
14 }
```

ThreadLocal: permet d'avoir des variables lues et écrites par la *même* thread. Si deux threads partagent le même code chacune aura un copie différente de la variable

- **ThreadLocal<T>**

- **Constructor and Description**
- **ThreadLocal**() Creates a thread local variable.

- **Method and Description**

get()Returns the value in the current thread's copy of this thread-local variable.

protected **T** **initialValue**()Returns the current thread's "initial value" for this thread-local variable.

void **remove**()Removes the current thread's value for this thread-local variable.

void **set**(**T** value)Sets the current thread's copy of this thread-local variable to the specified value.

Retour sur l’exclusion mutuelle

Interface Lock (simplifié)

`void lock()`
Acquires the lock.

`void unlock()`
Releases the lock.

Class ReentrantLock

- Implémente **`Lock`**
- Il peut être acquis plusieurs fois par la même thread
- Le constructeur de cette classe peut avoir un paramètre d'équité. Positionner à **`true`** il indique qu'en cas d'attente sur le verrou c'est la thread qui est la première arrivée qui est servie.
- Méthodes pour connaitre les threads en attente sur le lock ou sur la condition associée