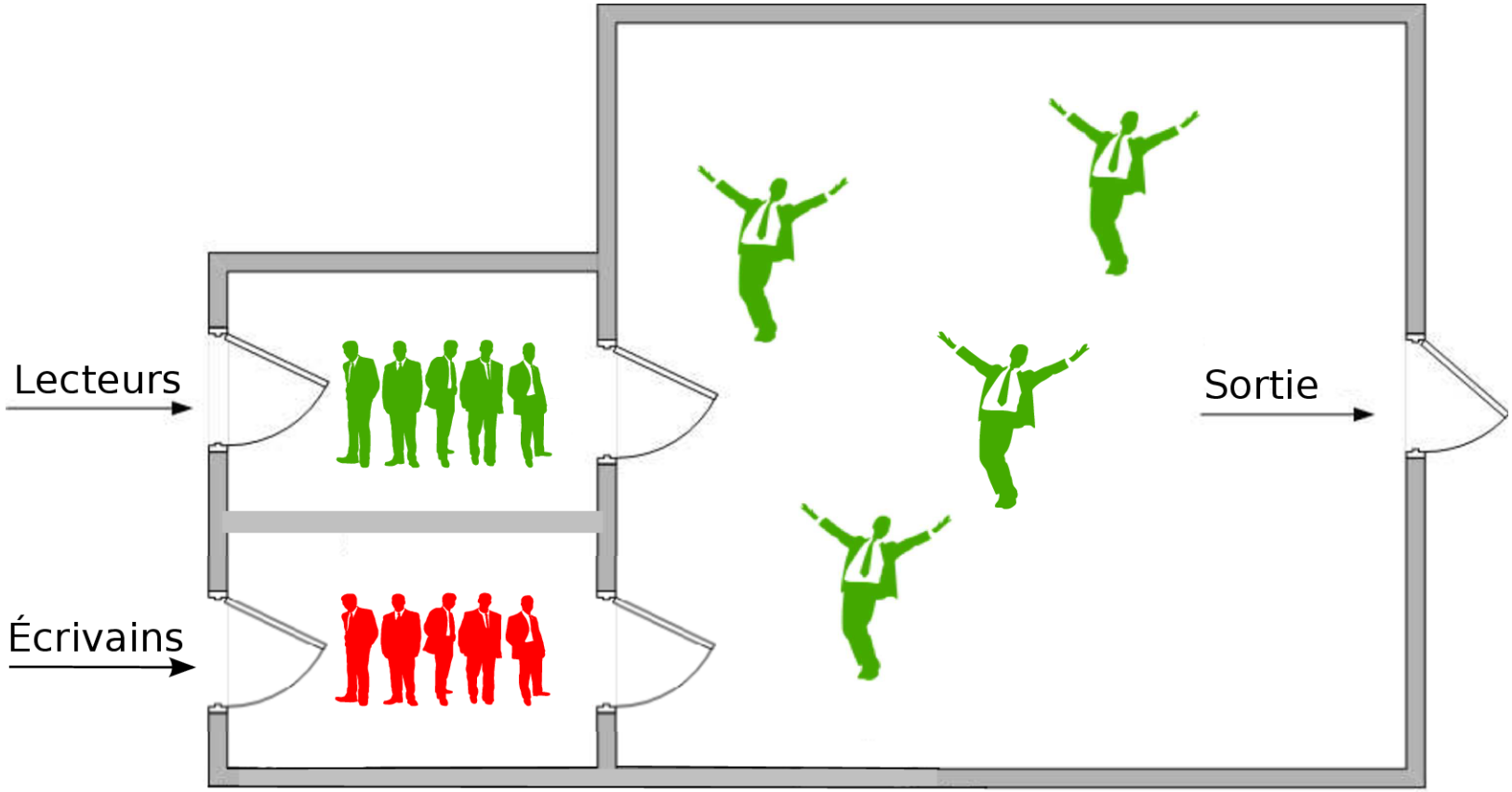
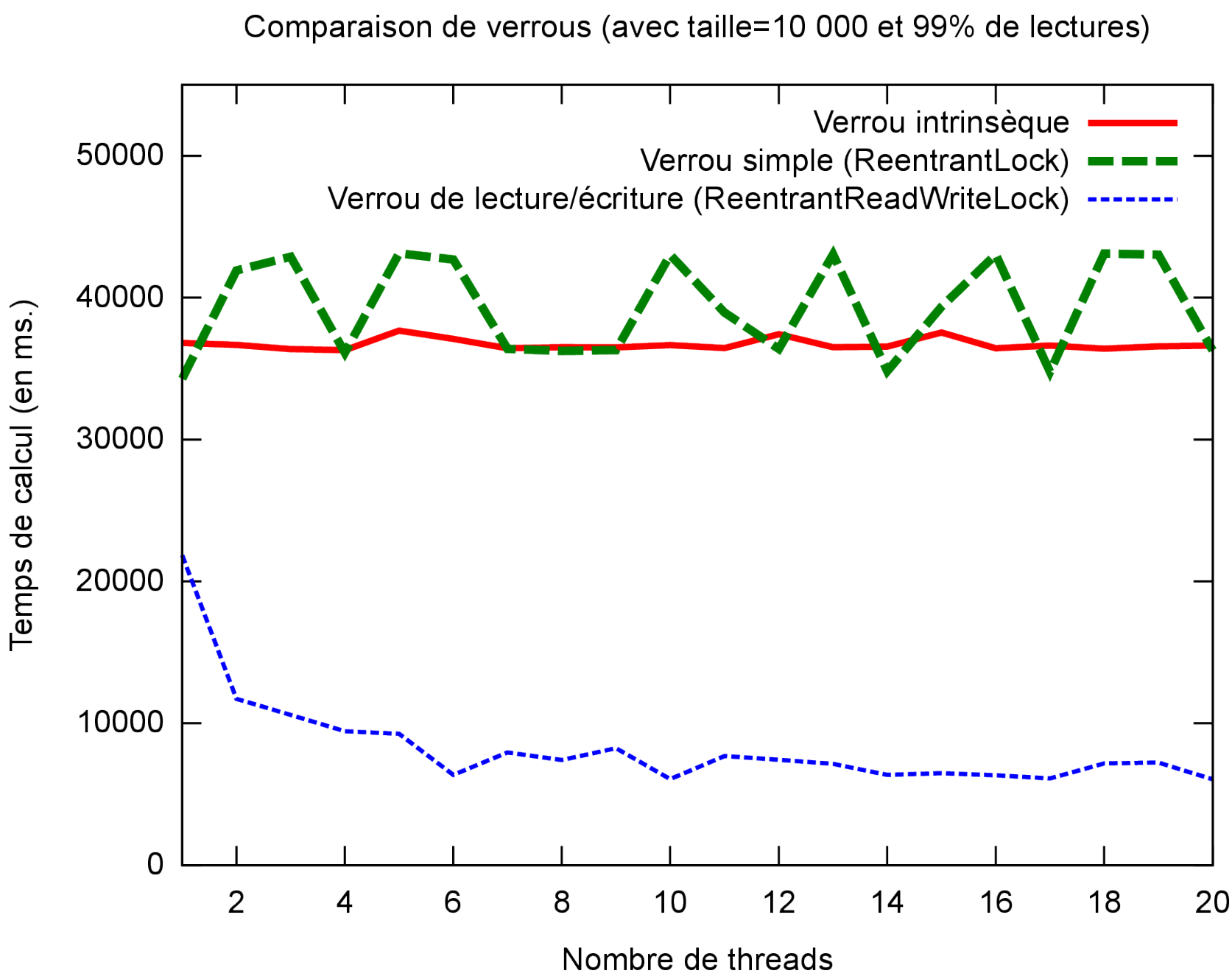
Fonctionnement d'un verrou de Lecture/Ecriture



Plusieurs threads peuvent posséder le verrou de lecture simultanément !

Illustration du gain sur un tableau à 10 000 éléments



Exemple de Callable

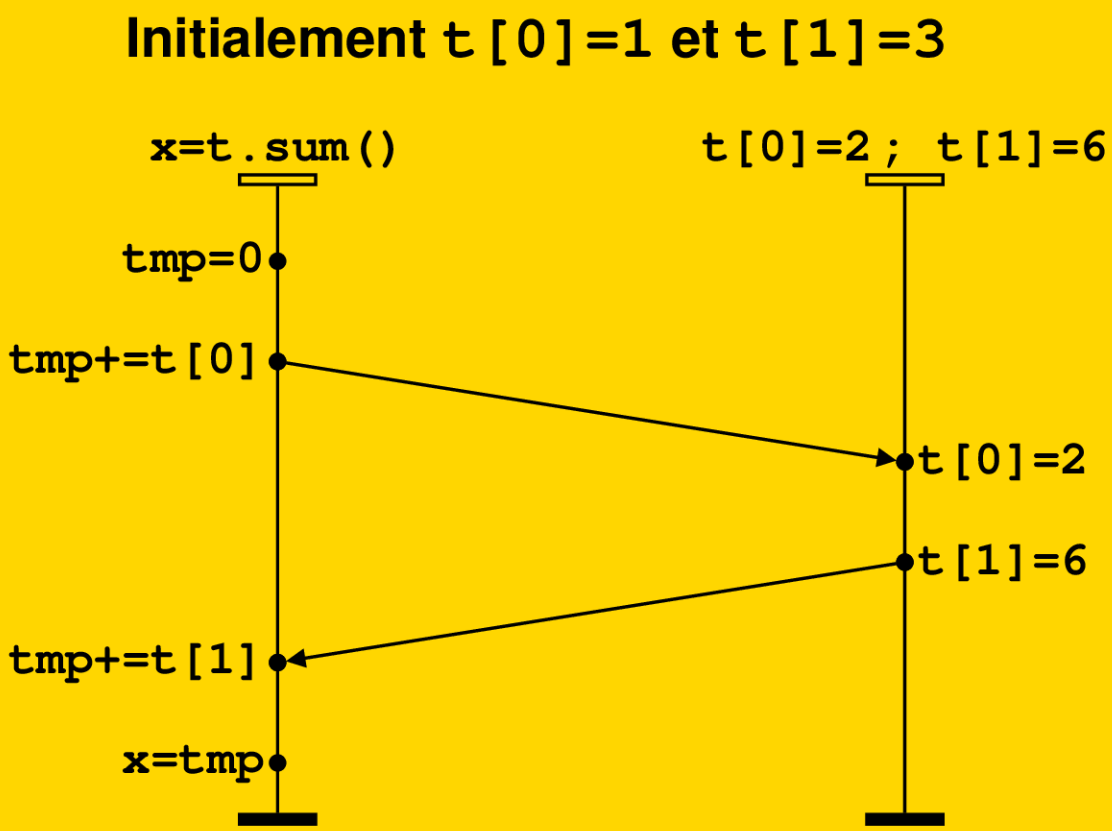
```
class MultipleLancer implements Callable<Long>{
  long nbTirages;
  long tiragesDansLeDisque = 0 ;
  MultipleLancer(long nbTiragesAEffectuer){
    nbTirages = nbTiragesAEffectuer;
  }
  public Long call(){
    double x, y;
    for (int i = 0; i < nbTirages; i++) {
      x = ThreadLocalRandom.current().nextDouble(1);
      y = ThreadLocalRandom.current().nextDouble(1);
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
    return tiragesDansLeDisque;
  }
}
```

Notez le return() dans la méthode call().

Utiliser un service de completion

```
ExecutorService executeur=Executors.newFixedThreadPool(10) ;
CompletionService<Long> ecs =
                    new ExecutorCompletionService<Long>(executeur) ;
for (int j = 0; j < 10 ; j++) {
  MultipleLancer tache = new MultipleLancer( nbTirages/10 );
  ecs.submit(tache);
}
int tiragesDansLeDisque = 0;
for (int j = 0; j < 10; j++) {
    Long resultatAttendu = ecs.take().get(); // Ordre indéterminé!
    tiragesDansLeDisque += resultatAttendu;
}
double resultat = (double) tiragesDansLeDisque / nbTirages ;
```

Itérer sur une collection peut poser des problèmes



Que vaut **x** *à la fin ?*

La valeur retournée par la méthode **`sum()`** peut correspondre à une liste qui n'a jamais existé !

Collection et concurrence

En Java, il y a trois façons d'obtenir des collections qui fonctionnent correctement en présence de plusieurs threads : on dit ≪ *thread-safe* ≫ pour faire court.

① Utiliser des collections ≪ synchronisées ≫, telles que **`Vector`** ou **`HashTable`**... Chaque appel requiert le *verrou intrinsèque* de l'objet (comme dans un moniteur).

② Construire des classes synchronisées à partir de collections qui ne le sont pas (depuis le JDK 1.2).

③ Utiliser les nouvelles collections ≪ concurrentes ≫ présentes depuis JDK 1.5 ; celles-ci n'utilisent pas **`synchronized`** ; elles sont construites à l'aide de verrous privés ou d'objets atomiques. Ces classes autorisent les accès simultanés et sont donc potentiellement plus performantes que les collections synchronisées.

Il faut retenir ici qu'il existe en effet des collections qui ne sont pas thread-safe : **`HashMap`**, **`ArrayList`**, etc. mais que l'on peut avantageusement utiliser si l'on sait qu'elles ne seront jamais utilisées par plusieurs threads simultanément.