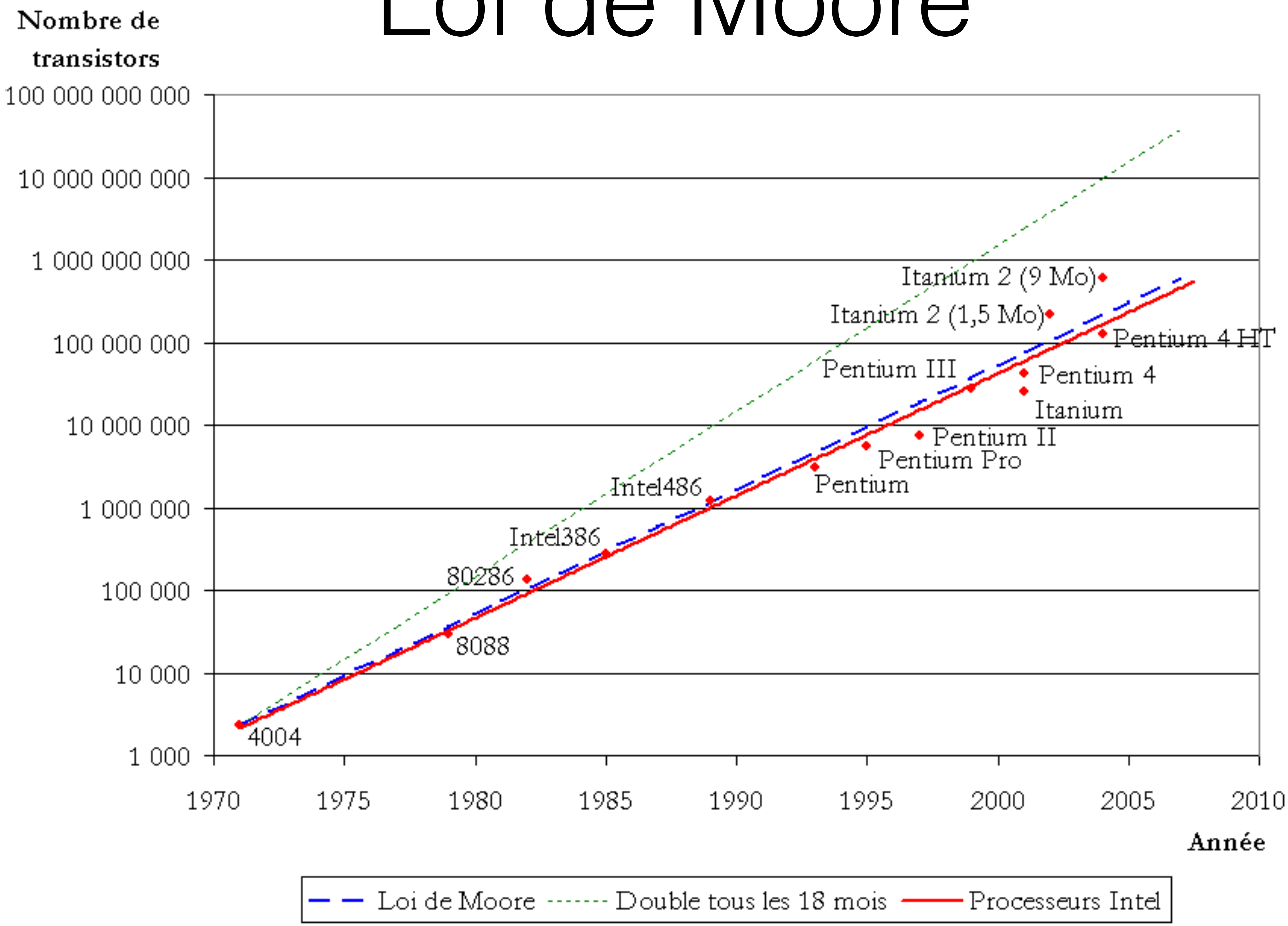
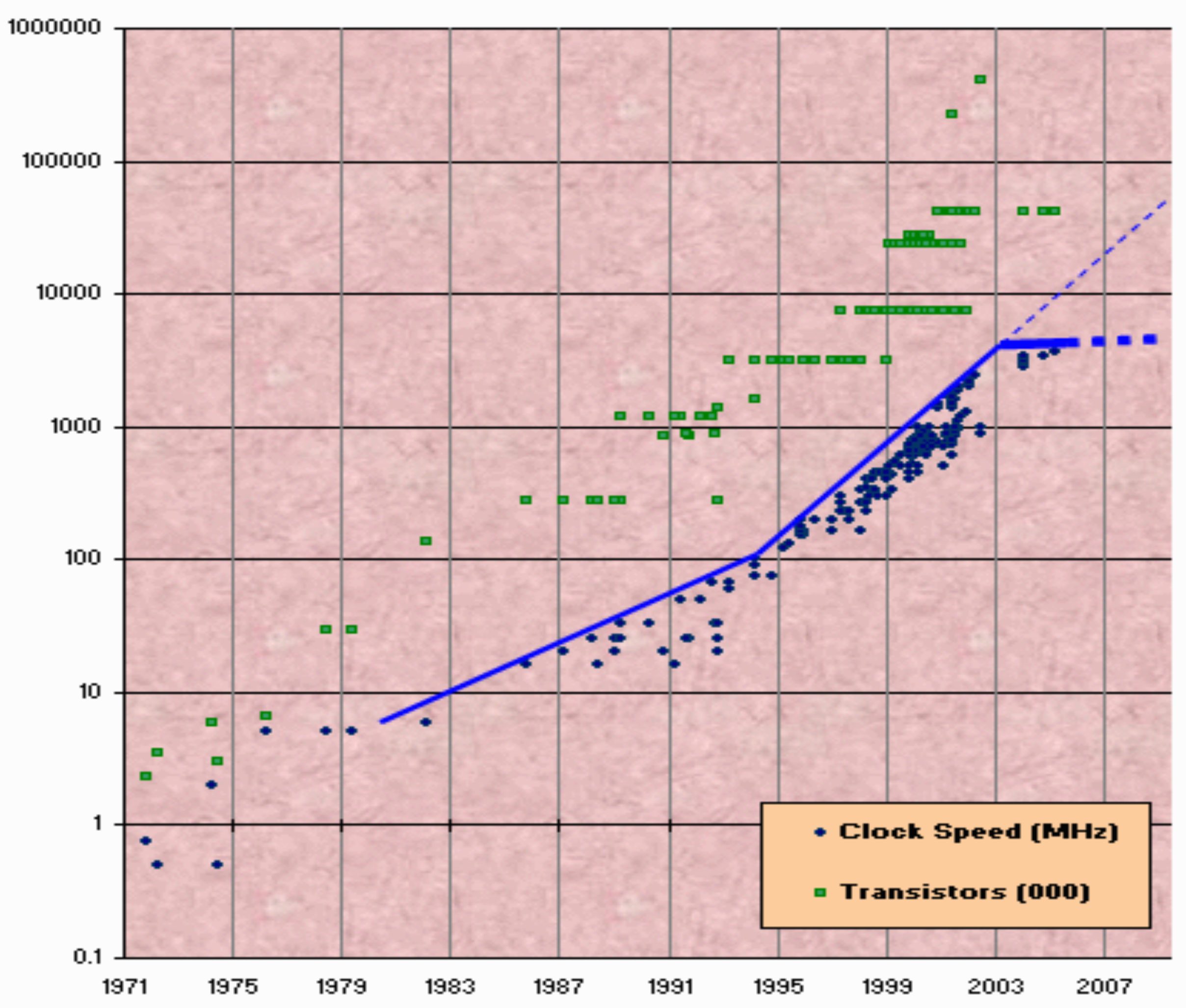
Introduction…

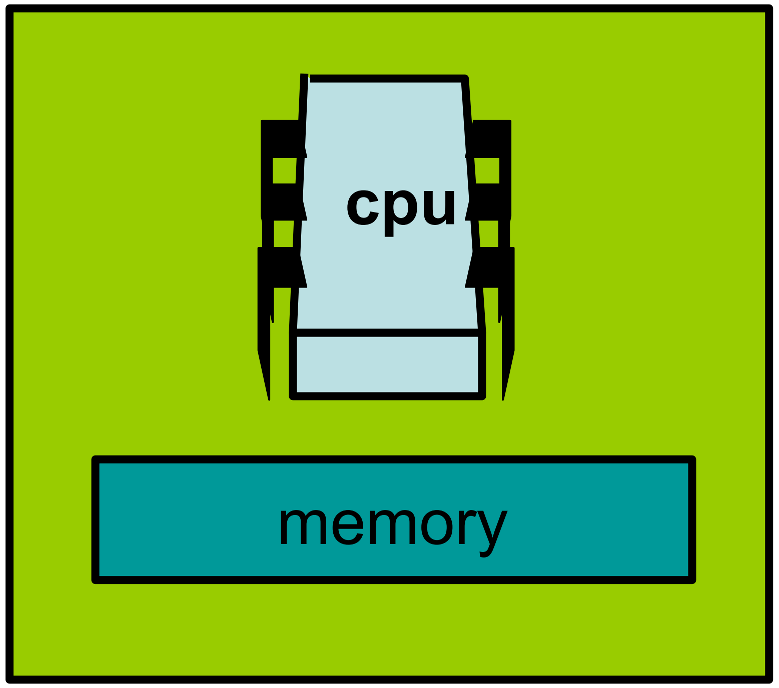
Loi de Moore


Nombre de transistors
100 000 000 000
10 000 000 000
1 000 000 000
100 000 000
10 000 000
1 000 000
100 000
10 000
1 000
1970
1975
1980
1985
1990
1995
2000
2005
2010
Année
4004
8088
80286
Intel386
Intel486
Pentium
Pentium Pro
Pentium II
Pentium III
Itanium
Pentium 4
Pentium 4 HT
Itanium 2 (1,5 Mo)
Itanium 2 (9 Mo)
Loi de Moore
Double tous les 18 mois
Processeurs Intel

Loi de Moore…

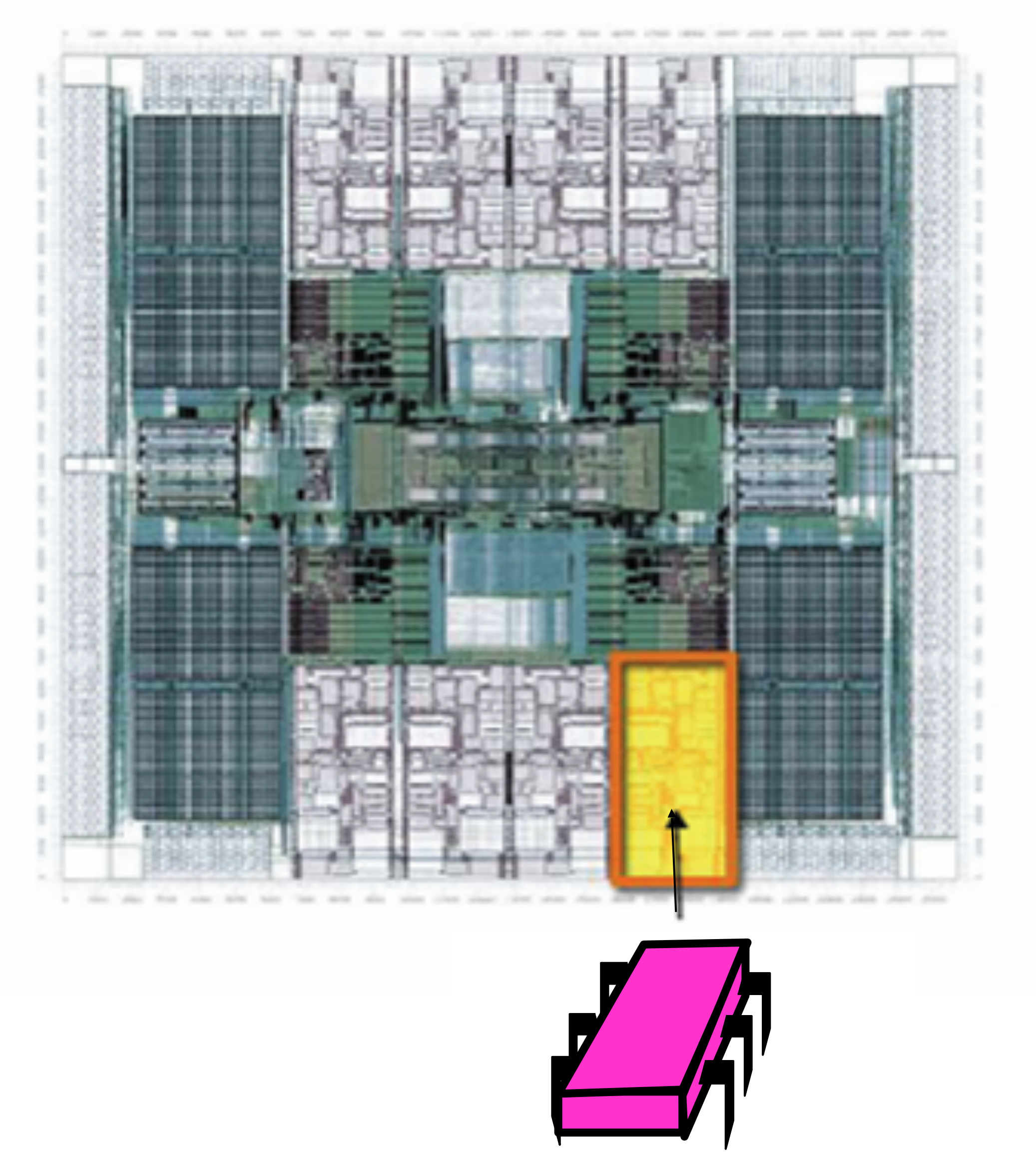

1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
● Clock Speed (MHz)
🟩 Transistors (000)

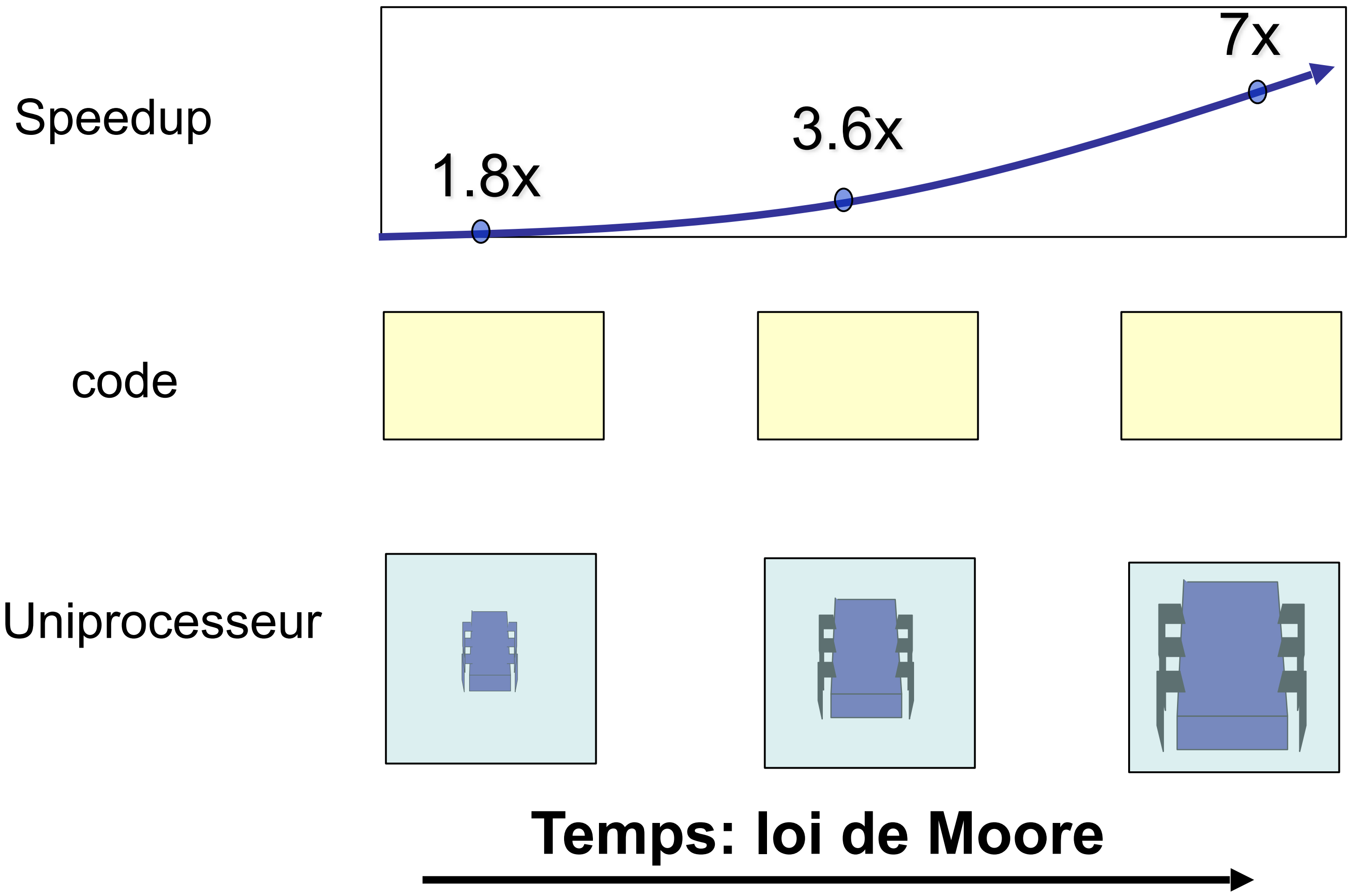
Uniprocesseur



Mémoire partagée

multicore





Speedup

1.8x

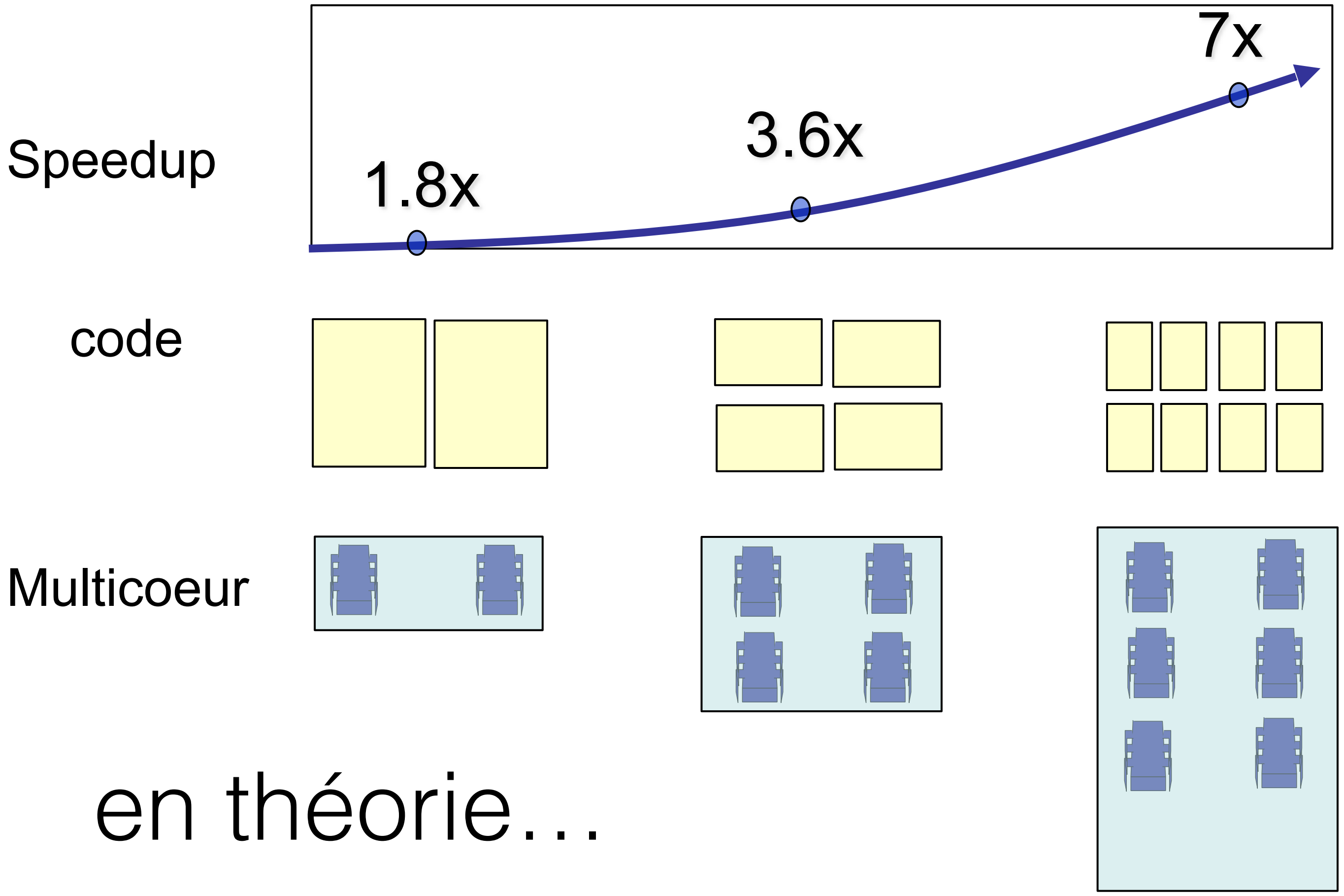
3.6x

7x

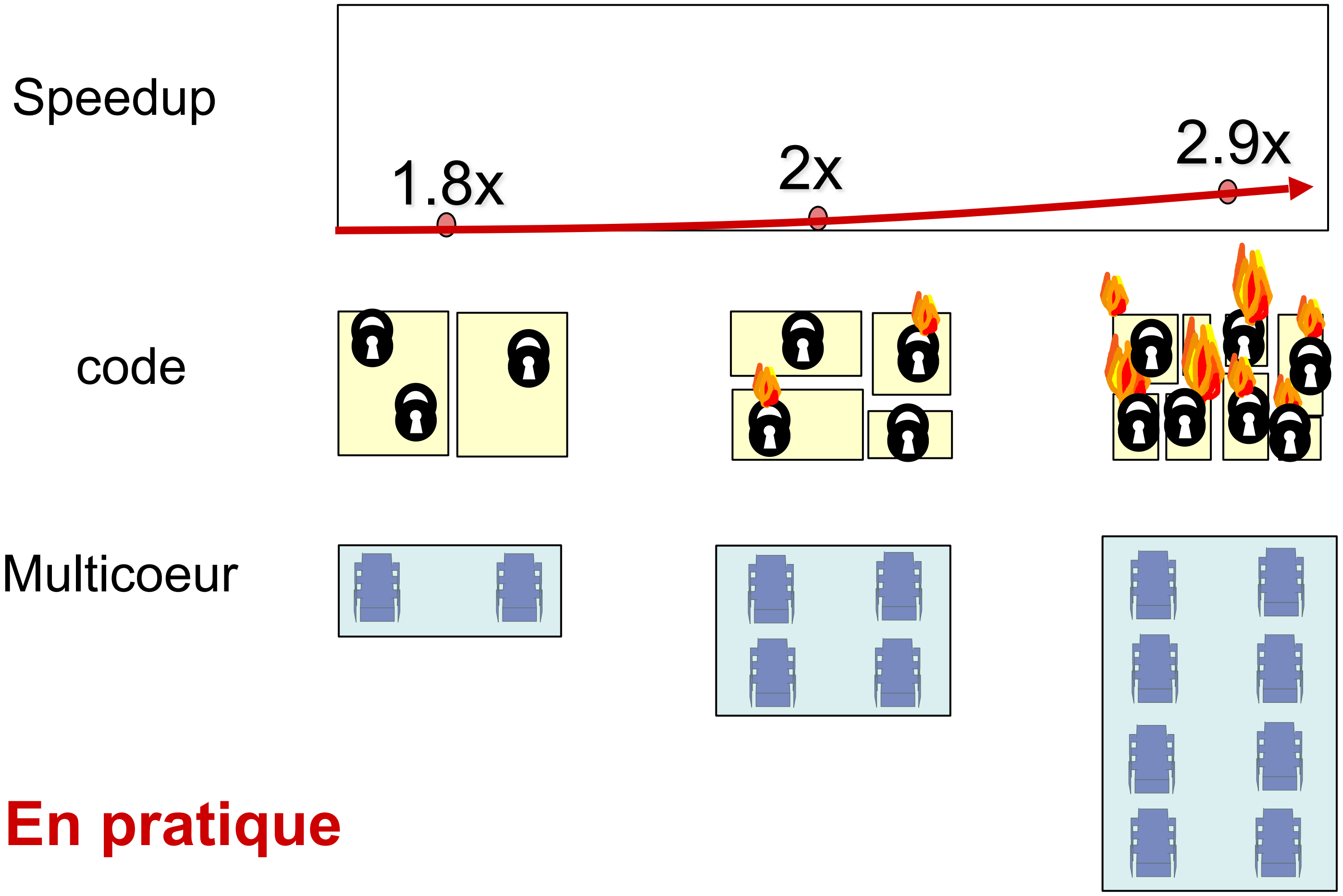
code

Uniprocesseur

Temps: loi de Moore



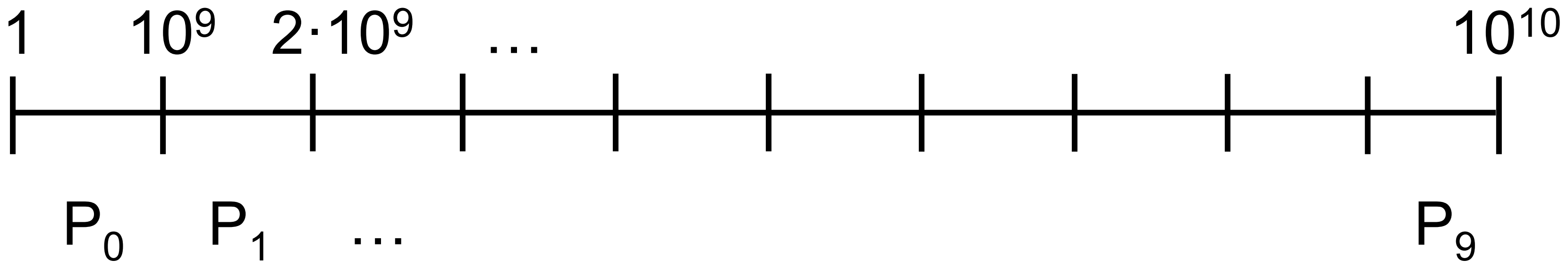
en théorie…



En pratique

Exemple

- afficher les nombres premiers entre 0 et 10^{10}



10 threads: chacune sur un intervalle de 10^9

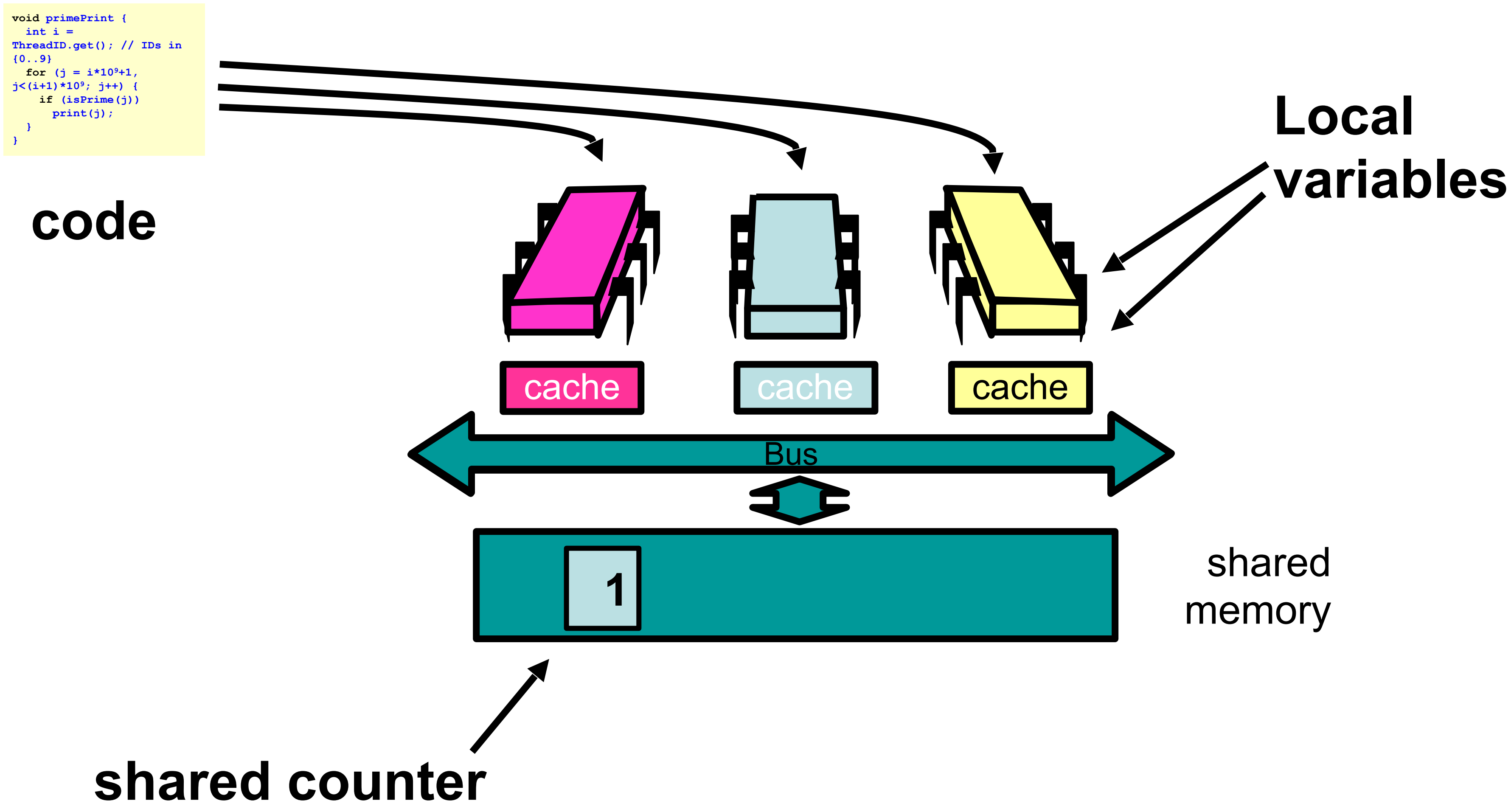
```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

Mais…

Autre solution

- Chaque thread teste la primarité par un nombre (obtenu par un compteur partagé)

```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```



```
void primePrint {
  int i =
ThreadID.get(); // IDs in
{0..9}
  for (j = i*10^9+1,
j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

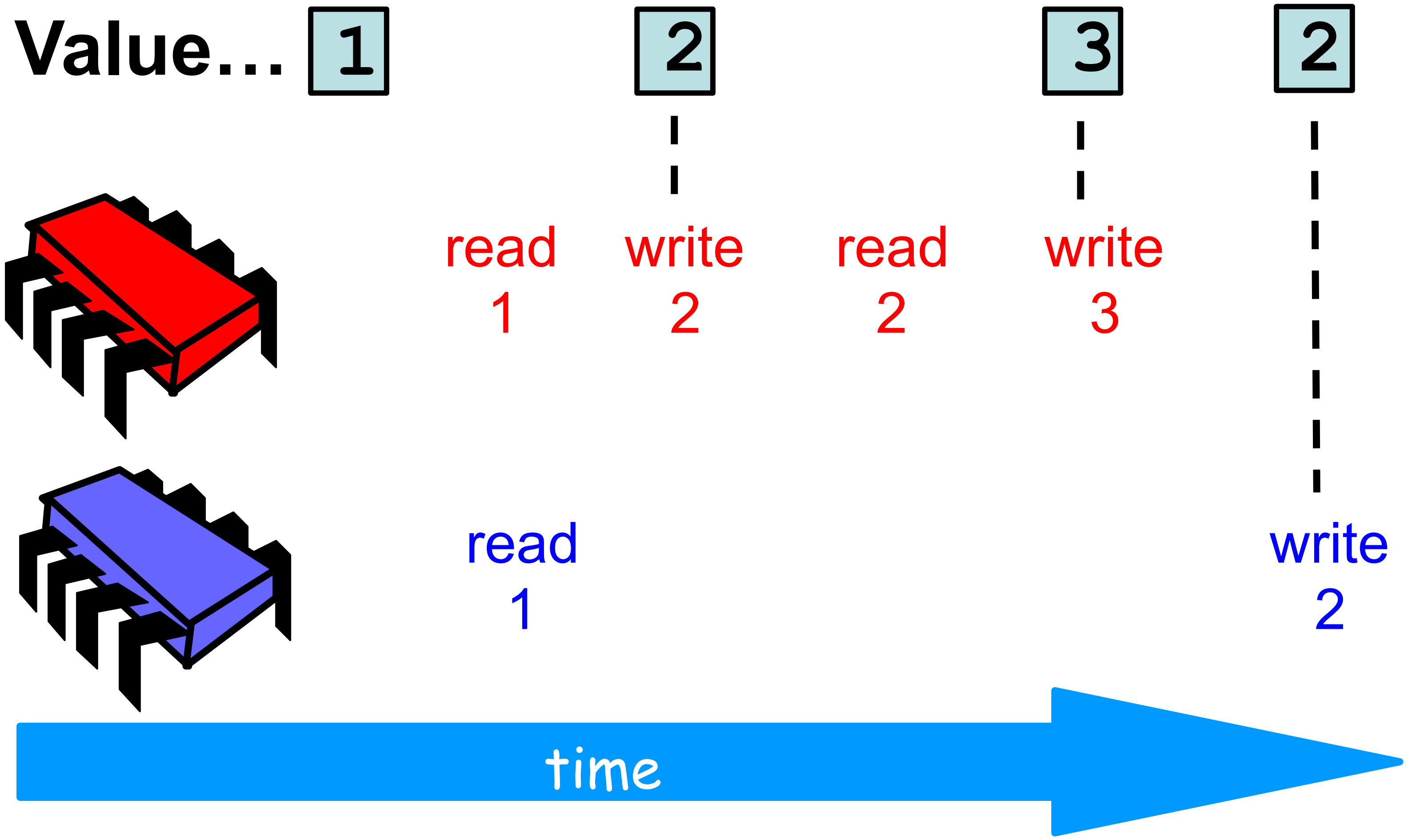
Shared counter?

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

`value++`

```
temp  = value;
value = temp + 1;
return temp;
```

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```



Value… 1 2 3 2

read 1 write 2 read 2 write 3

read 1 write 2

time

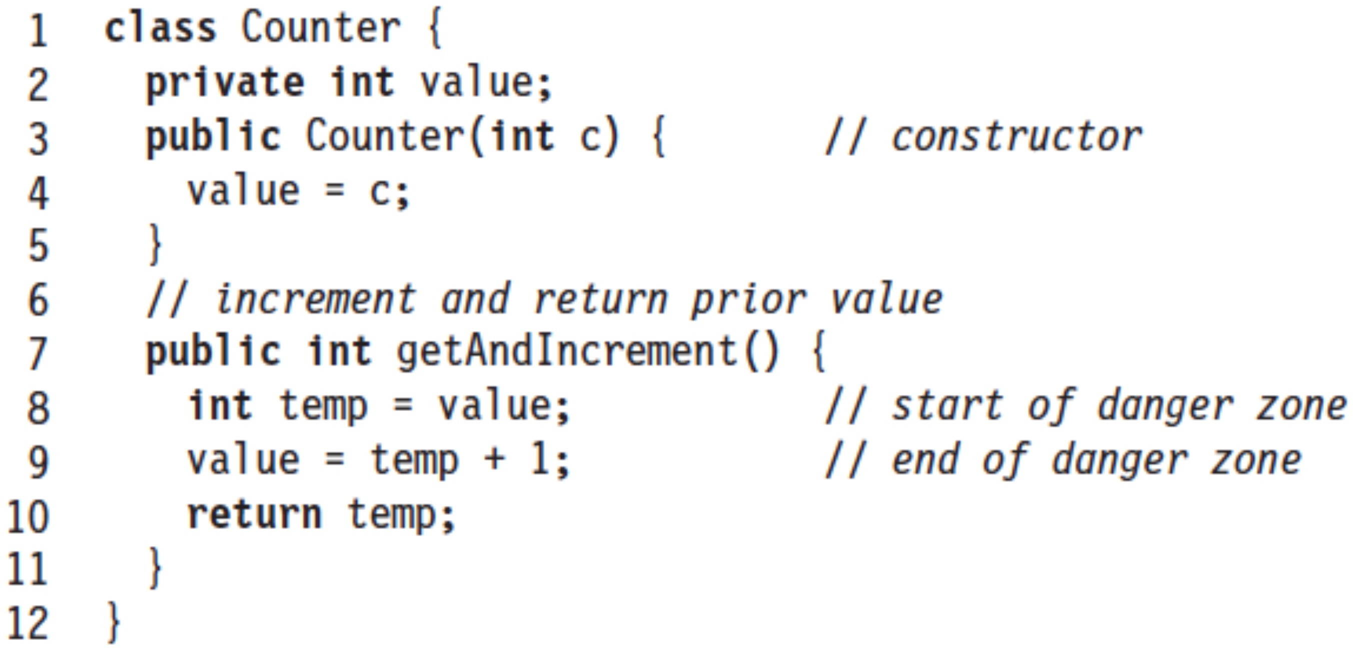
```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

atomique

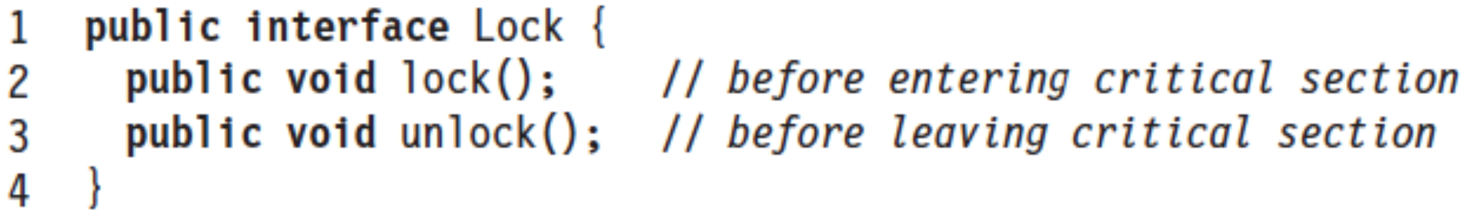
En java

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

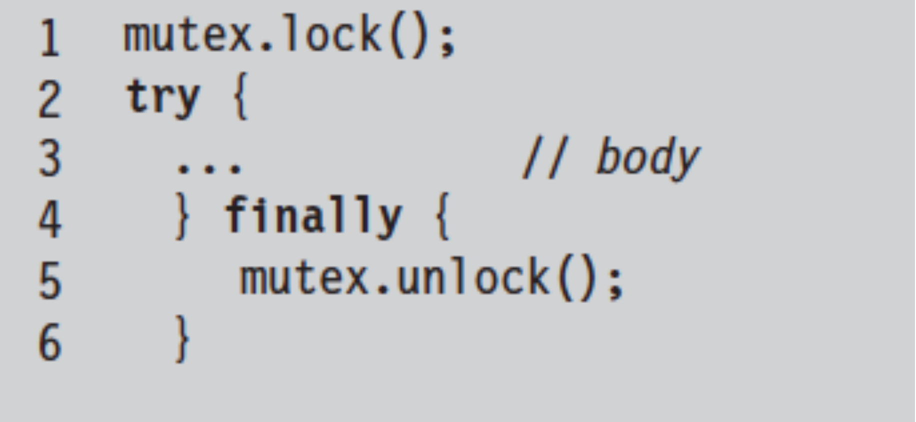
Exclusion mutuelle



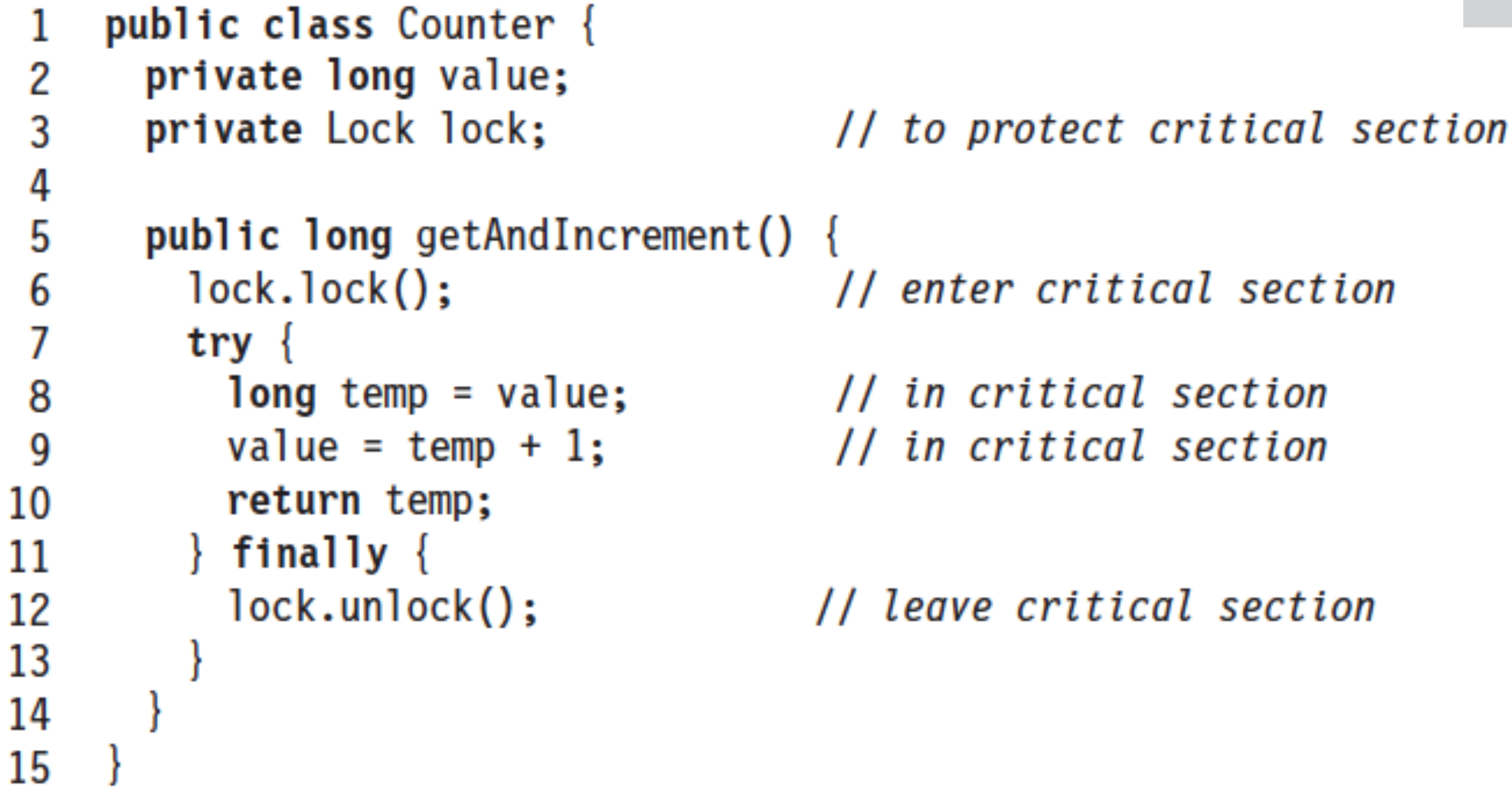
```
class Counter {
  private int value;
  public Counter(int c) {       // constructor
    value = c;
  }
  // increment and return prior value
  public int getAndIncrement() {
    int temp = value;           // start of danger zone
    value = temp + 1;           // end of danger zone
    return temp;
  }
}
```



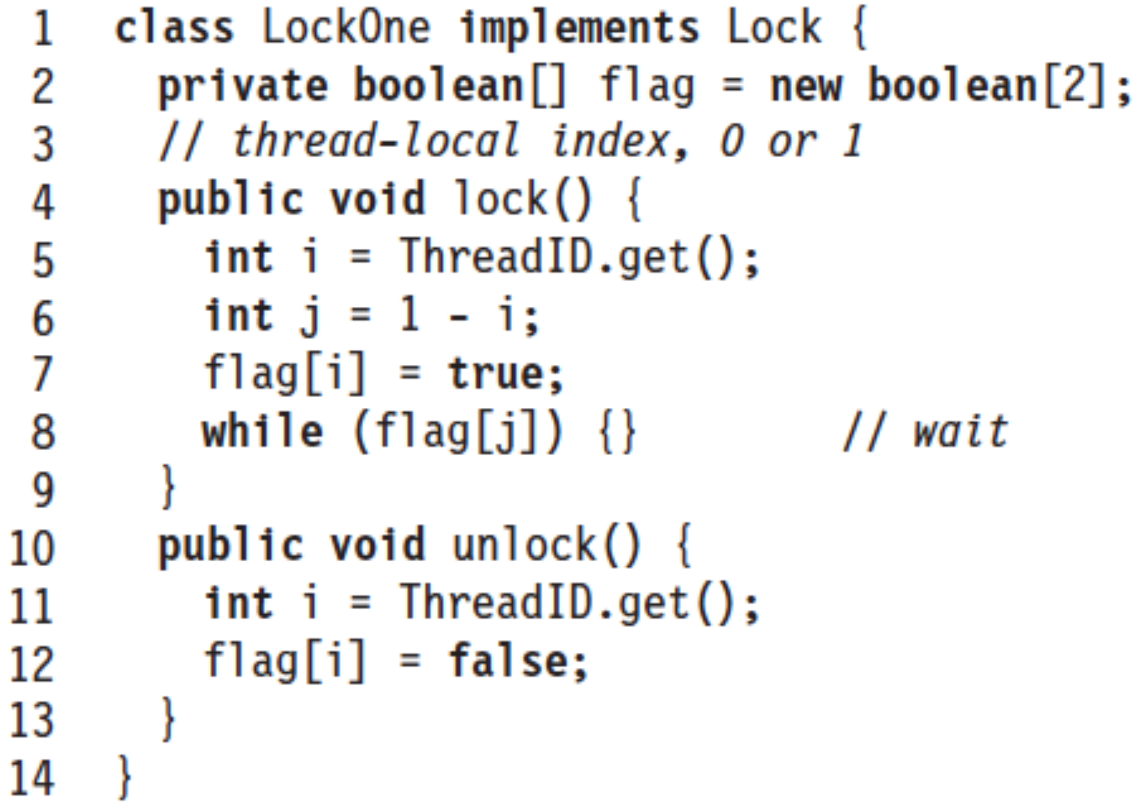
```
public interface Lock {
  public void lock();     // before entering critical section
  public void unlock();   // before leaving critical section
}
```



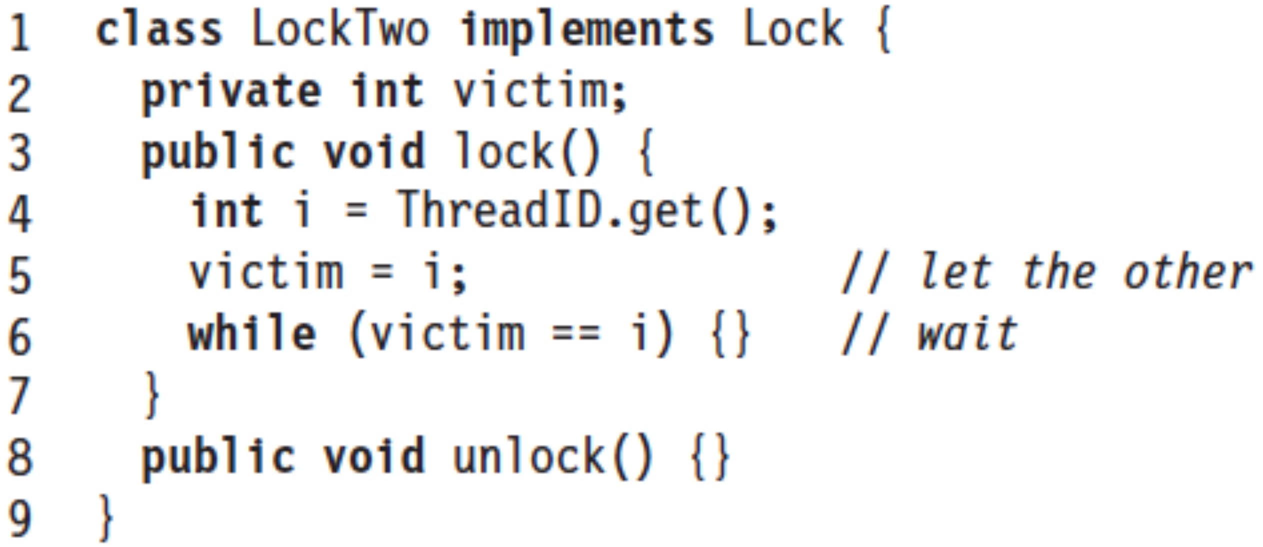
```
mutex.lock();
try {
  ...             // body
  } finally {
    mutex.unlock();
  }
```



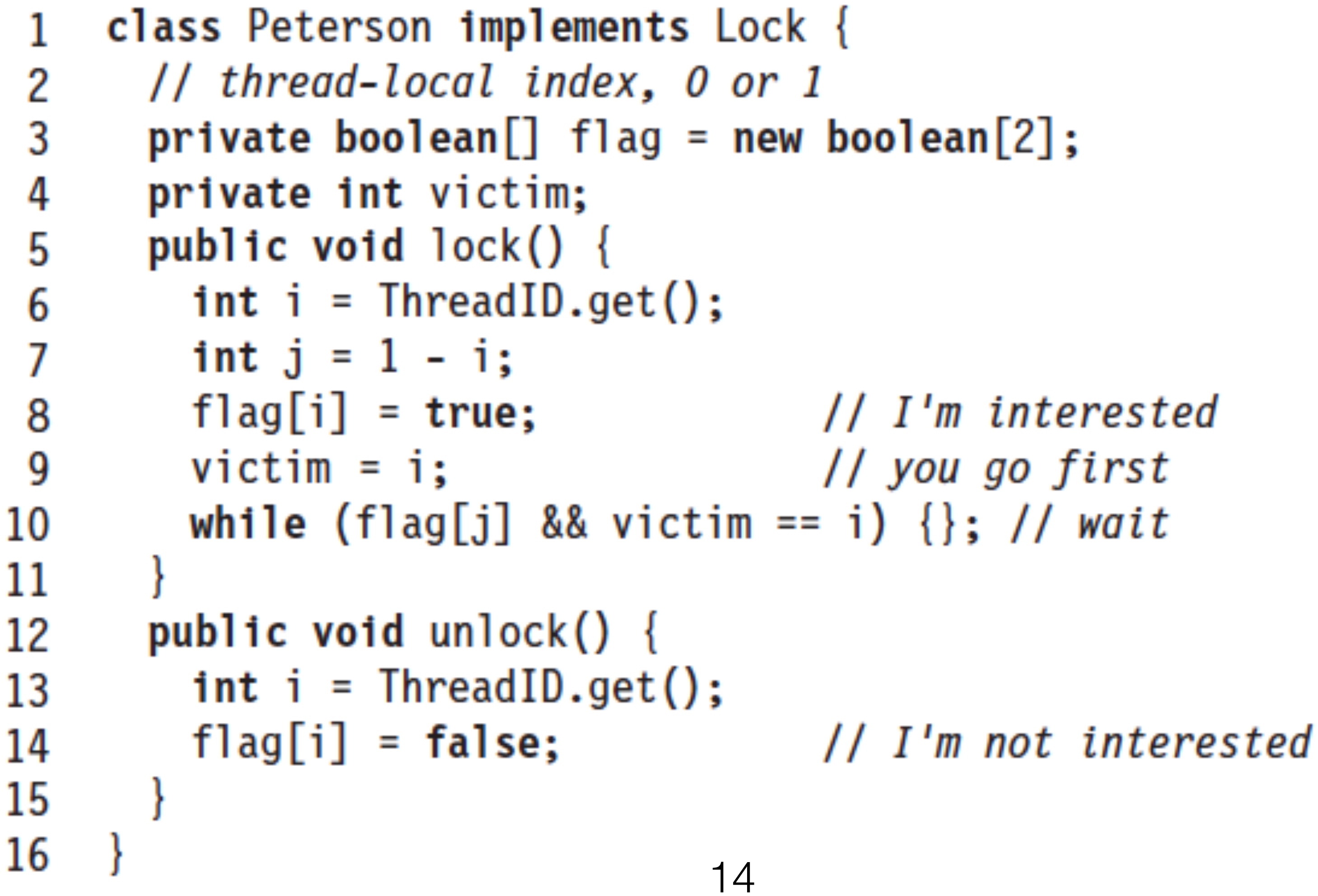
```
public class Counter {
  private long value;
  private Lock lock;              // to protect critical section

  public long getAndIncrement() {
    lock.lock();                  // enter critical section
    try {
      long temp = value;          // in critical section
      value = temp + 1;           // in critical section
      return temp;
    } finally {
      lock.unlock();             // leave critical section
    }
  }
}
```



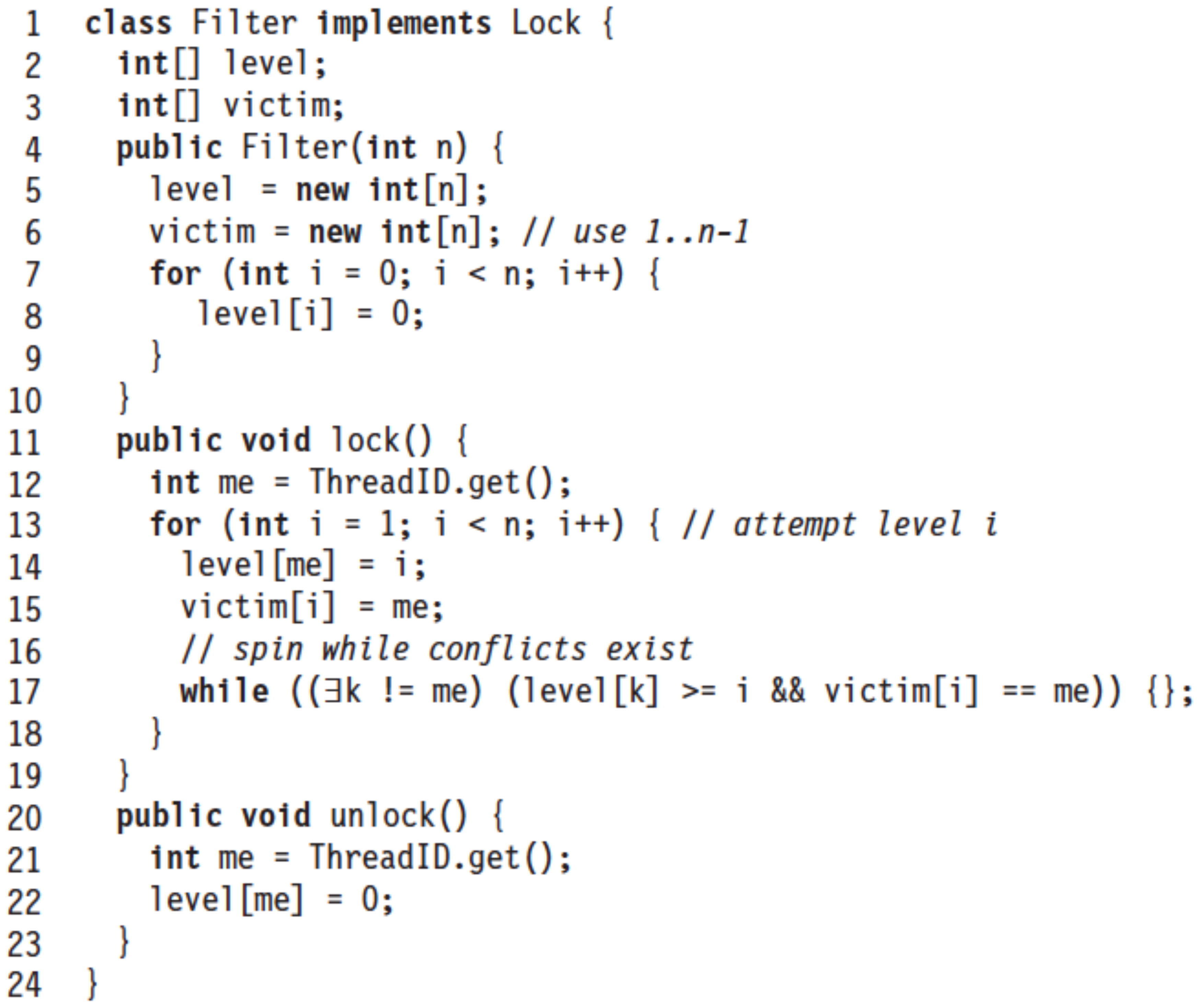
```
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];
  // thread-local index, 0 or 1
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    flag[i] = true;
    while (flag[j]) {}          // wait
  }
  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```



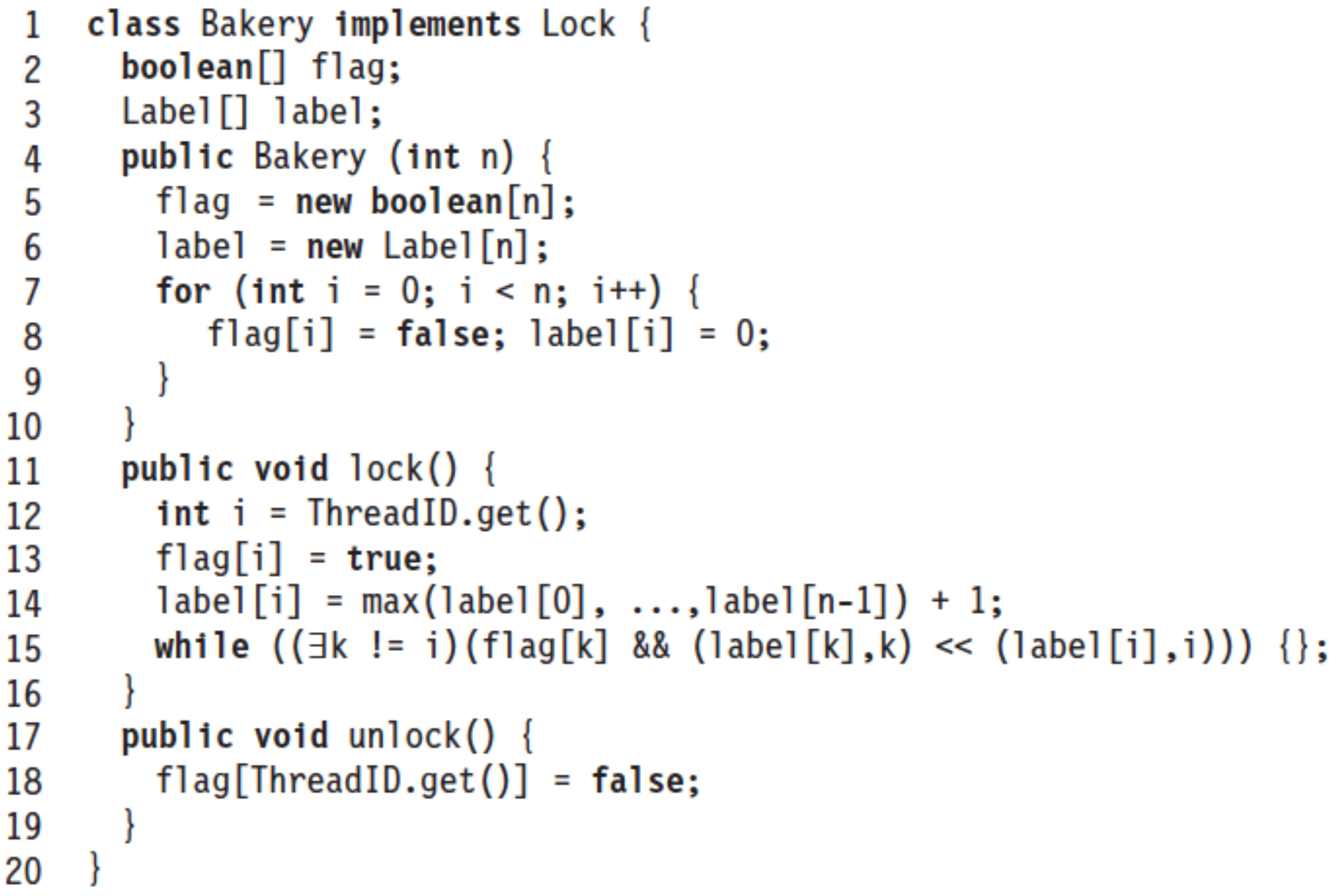
```
class LockTwo implements Lock {
  private int victim;
  public void lock() {
    int i = ThreadID.get();
    victim = i;                  // let the other
    while (victim == i) {}       // wait
  }
  public void unlock() {}
}
```



```
class Peterson implements Lock {
  // thread-local index, 0 or 1
  private boolean[] flag = new boolean[2];
  private int victim;
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    flag[i] = true;             // I'm interested
    victim = i;                 // you go first
    while (flag[j] && victim == i) {}; // wait
  }
  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;            // I'm not interested
  }
}
```



```
class Filter implements Lock {
  int[] level;
  int[] victim;
  public Filter(int n) {
    level  = new int[n];
    victim = new int[n]; // use 1..n-1
    for (int i = 0; i < n; i++) {
      level[i] = 0;
    }
  }
  public void lock() {
    int me = ThreadID.get();
    for (int i = 1; i < n; i++) { // attempt level i
      level[me] = i;
      victim[i] = me;
      // spin while conflicts exist
      while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
    }
  }
  public void unlock() {
    int me = ThreadID.get();
    level[me] = 0;
  }
}
```



```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;
  public Bakery (int n) {
    flag  = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ...,label[n-1]) + 1;
    while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
  }
  public void unlock() {
    flag[ThreadID.get()] = false;
  }
}
```