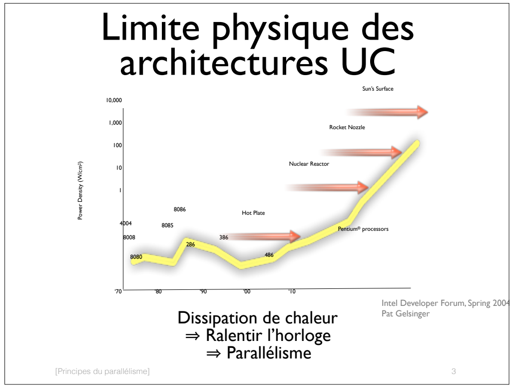
Algorithmes concurrents à grain fin

Les principes du parallélisme

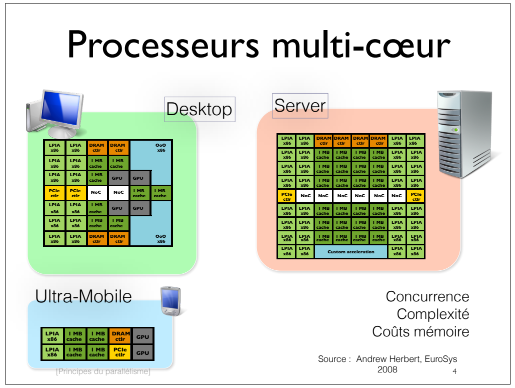
Marc Shapiro
INRIA & LIP6

Architectures modernes de processeurs et parallélisme



Deux mouvements de fond :

- rapprocher la mémoire du processeur ==> caches, réplicats ==> problème de cohérence (très similaire à ce que nous venons de voir)
- plus récemment : remplacer la course à la vitesse d'horloge par la course au parallélisme



What do you do with all these transistors? Well, you can make part of it into memory, perhaps, but what will really happen is we'll take and build a lot of specialty processes. If you look at a personal computer today, in fact, it is an assemblage of a bunch of specialty computers. There's little embedded ones in the disk drives, and in the audio subsystems, and many other parts of the system, and then there's the main CPU, and the graphics core. In a sense, all those things are going to shrink into some type of distributed, multiple-core, architecture, and they'll all get miniaturized and put on a single die.

If you buy into this strategy, the world simplifies in one sense, in that how we build server products, and desktop products, and even ultra-mobile, or telephone mobile handset products, will architecturally be more uniform than it is today. The challenge is, we don't have an architecture that allows us to naturally scale our software across all that, and in particular, we don't know how to write applications, other than in the scale out world of the server, and the data center, that will benefit from all this parallel execution capability.

So our challenge, as a company, and as an industry, is going to be to overcome these two problems, concurrency, and complexity. There are really three C words that I've cared about for the last five or six years, these are two of them, concurrency and complexity, the third one I'll talk about in a minute, is composability. In a sense, this transition that's being forced upon us, whether we like it or not, by the evolution of the microprocessor, I think is actually a good thing, because it's going to force us to create a solution not just to the challenge of programming the many-core machines, but it's going to give us an opportunity at the same time to address these incredible challenges of complexity.

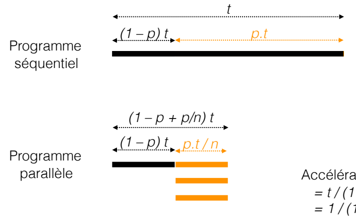
Révolution du multi-cœur

Des millions de processeur par personne
- Comment est-ce qu'on va programmer ça ?

Concurrence très complexe !
- Très difficile à raisonner, programmer, déboguer
- Pas nouveau, mais c'était affaire de spécialistes
- Démocratisation du parallélisme : tous les développeurs sont concernés !
  - *Système, stockage, applications, web, etc.*

Recherche : rendre plus accessible, efficacité
- Primitives, abstractions
- Problème : interférence, non compositionnel

Amdahl's law



Accélération *(speedup)*
= t / (1 – p + p/n)t
= 1 / (1 – p + p/n)

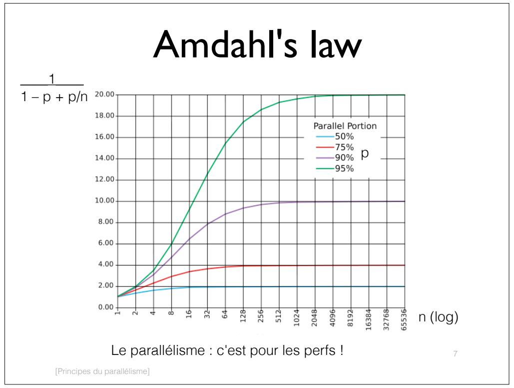
Le parallélisme : c'est pour les perfs !

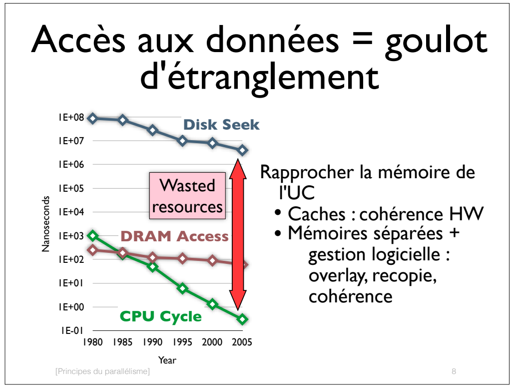
Le parallélisme pour les perfs = que se passe-t-il si je parallélise un programme séquentiel ? Disons que le programme séquentiel s'exécute en *x* unités de temps, et que je parallélise une fraction $p \times x$ unités. Pour simplifier, supposons qu'il n'y a aucun surcoût (parallélisme parfait, coût zéro de synchronisation et de passage séquentiel/parallèle/séquentiel)

n = nb de processeurs
p = proportion parallèle du programme
temps de la partie parallèle = p/n
temps de la partie séquentielle = 1-p. Temps total = 1-p+p/n
Speedup = 1/ temps total

Ex: si 10% du programme est séquentiel, avec 16 processeurs, maxi = 5 fois
Il faut éliminer le goulot d'étranglement séquentiel

The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20x as shown in the diagram, no matter how many processors are used.





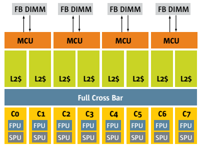
Le parallélisme pour les perfs = que se passe-t-il si je parallélise un programme séquentiel ? Disons que le programme séquentiel s'exécute en *x* unités de temps, et que je parallélise une fraction *pxx* unités. Pour simplifier, supposons qu'il n'y a aucun surcoût (parallélisme parfait, coût zéro de synchronisation et de passage séquentiel/parallèle/séquentiel)

n = nb de processeurs
p = proportion parallèle du programme
temps de la partie parallèle = p/n
temps de la partie séquentielle = 1-p. Temps total = 1-p+p/n
Speedup = 1/ temps total

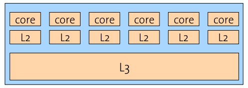
Ex: si 10% du programme est séquentiel, avec 16 processeurs, maxi = 5 fois
Il faut éliminer le goulot d'étranglement séquentiel

The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20x as shown in the diagram, no matter how many processors are used.

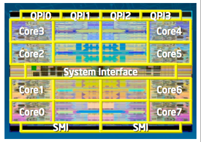
Diversité des architectures



Sun Niagara T2



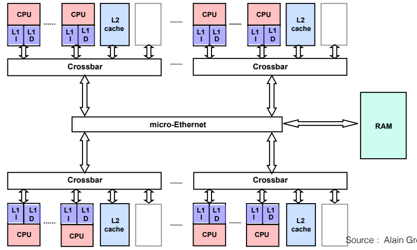
AMD Opteron Istanbul



Intel Nehalem Beckton

Compromis granularité / communication
Processeurs hétérogènes
Portabilité des performances ???

Interconnect : Tsar



Source : Alain Greiner

L2 partagé, L2 séparé
Plusieurs niveaux : cache chip, cache cluster, etc.
Change d'un constructeur à l'autre, d'une année à l'autre !!
Problème : portabilité des performances = ne pas optimiser trop tôt pour une architecture (elle changera l'an prochain !)

On en assemble plusieurs

Processus ≠ fils d'exécution

Processus Unix/Windows
- Espace d'adressage propre
- Exécution séquentielle *main*
- Communication : tuyau, signaux, sockets

Fil d'exécution *(thread)* *ci-après appelé processus* !
- Plusieurs fils à l'intérieur d'un même espace
- Communication par écriture / lecture mémoire
- Interférence

Chaque processus ou fil s'exécute à sa propre vitesse (sauf primitive de synchronisation)

Attention au contexte d'utilisation. Le mot processus désigne un fil d'exécution parallèle aux autre processus, mais le vocabulaire n'est pas standardisé
- Un “processus” Unix ou Windows possède son espace d'adressage propre, disjoint des autres processus
- À l'intérieur d'un “processus” Unix, il peut y avoir plusieurs *threads* (fils d'exécution) partageant sa mémoire

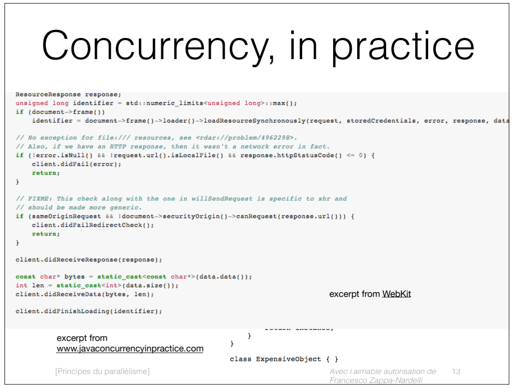
En-dehors du contexte Unix/Windows, le mot processus peut désigner l'une ou l'autre de ces abstractions

Difficulté intellectuelle

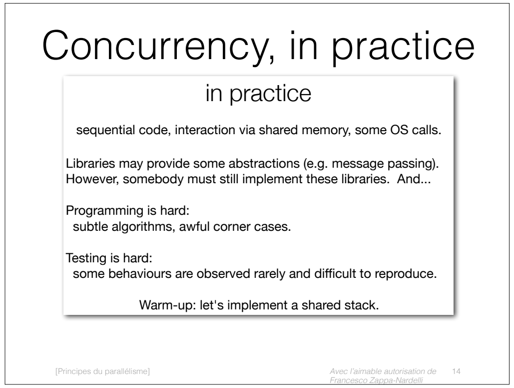
Programmer parallèle c'est dur :
- Interférence entre processus : considérer tous les entrelacements possibles
- Abstractions non composables : raisonnement non local
- Synchronisation coûteuse
- Association verrou-objet non documentée
- Parallélisme caché (bibliothèques, exceptions)

Il faut que ce soit correct !
- Pas de solution-miracle (¬ parallélisation automatique)

En séquentiel, objets sont composables : isolés entre eux, on peut raisonner localement
Isoler threads ==> synchro ==> Amdahl, perfs-- ; coût



On trouve du code concurrent de plus en plus partout :
- dans le noyau
- dans la machine virtuelle java
- dans le browser
- etc.



On trouve du code concurrent de plus en plus partout :
- dans le noyau
- dans la machine virtuelle java
- dans le browser
- etc.

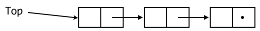
Example: stack

We implement a stack using a list living in the heap:

- each entry of the stack is a record of two fields:

```
typedef struct entry { value hd; entry *tl } entry
```

- the top of the stack is pointed by `Top`.



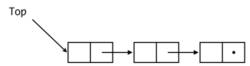

```
pop () {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```


A sequential stack: demo

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

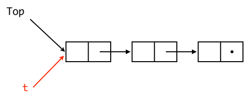
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: pop ( )

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

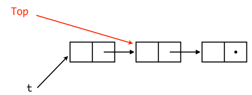
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: pop ( )

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

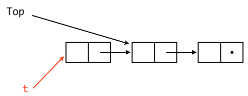
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: pop ( )

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

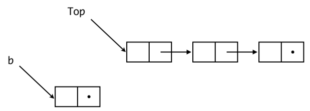


Avec l'aimable autorisation de Francesco Zappa-Nardelli

A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

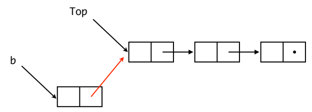


Avec l'aimable autorisation de Francesco Zappa-Nardelli

A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

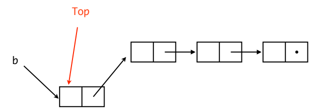
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

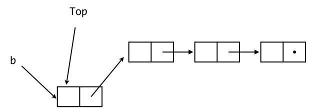
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



Example: shared stack

Notation

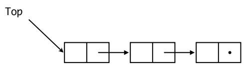
- local to thread: `x, y, …`
- heap: `Top, H, …` (shared between threads)
- data structures: arrays `H[i]`, records `n = t->tl, …`

A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

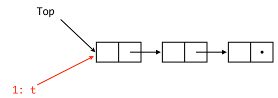


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

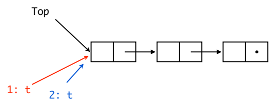


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

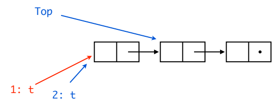


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...



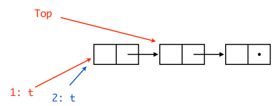
A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

...the two threads might pop the same entry!



Compare-and-Swap

Atomic *compare-and-swap* operation:

```
bool CAS (value_t *addr, value_t exp, value_t new) {
  atomic {
    if (*addr == exp) then {*addr = new; return true;}
    else return false;
  }}
```


CAS = CMPXCHG on x86
explain “atomic”
if CAS fails, loop and try again! busy wait / attente active…

Treiber Stack: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

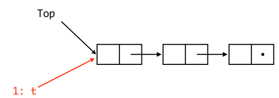
```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```


Treiber Stack: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Two concurrent `pop()` now work fine...

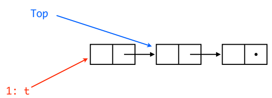


Treiber Stack: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Two concurrent `pop()` now work fine...



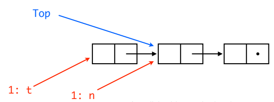
Treiber Stack: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Two concurrent `pop()` now work fine...

The `CAS` of Th. 1 fails.



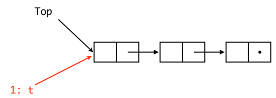
CAS atomically does a read, a test (is the value read the one expected?), and a write, and returns the old value.
If the CAS fails, retry [note *while(true)*]

The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1 starts popping...



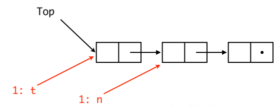
But there still is a problem: the ABA problem

The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1 starts popping but doesn't reach CAS yet



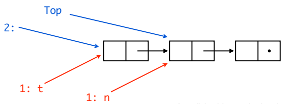
But there still is a problem: the ABA problem

The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pops the top node, now unused



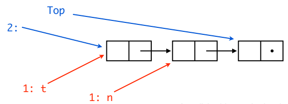
But there still is a problem: the ABA problem

The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pops again...



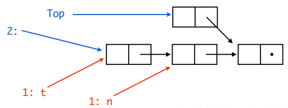
But there still is a problem: the ABA problem

The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pushes a new node...

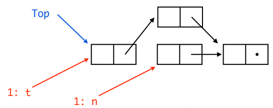


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pushes a new node, happens to re-use the old head of the stack...

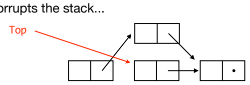


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1's CAS succeeds! and corrupts the stack...



The *hazard pointer* methodology

Maged Michael's *global array* `H` *of hazard pointers*:

- thread `i` alone is allowed to write to element `H[i]` of the array;
- any thread can read any entry of `H`.

The previous algorithm is then modified:

- before popping a cell, a thread puts its address into its own element of `H`. This entry is cleared only if `CAS` succeeds or the stack is empty;
- before pushing a cell, a thread checks to see whether it is pointed to from any element of `H`. If it is, push is delayed.

The CAS aims to check if *Top* has changed, but was fooled by accessing the same node again (ABA). I.e., *Top* changed once and changed back to the initial value; CAS cannot distinguish this from *Top* not changing at all. (Note that the alternative LL/SC does not have this issue; it actually checks if modified)

Solution: explicit check for ABA

Michael's algorithm, simplified

```
pop ( ) {
  while (true) {
    atomic { t = Top;
             H[tid] = t; };
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```


Avec l'aimable autorisation de Francesco Zappa-Nardelli

ABA happens when a pop **t* is stalled before its CAS, a pop concurrently removes **t*, then there is a later (sequentially) *push(t).*

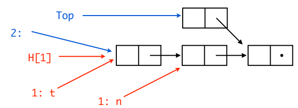
The hazard check (push / in blue) is not atomic with the CAS that follows, is this correct? Yes because of the above sequence. If *push* is stalled after the check, any pop would be concurrent, and argument *b* could not be the element being popped.

Michael's algorithm, simplified

```
pop ( ) {
  while (true) {
    atomic { t = Top;
             H[tid] = t; };
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```


Th 2 cannot push the old head, because Th 1 has an hazard pointer on it...



Avec l'aimable autorisation de Francesco Zappa-Nardelli

ABA happens when a pop **t* is stalled before its CAS, a pop concurrently removes **t*, then there is a later (sequentially) *push(t).*

The hazard check (push / in blue) is not atomic with the CAS that follows, is this correct? Yes because of the above sequence. If *push* is stalled after the check, any pop would be concurrent, and argument *b* could not be the element being popped.

Key properties of Michael's simplified algorithm

- A node can be added to the hazard array only if it is reachable through the stack;
- a node that has been popped is not reachable through the stack;
- a node that is unreachable in the stack and that is in the hazard array cannot be added to the stack;
- while a node is reachable and in the hazard array, it has a constant tail.

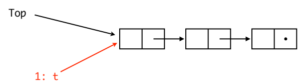
These are a good example of the properties we might want to state and prove about a concurrent algorithm.

The role of *atomic*

```
pop ( ) {
  while (true) {
    t = Top;
    H[tid] = t;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

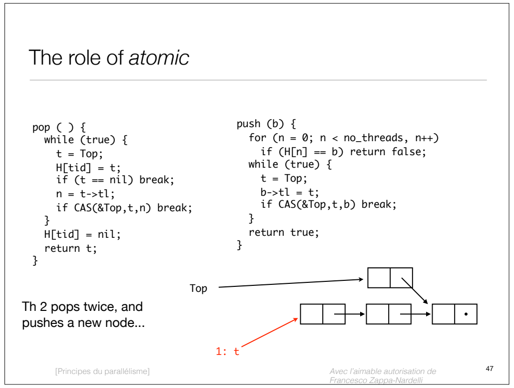
Th 1 copies Top...



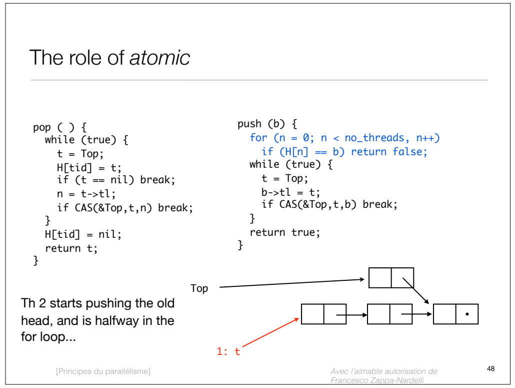
ABA happens when a pop **t* is stalled before its CAS, a pop concurrently removes **t*, then there is a later (sequentially) *push(t).*

The hazard check (push / in blue) is not atomic with the CAS that follows, is this correct? Yes because of the above sequence. If *push* is stalled after the check, any pop would be concurrent, and argument *b* could not be the element being popped.

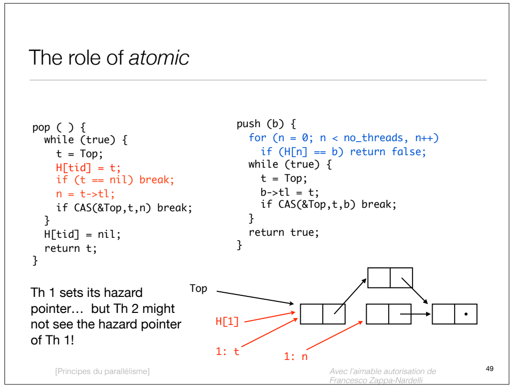
The two assignments at the top of the *pop* loop should be atomic, i.e. < H[tid]≔t≔Top>, otherwise the hazard pointer might be incorrect. However this is not possible (except maybe on machines that support DCAS), two separate assignments are necessary. The problem occurs even if you swap the two assignments.



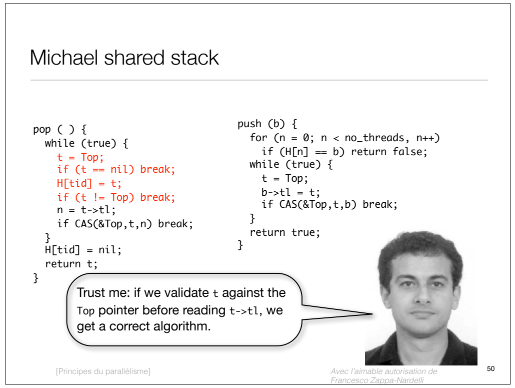
The two assignments at the top of the *pop* loop should be atomic, i.e. < H[tid]:=t:=Top>, otherwise the hazard pointer might be incorrect. However this is not possible (except maybe on machines that support DCAS), two separate assignments are necessary. The problem occurs even if you swap the two assignments.



The two assignments at the top of the *pop* loop should be atomic, i.e. < H[tid]:=t:=Top>, otherwise the hazard pointer might be incorrect. However this is not possible (except maybe on machines that support DCAS), two separate assignments are necessary. The problem occurs even if you swap the two assignments.



The two assignments at the top of the *pop* loop should be atomic, i.e. < H[tid]≔t≔Top>, otherwise the hazard pointer might be incorrect. However this is not possible (except maybe on machines that support DCAS), two separate assignments are necessary. The problem occurs even if you swap the two assignments.



The two assignments at the top of the *pop* loop should be atomic, i.e. < H[tid]≔t≔Top>, otherwise the hazard pointer might be incorrect. However this is not possible (except maybe on machines that support DCAS), two separate assignments are necessary. The problem occurs even if you swap the two assignments.

Solution: assign then check. If *Top* has not changed in the meantime, then we know that H[tid]=t.

Convince yourself! Better: prove it correct! We will see techniques for that.

Michael shared stack

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    H[tid] = t;
    if (t != Top) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

HOW CAN WE BE SURE?

Avec l'aimable autorisation de Francesco Zappa-Nardelli

Java et programmation concurrente

The two assignments at the top of the *pop* loop should be atomic, i.e. < H[tid]:=t:=Top>, otherwise the hazard pointer might be incorrect. However this is not possible (except maybe on machines that support DCAS), two separate assignments are necessary. The problem occurs even if you swap the two assignments.

Solution: assign then check. If *Top* has not changed in the meantime, then we know that H[tid]=t.

Convince yourself! Better: prove it correct! We will see techniques for that.

Lancer / attendre processus

```
Thread [] t = new Thread [...];
for (int i = ...) {
   final String msg = "fil numéro " + i;
   t[i] = new Thread (new Runnable () {
      public void run () {
         ... .println (msg);
      }
   });
}
for (int i = ...)
      t[i].start ();
for (int i = ...)
      t[i].join ();
```


Créer un tableau de threads (qui héritent de runnable et réalisent run())
Les lancer
Attendre qu'ils soient terminés

classe anonyme (non nécessaire mais plus pratique)

Synchronisation

```
public interface Lock {
   void lock ();            // bloquer appelant si non libre
   void unlock ();          // relâcher verrou
   ...
}
/* Utilisation */
public class Counter {
   private long value; private Lock mutex;
   public long getAndIncrement () {
      long tmp;
      mutex.lock ();
      try {
         tmp = value; value = tmp+1;
      } finally { // toujours utiliser finally
         mutex.unlock ();
      }
   }
   return tmp;
}
```


Quand on prend un verrou, toujours suivi d'un try...finally{unlock}. Cela assure que le verrou sera relâché, quelle que soit la façon dont on sort de la méthode (retour, fin d'exécution, ou exception).

"synchronized" == lock à l'appel + finally unlock

Variables locales à un fil (1)

```
interface ThreadLocal<T> {
   public ThreadLocal ();
   protected T initialValue ();
   public T get();
   public void set(T);
}
class ThreadLocalID
      extends ThreadLocal <Integer> {
         protected synchronized Integer
             initialValue () {…; return …}
}
```


Variables locales à un fil (2)

création dans chaque fil

anonymous inner class

initialisatio

```
public class UniqueThreadID {
   private static AtomicInteger
      nextID = new AtomicInteger(0);
   private static final ThreadLocal<Integer>
      myID = new ThreadLocal<Integer>() {
                     protected Integer initialValue() {
                          return nextID.getAndIncrement();
                } };
   public static int getCurrentThreadId() {
       return myID.get();
   } }
```


ThreadLocal<T>
- ThreadLocal ()
- initialValue ()
- T get()
- set(T)

Une variable locale à un processus est allouée statiquement (≠malloc, ≠pile) dans une zone privée à ce processus.

ThreadLocal<T>
- ThreadLocal ()
- initialValue ()
- T get()
- set(T)

Attention, danger !

Modèle de mémoire Java :

- En-dehors des blocs "synchronized" les lectures-écritures peuvent s'exécuter dans le désordre

Variable partagée entre processus:

```
volatile int i;
```

⇒ get/set linéarisables

- Coûteux

Règles complexes
volatile =

- Pas d'optimisations
- Barrières avant lecture / après écriture

TAS = getAndSet

Toutes machines Intel

```
class AtomicBoolean {
  private boolean b;
  public synchronized boolean getAndSet () {
      // comme si !!!
      boolean tmp = b;
      b = true;
      return tmp;
  }
  public synchronized void set (boolean v) {
      // comme si !!!
      b = v;
  }
}
```


version Java du Test-And-Set (TAS)
La spécification du TAS est équivalente à ce code, mais la réalisation est une unique instruction atomique du matériel. C'est ce qui fait son intérêt.

TAS = getAndSet

```
class AtomicReference<T> {
   private T ref;
   public synchronized T getAndSet (T newRef) {
         // comme si !!!
         T tmp = ref;
         ref = newRef;
         return tmp;
   }
   public synchronized void set (T newRef) {
         ref = newRef;
    }
}
```


version Java du Test-And-Set (TAS)
La spécification du TAS est équivalente à ce code, mais la réalisation est une unique instruction atomique du matériel. C'est ce qui fait son intérêt.

CAS = compareAndSet

Machines Intel récentes

```
class AtomicReference<T> {
   private T ref;
   public T getAndSet () { … } // comme ci-dessus
   public void set (T r)  { … } // comme ci-dessus
   public boolean synchronized
   compareAndSet (T expectedRef, T newRef) {
         // comme si !!!
         if (ref == expectedRef) {
             ref = newRef;
             return true;
         } else
             return false; // conflit ⇒ boucler
   }
}
```


version Java du Compare-And-Swap (CAS)

La spécification du CAS est équivalente à ce code, mais la réalisation est une unique instruction atomique du matériel. C'est ce qui fait son intérêt.

La classe AtomicReference (générique : réf à un objet de classe T) offre CAS et TAS.

CAS en C

attention en Java ref est

```
atomic bool
   compare_and_swap (void** ref,
                     void* expectedRef,
                     void* newRef) {
      // comme si !!!
      if (*ref == expectedRef) {
         ref = newRef;
         return true;
      } else
         return false; // conflit ⇒ boucler
   }
}
```


DCAS

Accès atomique à plusieurs mots
- DCAS = double CAS
- architecture 680x0
- ≠ double-width CAS

Mais :
- Coûteux en matériel
- Émulation logicielle
- Pourquoi deux et pas trois, quatre, … ?
- Mémoire transactionelle

Attention en Java la réf est implicite, ici elle est explicite

AtomicMarkableReference

```
class AtomicMarkableReference<T> {
   private T ref;
   private boolean mark;
   public synchronized boolean
         compareAndSet (T expectedRef, T newRef,
                     boolean expectedMk, boolean newMk) {
         if (ref == expectedRef && mark == expectedMk) {
            ref = newRef; mark = newMark;
            return true;
         } else
            return false;  // conflit !
   } }
   public synchronized boolean
set (T expectedRef, boolean mk) { … }
…
```


L'instruction DCAS n'existe pas sur les architectures courantes.
Pourtant on a souvent 2 valeurs à mettre à jour de façon atomique.
Dans le cas pointeur + booléen, un pis-aller est de les empaqueter en un seul mot, qui sera accédé par CAS.

AtomicMarkableReference

```
…
   public synchronized T getReference () { … }
   public synchronized boolean isMarked () { … }
   public synchronized T get (boolean [] mk) {
      mk [0] = mark;
      return ref;
   }
   public synchronized boolean
      attemptMark (T expectedRef, boolean mk) {
         if (ref != expectedRef) return false;
         mark = mk; return true;
      }
   }
```


get doit renvoyer deux valeurs : la réf et le bit de marquage. Comme c'est impossible en Java on triche en passant un tableau sur lequel on fera un effet de bord.

Load-Linked/Store-Conditional

Machines PowerPC
Permet le consensus à plusieurs processus
Même puissance que CAS
Plus facile à implémenter
Pas de problème ABA

Adresse mémoire surveillée *x*

Load-Link (x)

- La machine enregistre l'adresse *x*

Store-Conditional (x, y)

- *x := y, return true*
  uniquement si *x* n'a pas été modifié
  depuis *Load-Link(x)*
- *return false*
  sinon, ou même sans raison valable

Mémoire transactionnelle

```
atomic {
  x := 10
  y := z
  if (x+y < 0) retry
}
```

Bloc atomique d'instructions machine :

- ACID
- Si conflit (lecture/écriture par deux blocs concurrents) annulation & boucle
- Intercepter, journaliser tous accès mémoire, vérifier,

Software TM / *Hardware TM* / *Hybrid TM*

Primitives de synchronisation

Registres lecture-écriture

Synchronisation avec des mémoires simples lecture-écriture

- Pour simplifier (beaucoup) supposons que les accès à chaque case mémoire sont linéarisables

```
volatile int i;         // Java uniquement !
volatile boolean b;     // Java uniquement !
```

Synchronisation de base: attente active. *(Si on attend longtemps, mieux vaut passer la main au scheduler.)*

Verrou de Peterson

```
/* Synchro entre DEUX processus, pas plus ! */
public class Peterson implements Lock {
    // threadID = 0 ou 1
    private volatile boolean flag[] = new boolean [2]; //partagé
    private volatile int victim;                       //partagé
    public void lock () {
        int me = ThreadID.get ();
        int other = 1-me;
        flag [me] = true;
        victim = me;
        while (flag[other] && victim == me) {}; // attente active
    }
    public void unlock () {
        int me = ThreadID.get ();
        flag [me] = false;
    }
}
```


partagé = entre les deux processus
flag = je suis intéressé
victim = je laisse passer l'autre
lock :: pre (initialement): flag = {false, false }
lock :: post: no more than one thread holds the lock

rely:
- other thread does not change flag[me]
- victim = me or other

guarantee: I do not change flag[other]
- victim = me or other

Peterson [IPL 1981]

Algorithme de la boulangerie

```
public class Bakery implements Lock {
    volatile boolean [] flag;
    volatile int [] ticket;
    public Bakery (int n) {  // n processus
        flag = new boolean [n]; ticket = new int [n];
        for (int i = 0; i<n; i++) { flag[i] = false; ticket[i] = 0; }
    }
    public void unlock() {
        flag [ThreadID.get()] = false;
    }
    public void lock() {
        int me = ThreadID.get();
        flag [me] = true;
        ticket [me] = max (ticket) + 1; //attention partagé !
        while (∃k≠me : flag[k]
                    && (ticket[k] < ticket[me]
                        ||  (ticket[k] == ticket[me] && k < me))
                    {};      // attente active
        }
    }
}
```


n threads

flag = je suis intéressé
ticket: mon numéro d'ordre de passage
lock :: pre (initialement): flag = false; ticket = {0, 0, … }
lock :: post: no more than one thread holds the lock

rely:
- other thread does not change flag[me], ticket[me]
- k≠me (on ne suppose pas que ticket[me] est unique)
- tickets alloués en gros dans l'ordre d'arrivée

guarantee: I do not change flag[other], ticket[other]

Service FIFO (en gros)
Utilisation mémoire : proportionnelle au nombre max de processus
Ne convient pas pour allocation dynamique de fil, grand nombre de processeurs
Très simplifié par rapport à l'original de Lamport [CACM 1974]