**CPU Cache**

* Un calculator are propria sa memorie(RAM) unde datele si codul sunt stocate
* Totodata, are si un prcesor care executa codul si lucreaza cu datele.
* Un core citeste datele din memorie(RAM), le modifica dupa cum e necesar, si le rescrie inapoi in memorie
* Totusi, daca el ar lucra cu aceste date folosind memoria calculatorului, acest proces ar fi foarte lent, asa cum memora calculatorului e foarte lenta
* Iata de ce, fiecare Core are un Cache memory, care e foarte rapida
* In cache memory, CPU copie aceste date si cod din Registru, le executa, le modifica tot la el in cache, le scrie in registre, si apoi le rescrie in memoria calculatorului
* In timpul in care CPU modifica datele la el in cache memory, este evident ca aceste date nu sunt modificate si in memoria la pc, deci el inca urmeaza a fi rescrise acolo dupa ce vor fi repuse in registre
* Totusi, anume aici si apare problema. Un thread va face ca datele preluate din RAM sa fie puse in cache la CPU din registre. Cand doua CPU au in Cache o copie a unei date sau instructiuni din aceeasi zona de memorie, daca unul din ele o modifica, se va asigura ca si celalalt sa primeasca noua valoare in Cache
* Dar, nu mereu Cache la un alt core primeste aceasta noua valoare. Uneori un core poate considera ca nu e necesar sa anunte alte cores deaspre faptul ca valoarea ce ambele o acceseaza a fost modificata
* De ex:

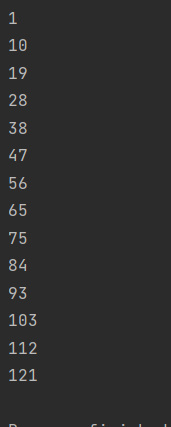
public class Test2 extends Thread{  
 private boolean ready = false;  
  
 @Override  
 public void run() {  
 while (!ready) {  
 // System.out.println("Core se va asigura sa dea update la cache ");  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Test2 test2 = new Test2();  
 test2.start();  
 Thread.*sleep*(500);  
 test2.ready = true;  
 System.*out*.println(test2.ready);  
  
 }  
}

Aici, test2 are valarea lui redy egala cu false, dar ideea e ca main thread modifica valoarea in true, dar, test2 considera ca nu e necesar de a face update la cache, si in cache vechi al sau , ready inca e false, si dea asta bucla sa while e executata la nesfarsit.

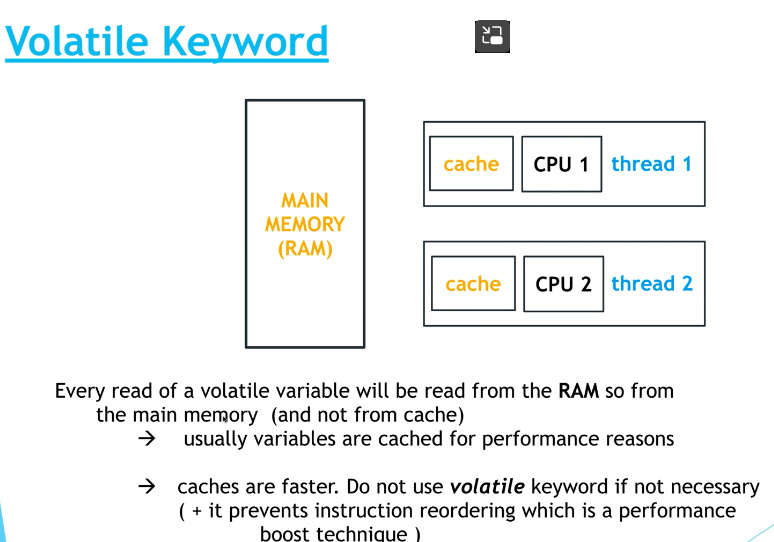
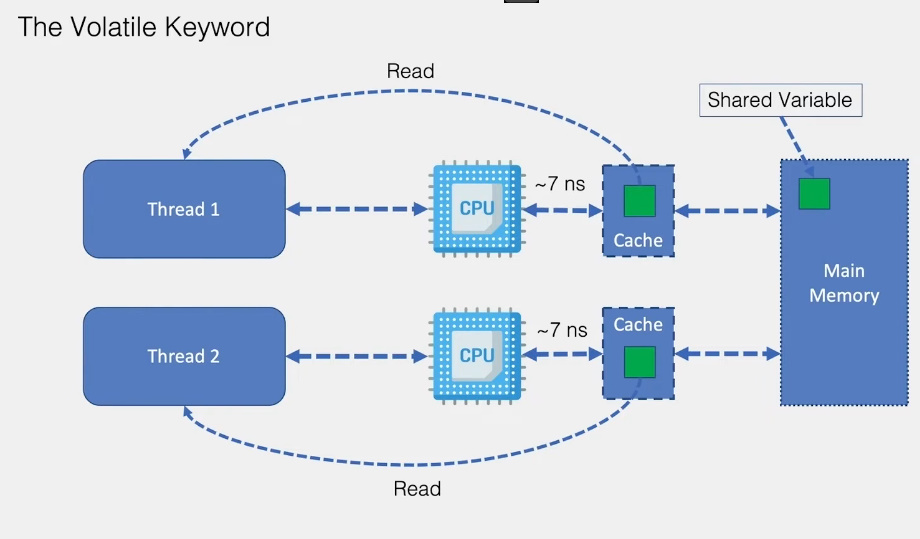
* Daca totusi eliminam comentariul din instructiunea la while, CPU va decide ca trebuie sa dea update la cache pe baza la ce e in Cache la celalalt CPU, si bucla se va opri dupa ce ready va fi true. In marea parte din cazuri, CPU da update la cache foarte **re**g**ulat**, desi sunt si exceptii cand nu da.
* **E imposibil de a prezice cand un CPU va da update la cache si in ce conditii,de aceea nici exemple nu pot fi aduse, de aceea trebuie sa facem volatile variabilele care stim la si**g**ur ca nu trebuie stocate in cache!!!Daca punem la ready volatile, bucla w**h**ile nu va fi niciodata infinita!**
* **Dar chiar volatile nu garanteaza ca 2 threaduri nu vor face concomitent o modificare asupra la aceeasi variabila**

**Exemplul de jos arata ca t**hread da update la cache, si asa desi 1 secunda a fost adormit, el nu foloseste datele din cache vechi, ci din cache nou(updated)

public class Test2 extends Thread{  
 private int ready = 0;  
  
 @Override  
 public void run() {  
 while (true) {  
 System.*out*.println(ready);  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 Test2 test2 = new Test2();  
 test2.start();  
 int i = 0;  
 while (true){  
  
 test2.ready = ++i;  
 Thread.*sleep*(100);  
 }  
  
 }  
}

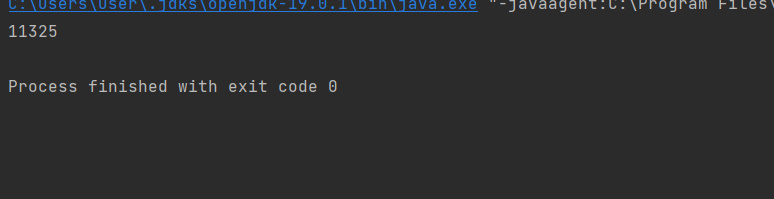
****

**Volatile**

* 
* 
* Deci, fiecare thread e executat de un core, si e evident ca mai multe threaduri pot fi executate fie de acelasi core, fie de diferite
* CPU are cache, deoarece e mult mai rapid sa prelucreze datele in cache decat in memoria la pc
* **Volatile** – acest keyword va face ca datele sa nu fie copiate in cache la CPU, dar sa fie direct citite din RAM de catre CPU cand va face ceva. Deci volatile inseamna ca datele nu trebuie sa fie cached in CPU ci accesate si accesate direct din RAM!!!
* **Deci, datele volatile nu vor fi stocate in Cache nici cand sunt citite, nici dupa ce sunt executate operatiile asupra lor si se obtine rezultatul final**. Rezultatul final, dar si datele initiale, mereu va fi pus direct in RAM Memory si nu in cache nicidecum.
* Acest lucru face ca operatiile sa fie executate mai incet, de aceea Volatile trebuie folosit strict doar unde e necesar
* **Volatile poate fi folosit doar pentru variabile, nu si clase sau metode!!!**
* **Volatile este doar pentru operatii atomice, ca x=10, nu si x++!!**

**Daca am folosi x++, pai CPU** iar ar stoca niste date in cache ca sa faca operatia de adunare, si tot la aceeasi problema ajunem.Dar daca avem o operatie atomica, nu e nevoie de nimic in cache caci nu are de facut calcule.

* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   Run run = new Run();  
   Thread thread1 = new Thread(run);  
   Thread thread2 = new Thread(run);  
    
   thread1.start();  
   thread2.start();  
   thread1.join();  
   thread2.join();  
    
   System.*out*.println(Run.*a*);  
    
   }  
  }  
    
  class Run implements Runnable {  
   public static int *a* = 0;  
    
    
   @Override  
   public void run() {  
   for(int i = 0;i<10000;i++)  
   ++*a*;  
   }  
  }



iar avem aceeasi problema, adica asa cum noi nu efectuam o operatie atomica, fiecare CPU care executa threadurile va fi nevoie sa stocheze in cache nu propriu zis valoarea variabilei atomice, dar rezultatul operatiei, si iar se intampla aceeasi problema.

**Stopping a thread**

* metoda .stop() e depreciata si periculoasa
* Cea mai buna solutie este de a crea in clasa o variabila de tip boolean, ceva de genul:

boolean terminated = false;

Si apoi in metoda run sa facem un while:

void run() {

while(!terminated) { }

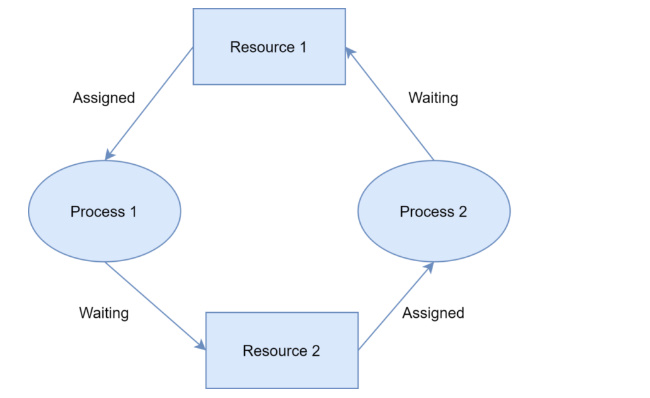
}

* Asa, cand vom vrea sa oprim un thread, pur si simplu vom seta variabila cu true si while se va opri si deci si thread tot

**DeadLock and LiveLock**

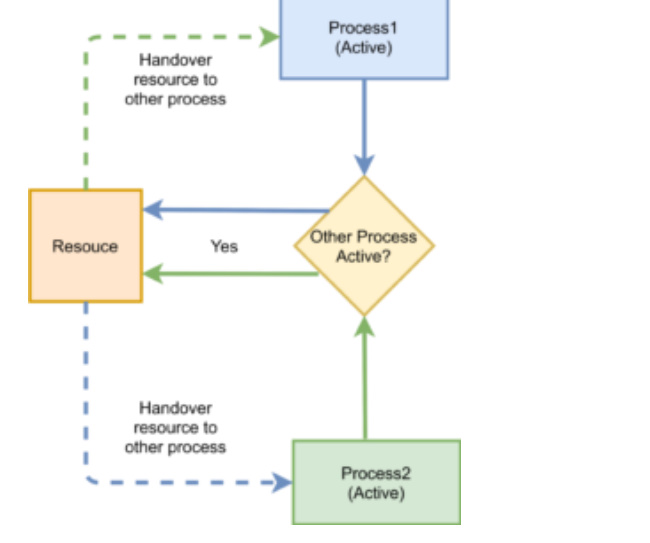
* **DeadLock** – situatie in care 2 sau mai multe threaduri/procese detin cate o resursa, si fiecare vrea sa acceseze resursa celuilalt, dar nu poate deoarece resursele sunt blocate si nici unul nu renunta la ele
* **In DeadLock procesele sunt mereu in waiting state!!Primul vede ca nu poate accesa resursa la al 2 si intra in waiting state ca crede ca primul inca nu a terminat si al doilea vede ca nu poate la 1 si tot intra in waiting state ca sa il lase pe primul sa termina si nu mai ies din waiting state deoarece nici unul nu termina.**
* Exemple:
* In baza de date:

Un proces actualizeaza linia 1 si apoi 2, iar alt proces tot in acest timp actualizeaza linia 2 si apoi 1



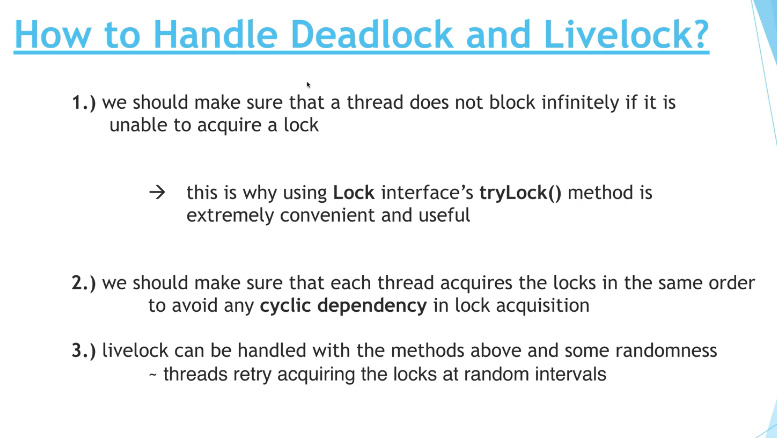
process1 tine resursa1 si are nevoie de resursa 2, dar process2 tine resursa 2 si are nevoie de resursa 1. Nici unul dintre ele nu poate accesa cealalta resursa pana unul dintre ele nu renunta la ea, dar nici unul nu poate renunta la ea pana cealalta nu e disponibila, si asta nu se intampla/

* **LiveLock** – se intampla atunci cand fiecare thread lasa executia pe mana celuilalt. Adica, fiecare lasa resursele pentru celalalt, si asa nici unul nu mai pun mana pe resurse(tot ca deadlock).
* **In LiveLock, procesele/threadurile nu sunt in waiting state, ci ruleaza concurent, incercand sa tot acceseze resursa,si asa nu fac niciun progres, dar nici nu intra in waiting state caci doar ele nu ruleaza ceva, ci urmeaza sa ruleze**
* De ex: 2 oameni vor sa treaca printr-un coridor foarte ingust. Prima persoana se da in stanga,ca a 2 sa treaca si a doua in dreapta, ca prima sa treaca, si asa nici una nu trece. Apoi prima se da in dreapta si deja a 2 in stanga, si iar nu pot trece.



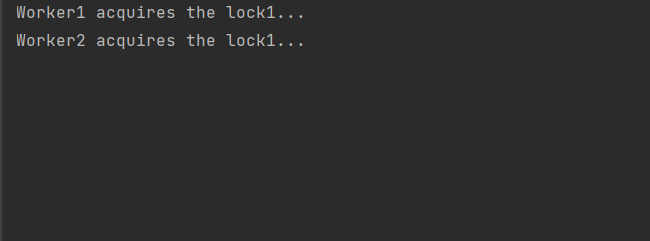
proces1 si proces2 vor sa acceseze aceeasi resursa in acelasi timp, insa proces1 il lasa pe proces2, si proces2 pe proces1, si asa nici unul nu o acceseaza si incearca sa o acceseze iar si iar.

**LiveLock si DeadLock sunt identice ca principiu, unica diferenta fiind ca la DeadLock threadurile sunt in waiting state pentru vesnicie, la livelock nu sunt in waiting si tot incearca sa accese resursa.**

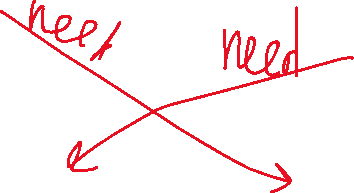


**DeadLock exemple:**

import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class Deadlock {  
  
 private Lock lock1 = new ReentrantLock(true);  
 private Lock lock2 = new ReentrantLock(true);  
   
 public static void main(String[] args) {  
   
 Deadlock deadlock = new Deadlock();  
   
 new Thread(deadlock::worker1, "worker1").start();  
 new Thread(deadlock::worker2, "worker1").start();  
 }  
   
 public void worker1() {  
 lock1.lock();  
 System.*out*.println("Worker1 acquires the lock1...");  
   
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
   
 lock2.lock();  
 System.*out*.println("Worker1 acquires the lock2...");  
   
 lock1.unlock();  
 lock2.unlock();  
 System.*out*.println("Worker 1 finished");  
  
 }  
   
 public void worker2() {  
 lock2.lock();  
 System.*out*.println("Worker2 acquires the lock1...");  
   
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
   
 lock1.lock();  
 System.*out*.println("Worker2 acquires the lock2...");  
   
 lock1.unlock();  
 lock2.unlock();  
 System.*out*.println("Worker 2 finished");  
  
 }  
}



Executia nu se termina



* Deci, avem 2 obiecte de tip ReentrantLock, asa dar 2 monitor locks(vezi in documentul (2))
* Acum, thread1 acceseaza worker1 si blocheaza monitor lock1, apoi intra in somn. Apoi, thread2 acceseaza worker2 si blocheaza monitor lock2 si intra in somn
* Dupa asta thread1 iese din somn, si vrea sa dea lock in lock2, dar deja thread2 a dat lock la lock2 si deci thread1 va astepta ca thread2 sa termine si sa elibereze lock2
* Acum, thread2 va vrea sa blocheze lock1, dar lock1 deja e blocata de thread2, si nu poate, si iar asteapta sa fie eliberata de thread1.
* Problema e ca thread2 asteapta ca thread1 sa elibereze lock1 si thread1 asteapta ca thread2 sa elibereze lock2 si ambele au intrat in waitin state!

Solutii pot fi mai multe, dar iata una de ex:

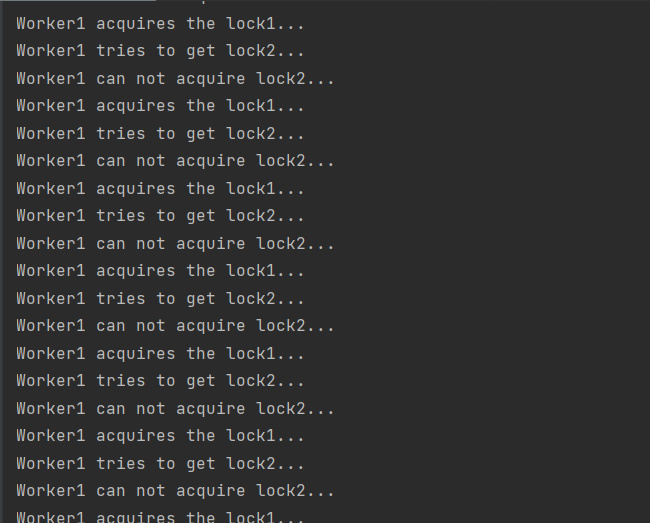
public void worker1() {  
 lock1.lock();  
 System.*out*.println("Worker1 acquires the lock1...");  
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 lock2.lock();  
 System.*out*.println("Worker1 acquires the lock2...");  
 lock1.unlock();  
 lock2.unlock();  
 System.*out*.println("Worker 1 finished");  
}  
 public void worker2() {  
 lock1.lock();  
 System.*out*.println("Worker2 acquires the lock1...");  
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 lock2.lock();  
 System.*out*.println("Worker2 acquires the lock2...");  
 lock1.unlock();  
 lock2.unlock();  
 System.*out*.println("Worker 2 finished");  
  
}

* Thread1 blocheaza lock1 in worker1, apoi intra in somn
* Thread2 vrea sa bloceze lock1 in worker2, dar nu poate deoarece Thread1 deja a blocat lock1, si deci thread2 intra in waitin state
* Acum thread1 da lock la lock2, apoi unlock la lock1 si lock2
* Apoi thread2 da lock la lock1 cu succes si apoi la lock2 si unlock la fiecare pe rand si totul e bine

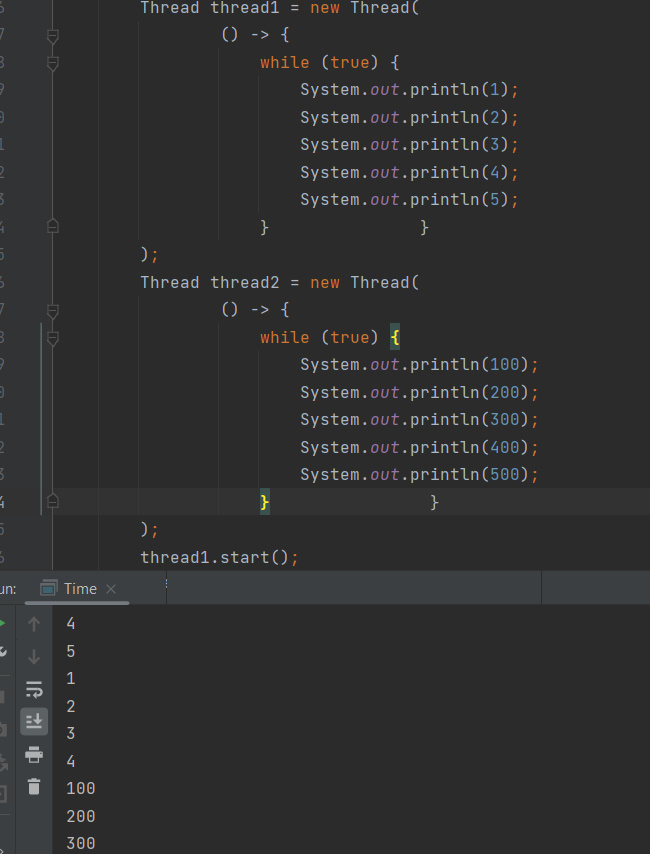
**LiveLock example**

import java.util.concurrent.TimeUnit;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class LivelockExample {  
  
 private Lock lock1 = new ReentrantLock(true);  
 private Lock lock2 = new ReentrantLock(true);  
  
 public static void main(String[] args) {  
  
 LivelockExample livelock = new LivelockExample();  
  
 new Thread(livelock::worker1, "worker1").start();  
 new Thread(livelock::worker2, "worker2").start();  
 }  
  
 private void worker1() {  
 while (true) {  
 try {  
 lock1.tryLock(50, TimeUnit.*MILLISECONDS*);  
 System.*out*.println("Worker1 acquires the lock1...");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("Worker1 tries to get lock2...");  
 if (lock2.tryLock()) {  
 System.*out*.println("Worker1 acquires the lock2...");  
 lock2.unlock();  
 System.*out*.println("Worker1 frees the lock2...");  
 } else {  
 System.*out*.println("Worker1 can not acquire lock2...");  
 continue;  
 }  
 break;  
 }  
 lock1.unlock();  
 lock2.unlock();  
 }  
 private void worker2() {  
 while (true) {  
 try {  
 lock2.tryLock(50, TimeUnit.*MILLISECONDS*);  
 System.*out*.println("Worker2 acquires the lock2...");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("Worker2 tries to get lock1...");  
 if (lock1.tryLock()) {  
 System.*out*.println("Worker2 acquires the lock1...");  
 lock1.unlock();  
 System.*out*.println("Worker2 frees the lock1...");  
 } else {  
 System.*out*.println("Worker2 can not acquire lock1...");  
 continue;  
 }  
 break;  
 }  
 lock1.unlock();  
 lock2.unlock();  
 }  
}

Si se intampla la infinit fie asta:



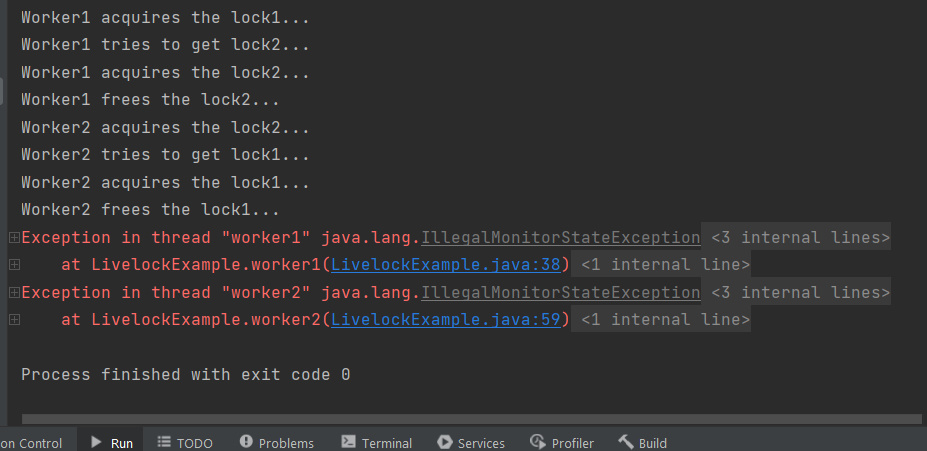
* Atentie! Un thread poate fi intrerupt de time slacin alhorithm chiar si de nu termina iteratia complet:





* deci thread1 da lock la lock1, apoi el nu reuseste sa dea lock la lock2, ca e intrerupt de time slicing algorithm si thread2 da lock la lock2.
* Mai departe, nu mai conteaza cum se executa de time slicing alghoritm, oricum se va intampla mereu ca thread1 nu va putea da lock la lock2 si nici thread2 la lock1 si asta e bucla infinita
* **Iata de ce apare livelock, ambele incearca sa accese o resursa blocata de alt thread si nu intra in waitin state, dar incearca la infinit!**
* Dar, poate thread1 reuseste sa execute o iteratie complet

si se intampla asta



* thread1 da lock la lock1 si la lock2 apoi da free la lock2
* thread2 da lock la lock2 si la lock1 si apoi le da free la lock
* Problema e ca dupa ce termina while, ambele au asa ceva:
* lock1.unlock();  
  lock2.unlock();

dar thread1 deja a dat unlock la lock2 si thread2 la lock1 si se da unlock iar, dar nu poate sa dea unlock la un lock pe care nu il apartine!

**Variabile Atomice**

* Sa zicem ca avem o variabila a carei valoare va fi marita cu 1 de 1000 de ori de 2 threaduri
* Iarasi, daca nu facem blocul pentru incrementare sincronizat, vor aparea probleme, caci variabila nu va 2000
* Sincronizarea este o solutie, dar este foarte scumpa, asa cum un thread este blocat si trebuie apoi repornit.
* Este mai usor sa folosim variabile atomice, deja create de Java, care se asigura ca sa fie executate asa cum trebuie de fiecare thread si sa nu fie stocate in cache. Cu volatile nu putem face asa ceva, caci ++a nu e atomic
* Avem clasele:

AtomicInteger, AtomicLong, AtomicBoolean etc.

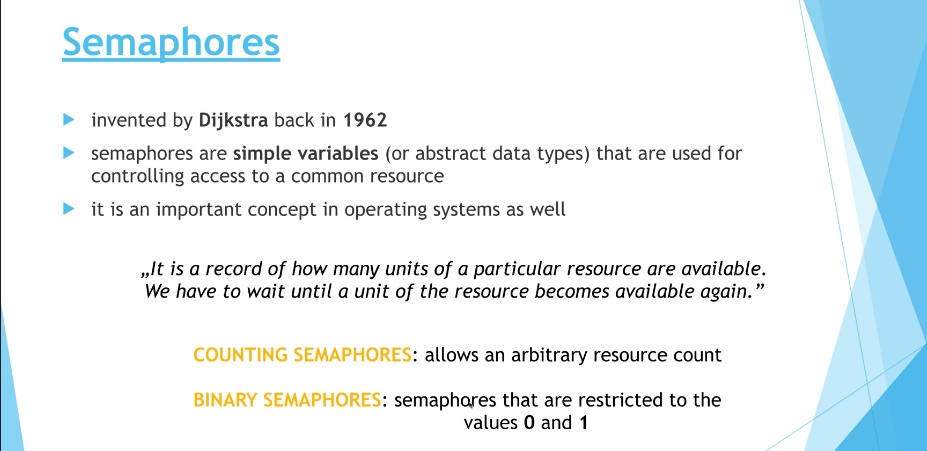
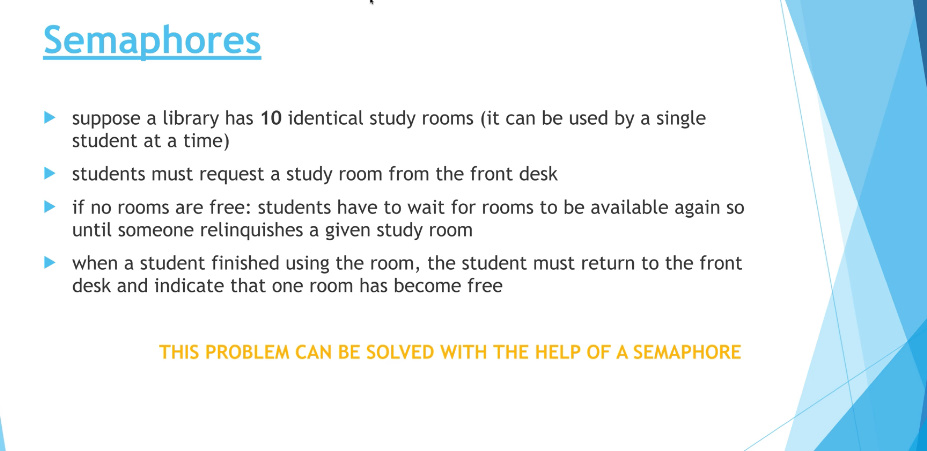
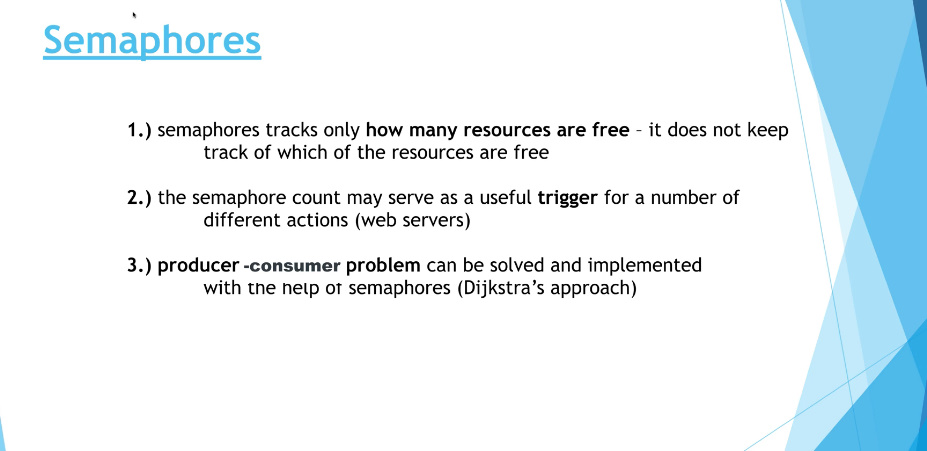
* **AtomicInteger variabila = new AtomicInteger(valoare);**
* get() – returneaza valoarea curenta
* **incrementAndGet()** – mareste cu 1 si returneaza noua valoare
* **set(val)** – seteaza noua valoare

**Semaphores**

* **Semaphores** – controleaza accesul la o resursa shared prin folosirea unui counter.
* Daca valoarea la acest counter e mai mare ca 0, accesul e permis, daca e 0, nu e permis.
* Acest counter indica cate threaduri mai pot accesa aceasta resursa sau cate accesuri mai sunt permise
* Semaforul decide daca sa permita accesul la resursa petru thread, Daca threadul primeste acces de la semafor, executa resursa si semaforul decrementeaza counterul cu 1
* Daca semaforul bloceaza accesul, threadul e blocat
* Cand threadul termina cu resursa, el i da drumul si counter de la semafor va fi marit.
* **Constructori:**
* Semaphore(int num)
* Semaphore(int num, boolean how)

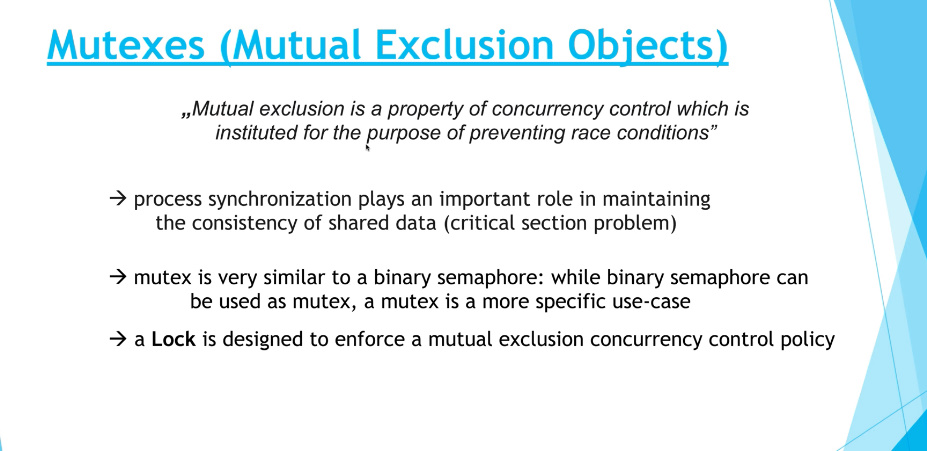
num – numarul de threaduri ce pot accesa in acelasi timp aceeasi resursa

how – in mod normal, threadurile acceseaza resursa intr-o ordine nedeterminata, dar daca punem true, ele vor primi permisiunea in ordinea venirii lor

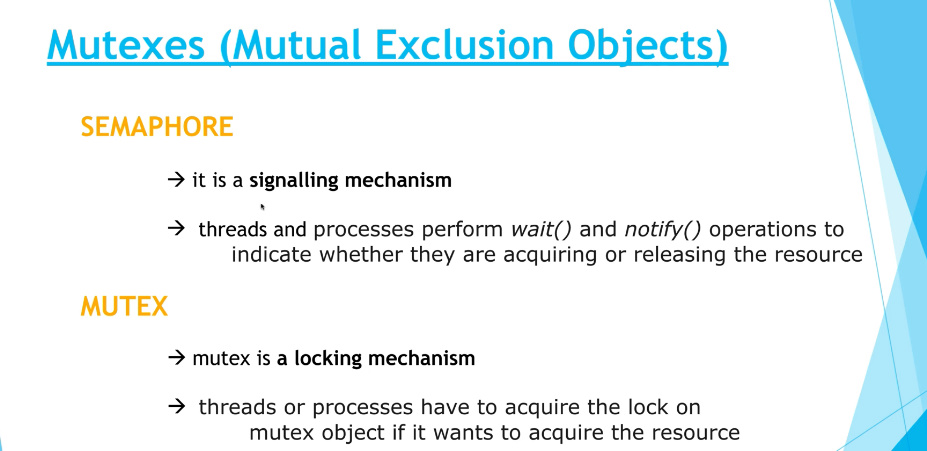
* Pentru ca o resursa sa poata accesa resursa, sa-i dea lock adica, trebuie sa cheme metoda **acquire() sau acquire(int permits),** unde permits este numarul de permisiuni ce ia un thread
* cand Threadul termina cu resursa, o elibereaza cu **release()** sau **release(int permits)**
* **availablePermis()** – returneaza nr de resurse permise
* 
* 
* 

**Mutexes**

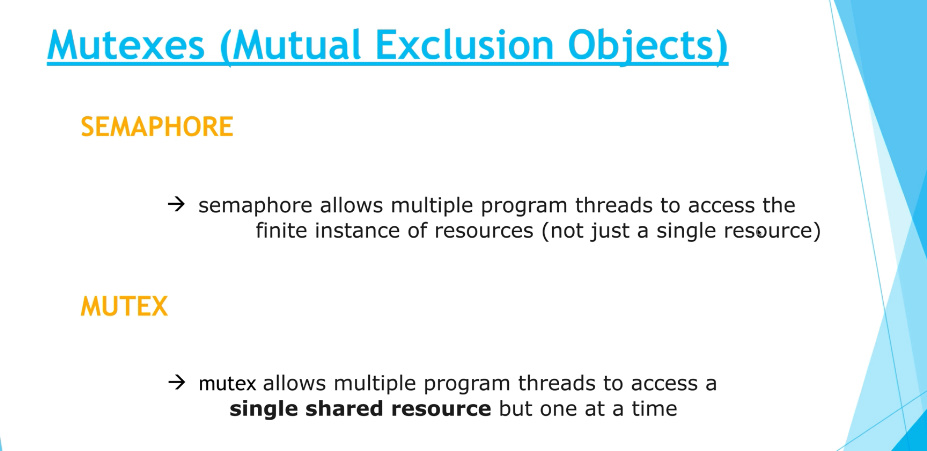
* **Mutual Exclusion Objects(Mutexes)** sunt foarte asemanatoare cu semafoarele
* Mutex este acel **lock** care permite unui singur Thread sa dea lock si sa acceseze resursa.
* **synchronized(mutex) { }**

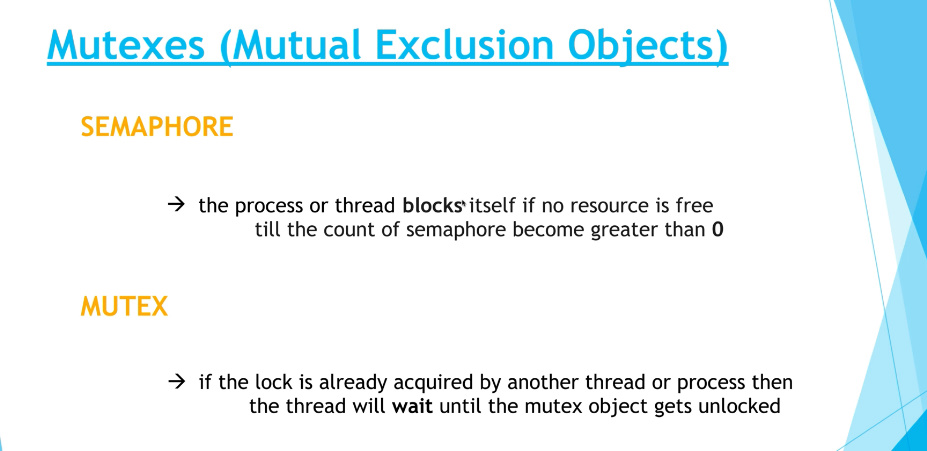


Iata de ce, mutex e similiar cu un semafor binar, deoarece poate avea fie un thread care da lock, fie nici unul(0 sau 1)

* 

semaforul nu este un lockin mecanism!





Daca semaforul nu ofera permisiunea la thread, atunci threadul va fi blocat pana nu o primeste, adica pana un alt thread nu elibereaza resursa

class Time {  
 public static void main(String[] args) throws InterruptedException {  
  
 Run run = new Run();  
  
 for(int i = 0;i<12;i++){  
 new Thread(run).start();  
 }  
 }  
}  
  
class Run implements Runnable{  
  
 private Semaphore semaphore = new Semaphore(2);  
  
 @Override  
 public void run() {  
 try {  
 semaphore.acquire();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println("A tread is executing");  
  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 semaphore.release();  
 }  
}