TECHNIQUES MODERNES DE PROGRAMMATION CONCURRENTE

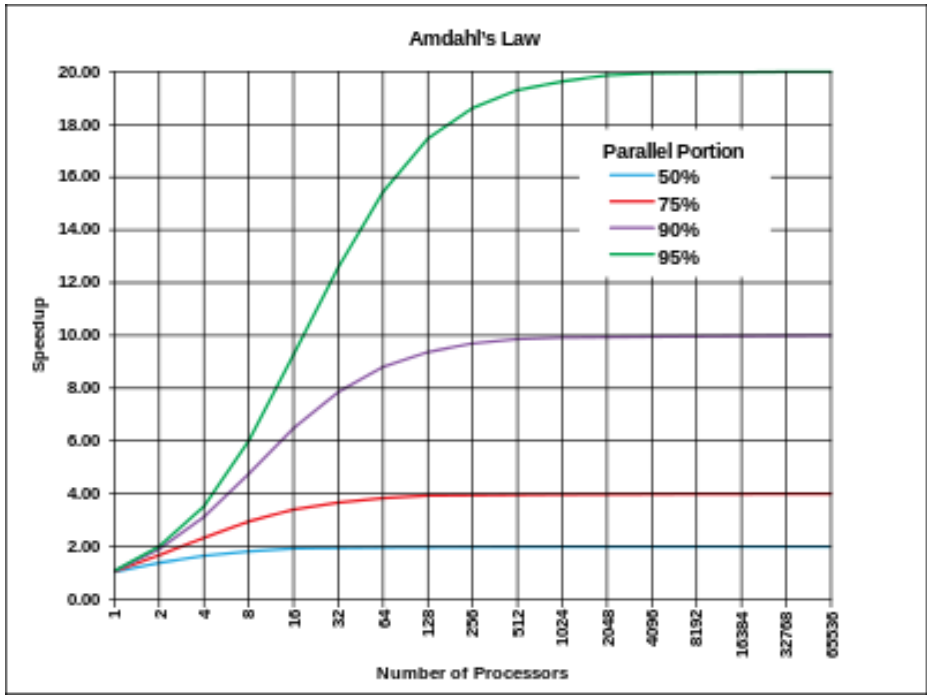
COURS 4
MÉMOIRES TRANSACTIONNELLES

Par Jean-Pierre Lozi
Basé sur les cours de
Gaël Thomas

RAPPEL : LIMITE À L'ACCELÉRATION

- **Limite : la loi d'Amdahl**
- **p : pourcentage du code exécutable en parallèle**
  - (1 – p) : pourcentage du code exécuté en séquentiel
  - p / n : exécution du code parallèle sur n cœurs
  - **Temps d'exécution :** (1 - p) + p / n
- **Accélération maximale théorique :** a = 1/(1 – p + p/n)
  - Limite pour n ⟶ ∞ : a ⟶ 1/(1 – p)
- Application numérique :

p = 0,25 ⇒ a ⟶ 1/0,75 = 4 quand n ⟶ ∞ (3,7 à 32 cœurs)
p = 0,95 ⇒ a ⟶ 1/0,05 = 20 quand n ⟶ ∞ (12,55 à 32 cœurs, 17,42 à 128 cœurs)
p = 0,96 ⇒ a ⟶ 1/0,05 = 25 quand n ⟶ ∞ (14,28 à 32 cœurs : +12.1%, 21,05 à 128 cœurs : +17.2% !)



⇒ Ça vaut le coup de se battre pour paralléliser les quelques pourcents restants!

SOLUTION NATURELLE

Prendre les verrous les plus fins possibles...

```
lock(x)
...
...
...
...
unlock(x)
```

0% en //

```
lock(x)
...
unlock(x)
...
...
lock(y)
...
unlock(y)
```

50% en //

```
CAS(x)
...
CAS(z)
...
...
TAS(y)
```

95% en //

LIMITES DU VERROUILLAGE FIN

- Verrouillage fin complexifie très fortement le code !
  - Difficile à maintenir, à faire évoluer.
  - Bugs (très !) difficiles à trouver.
  - Preuve de programme souvent quasi-impossible à faire (pas de famine, pas de deadlock ?)
  - **Difficile à composer**
  - Effet de bord important !
    - Protocole d'accès à une variable n'existe souvent que dans la tête du programmeur…

LIMITES DU VERROUILLAGE FIN

- Verrouillage fin complexifie très fortement le code !
  - Difficile à maintenir, à faire évoluer.
  - Bugs (très !) difficiles à trouver.
  - Preuve de programme souvent quasi-impossible à faire (pas de famine, pas de deadlock ?)
  - **Difficile à composer**
  - Effet de bord important !
    - Protocole d'accès à une variable n'existe souvent que dans la tête du programmeur…
- Verrouillage fin : reste très pessimiste…
  - Nombre de cœurs augmente ⇒ probabilité de conflit augmente
  - Conflit ⇒ cœurs oisifs…

LES MÉMOIRES TRANSACTIONNELLES

- **Verrouillage pessimiste toujours adéquat à l'heure des multicœurs ?**
  - Certains parlent d'un concept dépassé…

LES MÉMOIRES TRANSACTIONNELLES

- **Verrouillage pessimiste toujours adéquat à l'heure des multicœurs ?**
  - Certains parlent d'un concept dépassé…
- **Pourquoi prend-on des verrous ?**
  - Pour exécuter de façon atomique un ensemble d'instructions…

LES MÉMOIRES TRANSACTIONNELLES

- **Verrouillage pessimiste toujours adéquat à l'heure des multicœurs ?**
  - Certains parlent d'un concept dépassé…
- **Pourquoi prend-on des verrous ?**
  - Pour exécuter de façon atomique un ensemble d'instructions…
- **Proposition des mémoires transactionnelles :**
  - Définir des blocs atomiques
  - Les exécuter de façon optimiste, si conflit, on recommence…

LES MÉMOIRES TRANSACTIONNELLES

- **Verrouillage pessimiste toujours adéquat à l'heure des multicœurs ?**
  - Certains parlent d'un concept dépassé…
- **Pourquoi prend-on des verrous ?**
  - Pour exécuter de façon atomique un ensemble d'instructions…
- **Proposition des mémoires transactionnelles :**
  - Définir des blocs atomiques
  - Les exécuter de façon optimiste, si conflit, on recommence…
- **Avantages attendus :**
  - Simplifier le code : plus de verrous différents acquis dans un certain ordre, juste blocs atomiques
  - Pas de deadlock, pas de famine, composabilité assurée par la plateforme
  - Ne bloquer un cœur que si stictement nécessaire (i.e. car conflit)

LES MÉMOIRES TRANSACTIONNELLES

- Avec mémoires transactionnelles, blocs atomiques :
  - Un bloc atomique est exécuté d'une façon qui apparait atomique…
  - Il peut y avoir des entrelacements s'ils n'ont aucun impact (e.g. variables différentes, ou lecture même var).

LES MÉMOIRES TRANSACTIONNELLES

- Avec mémoires transactionnelles, blocs atomiques :
  - Un bloc atomique est exécuté d'une façon qui apparait atomique…
  - Il peut y avoir des entrelacements s'ils n'ont aucun impact (e.g. variables différentes, ou lecture même var).
- Par exemple :

Initialement : x = 21

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

Deux ordonnancements possibles :
[b,c,d] ***puis*** *[g,h,i] (⇒ 44)* *ou* *[g,h,i]* ***puis*** *[b,c,d] (⇒ 43)*

LES MÉMOIRES TRANSACTIONNELLES

- Passer du verrouillage au transactions, très simple :

```
synchronized(o) {
 if(!x) {
  x = true;
  doSomething();
 }
}
```

0% en //

```
atomic {
 if(!x) {
  x = true;
  doSomething();
 }
}
```

Parallélisme maximal?

LES MÉMOIRES TRANSACTIONNELLES

- Passer du verrouillage au transactions, très simple :

```
synchronized(o) {
 if(!x) {
  x = true;
  doSomething();
 }
}
```

0% en //

```
atomic {
 if(!x) {
  x = true;
  doSomething();
 }
}
```

Parallélisme maximal?

La transaction lit x. Si x a été modifié par un autre thread pendant la transaction (entre première lecture et fin)
⇒ transaction avortée et on recommence

MÉMOIRES TRANSACTIONNELLES : ÉVÉNEMENTS

- Problème : comment implémenter les variables de condition ?

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

MÉMOIRES TRANSACTIONNELLES : ÉVÉNEMENTS

- Problème : comment implémenter les variables de condition ?

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

- Notion de retry [Harris05] :

```
atomic {
 while(!x)
  retry;
}
```

```
atomic {
 x = true;
}
```

MÉMOIRES TRANSACTIONNELLES : ÉVÉNEMENTS

- Problème : comment implémenter les variables de condition ?

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

- Notion de retry [Harris05] :

```
atomic {
 while(!x)
  retry;
}
```

```
atomic {
 x = true;
}
```

Transaction attend une modif sur la variable x (read set)

MÉMOIRES TRANSACTIONNELLES : ÉVÉNEMENTS

- Problème : comment implémenter les variables de condition ?

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

- Notion de retry [Harris05] :

```
atomic {
 while(!x)
  retry;
}
```

Transaction attend une modif sur la variable x (read set)

```
atomic {
 x = true;
}
```

commit : x a été modifié

MÉMOIRES TRANSACTIONNELLES : ÉVÉNEMENTS

- Problème : comment implémenter les variables de condition ?

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

- Notion de `retry` [Harris05] :

```
atomic {
 while(!x)
  retry;
}
```

```
atomic {
 x = true;
}
```

commit : x a été modifié
⇒ réveille toute attente sur x

Transaction attend une modif sur la variable x (read set)

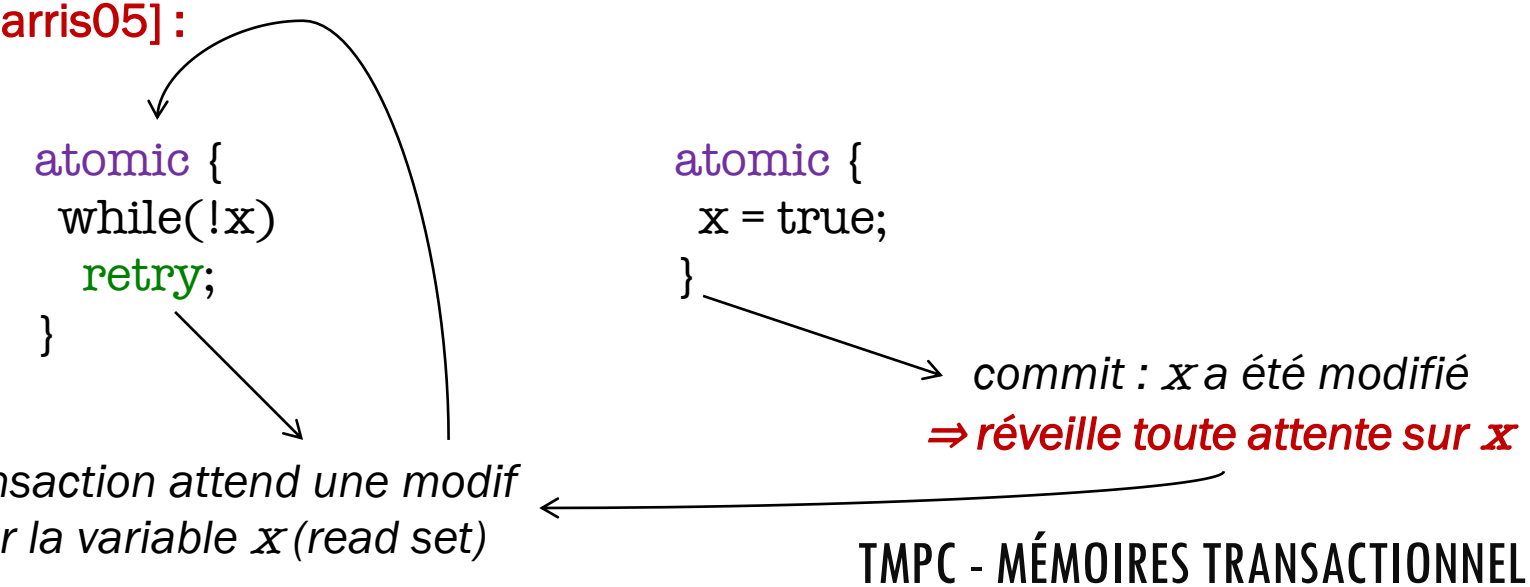
MÉMOIRES TRANSACTIONNELLES : ÉVÉNEMENTS

- Problème : comment implémenter les variables de condition ?

```
synchronized(o) {
 while(!x)
  o.wait()
}
```

```
synchronized(o) {
 x = true;
 o.notify();
}
```

- Notion de retry [Harris05] :



MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Bonne propriété :** si A et B atomiques, on peut les composer dans un autre bloc atomique !

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Bonne propriété :** si A et B atomiques, on peut les composer dans un autre bloc atomique !
- **Un exemple :** la queue

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Bonne propriété :** si A et B atomiques, on peut les composer dans un autre bloc atomique !
- **Un exemple :** la queue

```
atomic move(Queue dst, Queue src) {
  Elmt e = src.deq();
  dst.enq(e);
}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Bonne propriété :** si A et B atomiques, on peut les composer dans un autre bloc atomique !
- **Un exemple :** la queue

```
atomic move(Queue dst, Queue src) {
  Elmt e = src.deq();   ──────────►  atomic Elmt deq()
  dst.enq(e);           ──────────►  atomic void enq(Elmt e);
}
```

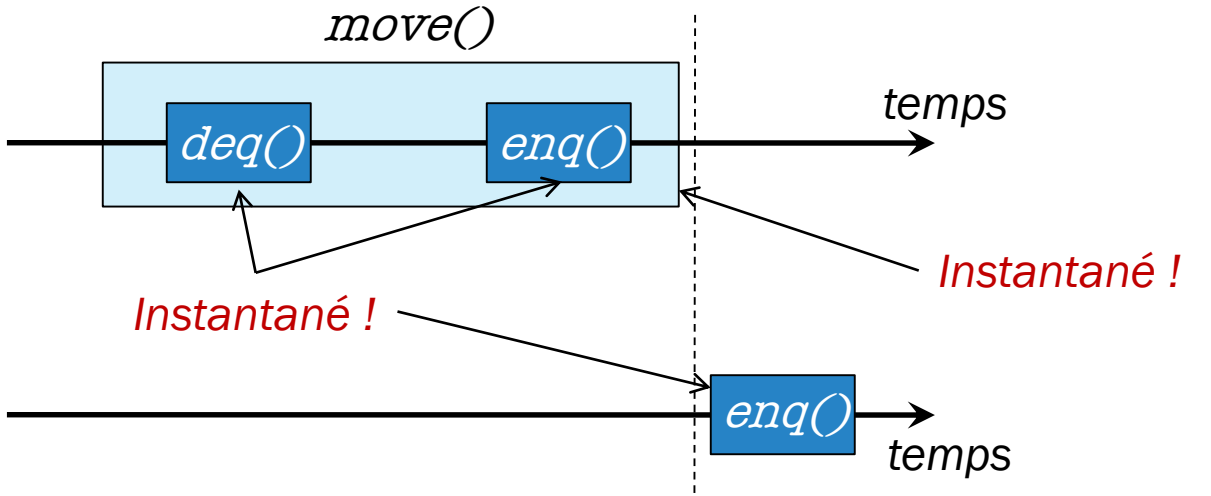
MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Bonne propriété :** si A et B atomiques, on peut les composer dans un autre bloc atomique !
- **Un exemple :** la queue

```
atomic move(Queue dst, Queue src) {
  Elmt e = src.deq();
  dst.enq(e);
}
```

atomic Elmt deq()

atomic void enq(Elmt e);



MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

```
class Queue {

 LinkedList<Elmt> queue;

}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

```
class Queue {

 LinkedList<Elmt> queue;


      void enq(Elmt e) {
       atomic {
        if(queue.size() == MAX_SIZE)
         retry;
        queue.addFirst(e)
       }
      }

}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

```
class Queue {

 LinkedList<Elmt> queue;

      void enq(Elmt e) {
       atomic {
        if(queue.size() == MAX_SIZE)
         retry;
        queue.addFirst(e)
       }
      }

      void deq(Elmt e) {
       atomic {
        if(queue.empty())
         retry;
        return queue.removeLast(e)
       }
      }

}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

```
class Queue {

 LinkedList<Elmt> queue;

      void enq(Elmt e) {
       atomic {
        if(queue.size() == MAX_SIZE)
         retry;
        queue.addFirst(e)
       }
      }

}
```

```
      void deq(Elmt e) {
       atomic {
        if(queue.empty())
         retry;
        return queue.removeLast(e)
       }
      }
```

Attend qu'une des variables lues soit modifiée pour réessayer

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

```
class Queue {

 LinkedList<Elmt> queue;

      void enq(Elmt e) {
       atomic {
        if(queue.size() == MAX_SIZE)
         retry;
        queue.addFirst(e)
       }
      }

      void deq(Elmt e) {
       atomic {
        if(queue.empty())
         retry;
        return queue.removeLast(e)
       }
      }
}
```

addFirst(e) a modifié variable lue dans empty(), après commit, signal !

Attend qu'une des variables lues soit modifiée pour réessayer

Université Nice Sophia Antipolis

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- Supposons que la queue a une taille maximum. Exemple complet :

```
class Queue {

 LinkedList<Elmt> queue;

    void enq(Elmt e) {
     atomic {
      if(queue.size() == MAX_SIZE)
       retry;
      queue.addFirst(e)
     }
    }

    void deq(Elmt e) {
     atomic {
      if(queue.empty())
       retry;
      return queue.removeLast(e)
     }
    }
}
```

addFirst(e) a modifié variable lue dans empty(), après commit, signal !

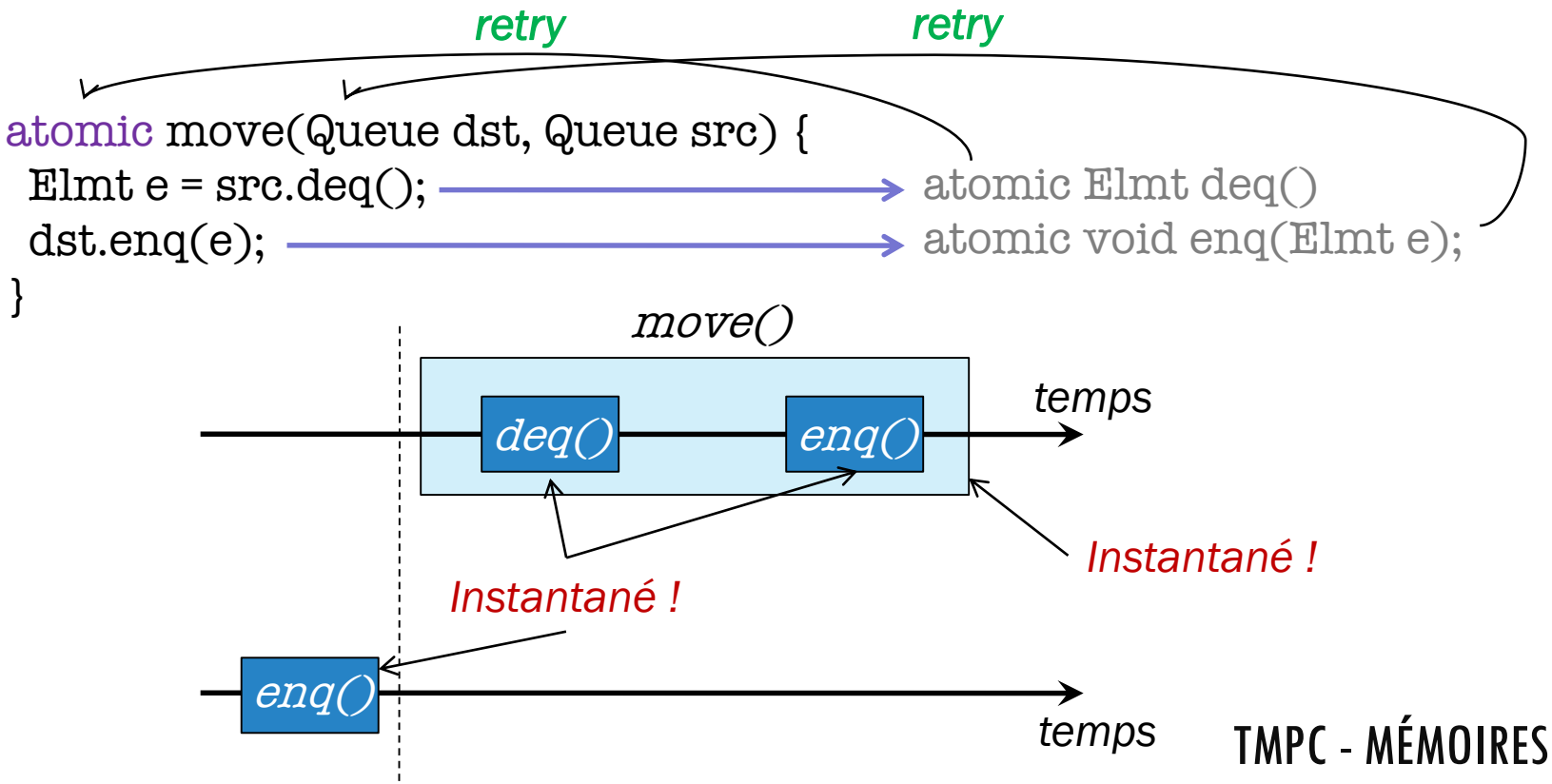
On redémarre !

Attend qu'une des variables lues soit modifiée pour réessayer

Université Nice Sophia Antipolis

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition + attente sur événement :** pour que ça marche, un `retry` fait sortir de la transaction englobante !
- Sinon, un `retry` qui fait échouer `enq()` pourrait mener à des événements entre `deq()` et `enq()`…



MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé `orElse`. Par exemple, dequeue non-bloquant :

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé orElse. Par exemple, dequeue non-bloquant :

```
Elmt deqNoWait() {
 atomic {
  return deq();
 } orElse {
  return null;
 }
}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé orElse. Par exemple, dequeue non-bloquant :

```
Elmt deqNoWait() {
 atomic {
  return deq();          Exécute premier bloc
 } orElse {
  return null;
 }
}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé `orElse`. Par exemple, dequeue non-bloquant :

```
Elmt deqNoWait() {
 atomic {
  return deq();          Exécute premier bloc
 } orElse {              Si retry
  return null;
 }
}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé `orElse`. Par exemple, dequeue non-bloquant :

```
Elmt deqNoWait() {
 atomic {
  return deq();          Exécute premier bloc
 } orElse {              Si retry
  return null;           Abort, puis exécute le second bloc
 }
}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé `orElse`. Par exemple, dequeue non-bloquant :

```
Elmt deqNoWait() {
 atomic {
  return deq();          Exécute premier bloc
 } orElse {              Si retry
  return null;           Abort, puis exécute le second bloc
 }                              Si retry dans le second bloc (impossible ici), on attend un changement
}
```

MÉMOIRES TRANSACTIONNELLES : COMPOSITION

- **Composition par alternative :** possible d'éviter qu'une section atomique avec un retry soit bloquante
  - **Un peu l'idée d'un trylock :** en cas de problème, on abandonne et on fait autre chose
  - Avec les mémoires transactionnelles, ça implique qu'on avorte la transaction avant de faire autre chose
- **Fonctionnement :** mot-clé orElse. Par exemple, dequeue non-bloquant :

```
Elmt deqNoWait() {
 atomic {
  return deq();
 } orElse {
  return null;
 }
}
```

Exécute premier bloc

Si retry

Abort, puis exécute le second bloc

Si retry dans le second bloc (impossible ici), on attend un changement et on recommence la transaction.

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Deux implémentations possibles :** mémoire transactionnelle retardée vs. immédiate

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Deux implémentations possibles :** mémoire transactionnelle retardée vs. immédiate
- **Mémoire transactionnelle retardée (deferred update) :**
  - Ecriture dans une copie locale, propagée en mémoire centrale au commit
  - Du travail lors du commit (il faut refaire toutes les modifs), mais aborts peu coûteux (on efface sa copie locale)
    - *Marche généralement mieux si beaucoup de aborts par rapport aux commits*
  - Cohérence facile à maintenir

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Deux implémentations possibles :** mémoire transactionnelle retardée vs. immédiate
- **Mémoire transactionnelle retardée (deferred update) :**
  - Ecriture dans une copie locale, propagée en mémoire centrale au commit
  - Du travail lors du commit (il faut refaire toutes les modifs), mais aborts peu coûteux (on efface sa copie locale)
    - *Marche généralement mieux si beaucoup de aborts par rapport aux commits*
  - Cohérence facile à maintenir
- **Mémoire transactionnelle immédiate (direct update) :**
  - Ecriture directement dans la mémoire centrale, annulation si transaction avorte
  - Très peu de travail lors du commit, mais aborts coûteux (on doit annuler tous les changements)
    - *Marche généralement mieux si beaucoup de commits par rapport aux aborts*
  - Cohérence plus difficile à maintenir (doit garder les valeurs originales si la transaction avorte)

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

Mémoire centrale :
x = 20

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

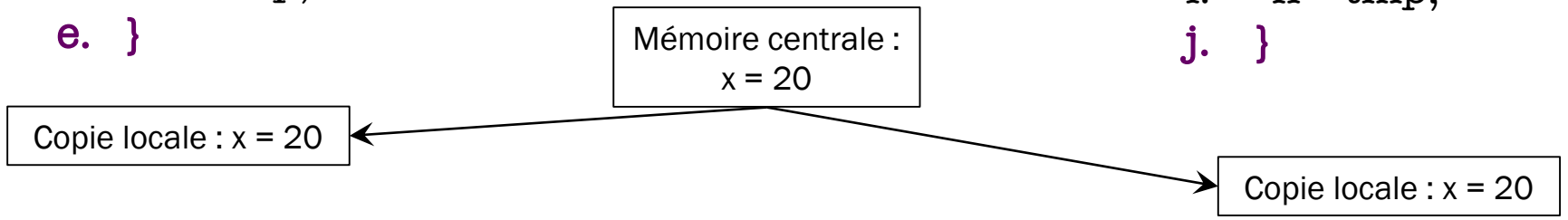
Read set = { x }

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```



Read set = { x }

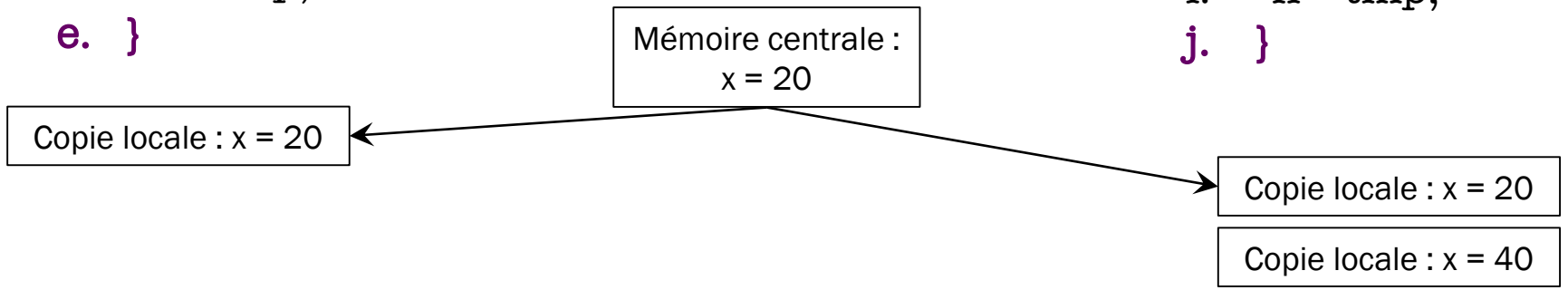
Read set = { x }

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```



Read set = { x }

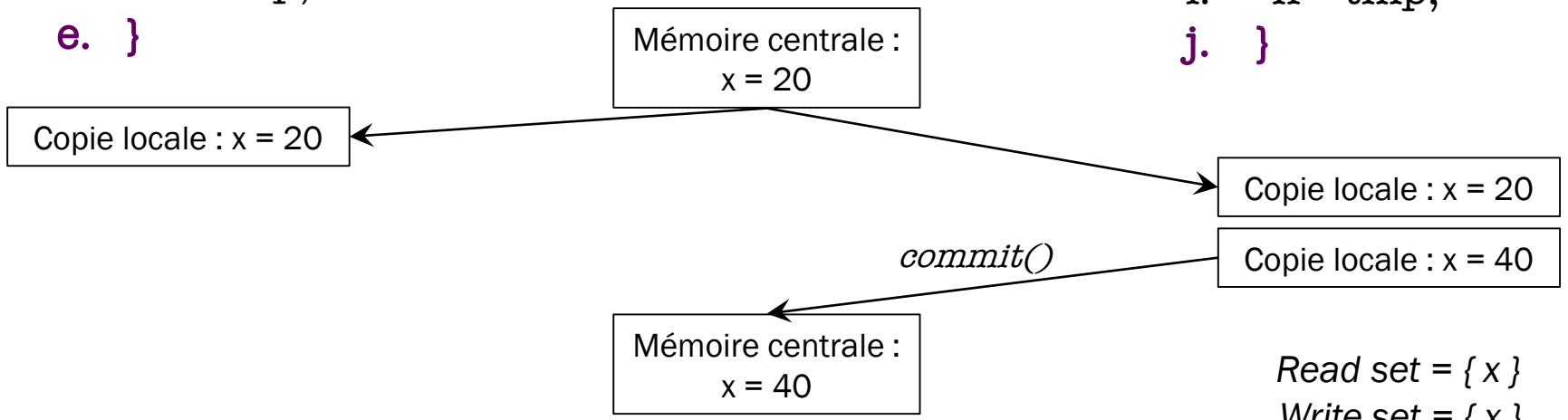
Read set = { x }
Write set = { x }

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

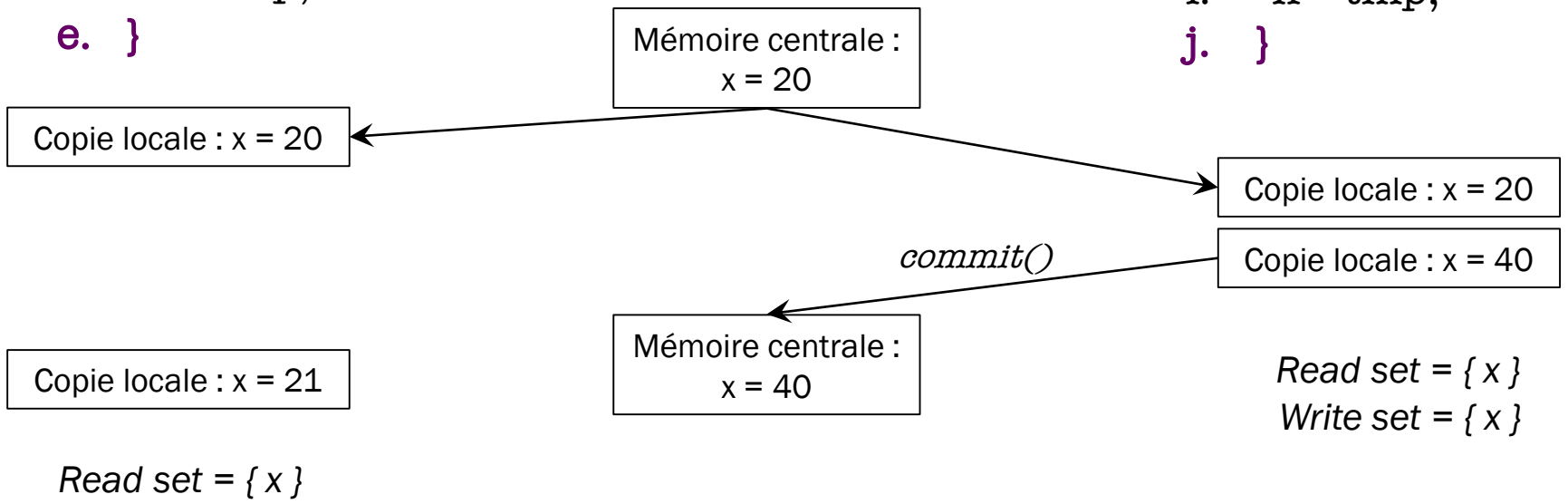


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

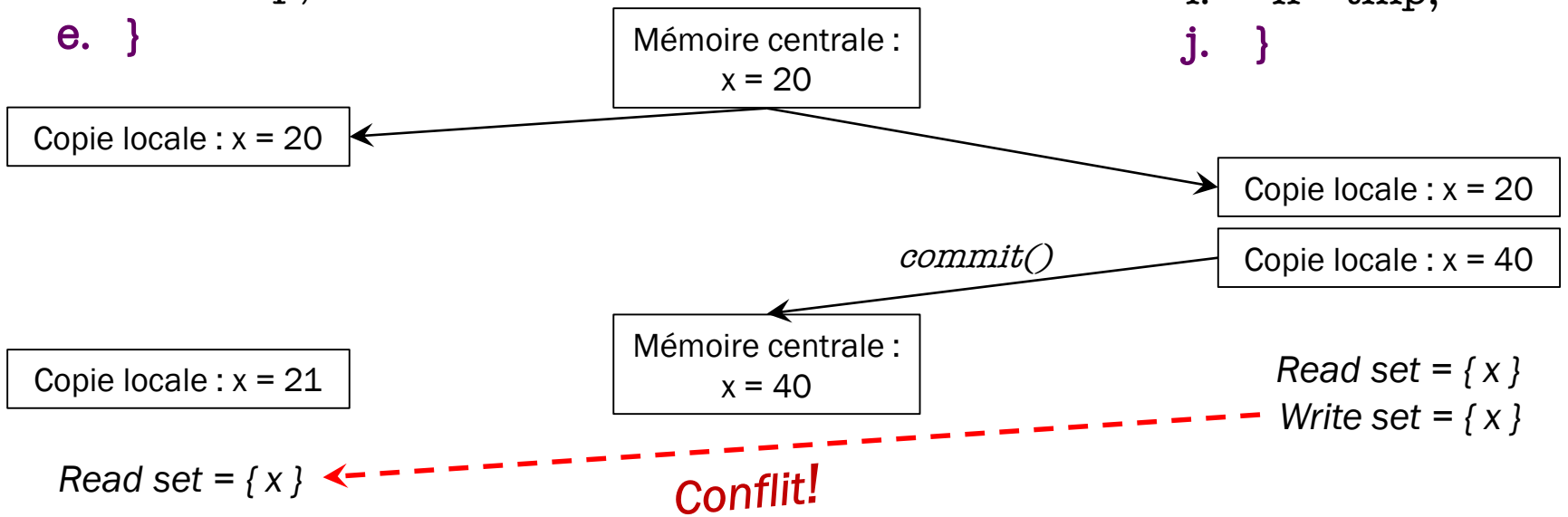


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```

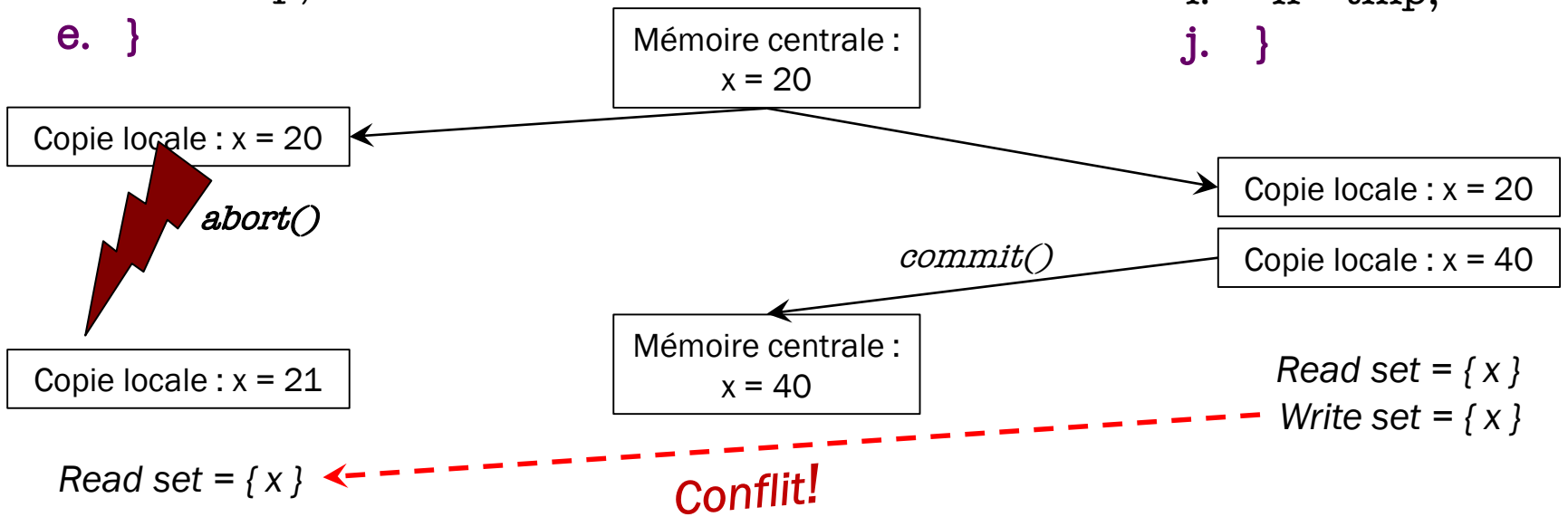


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle retardée (deferred update) :** aborts peu coûteux

```
a. atomic {
b.   tmp = x;
c.   tmp = tmp + 1;
d.   x = tmp;
e. }
```

```
f. atomic {
g.   tmp = x;
h.   tmp = tmp * 2;
i.   x = tmp;
j. }
```



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)

x = 3, y = 20, z = 30

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

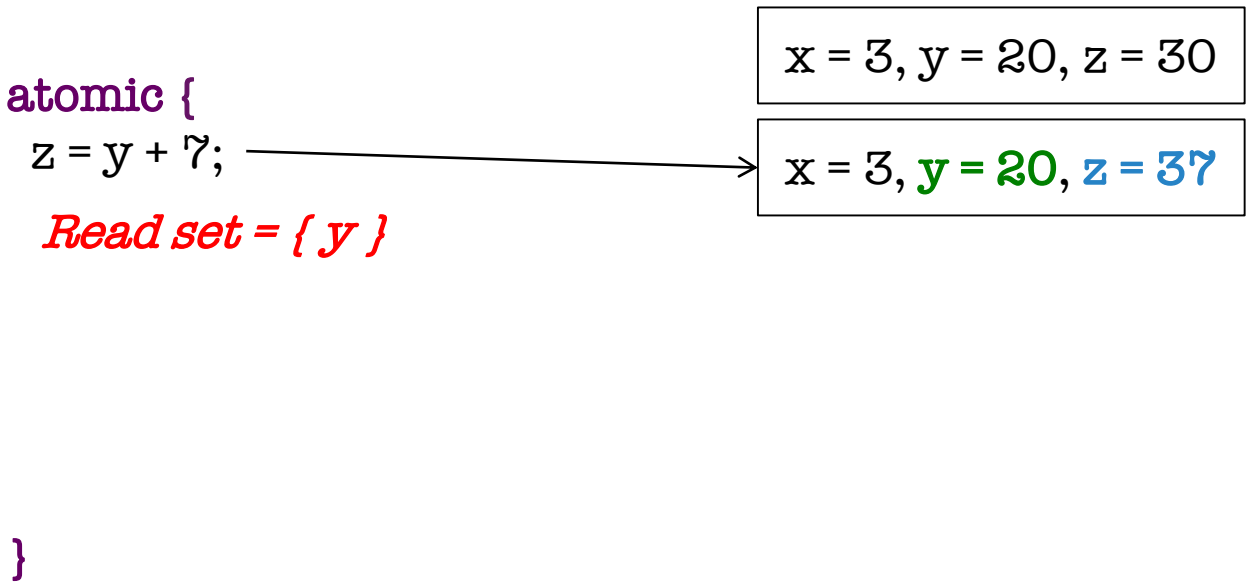
- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)

```
atomic {
  z = y + 7;



}
```

x = 3, y = 20, z = 30

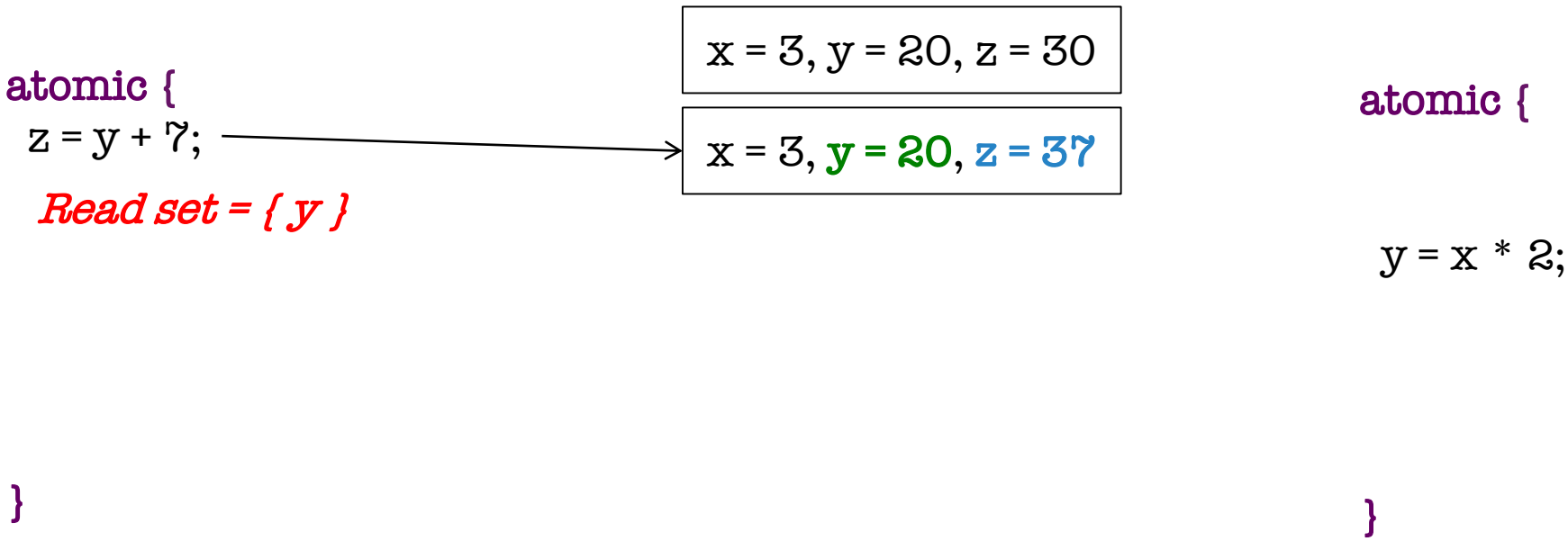
MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)



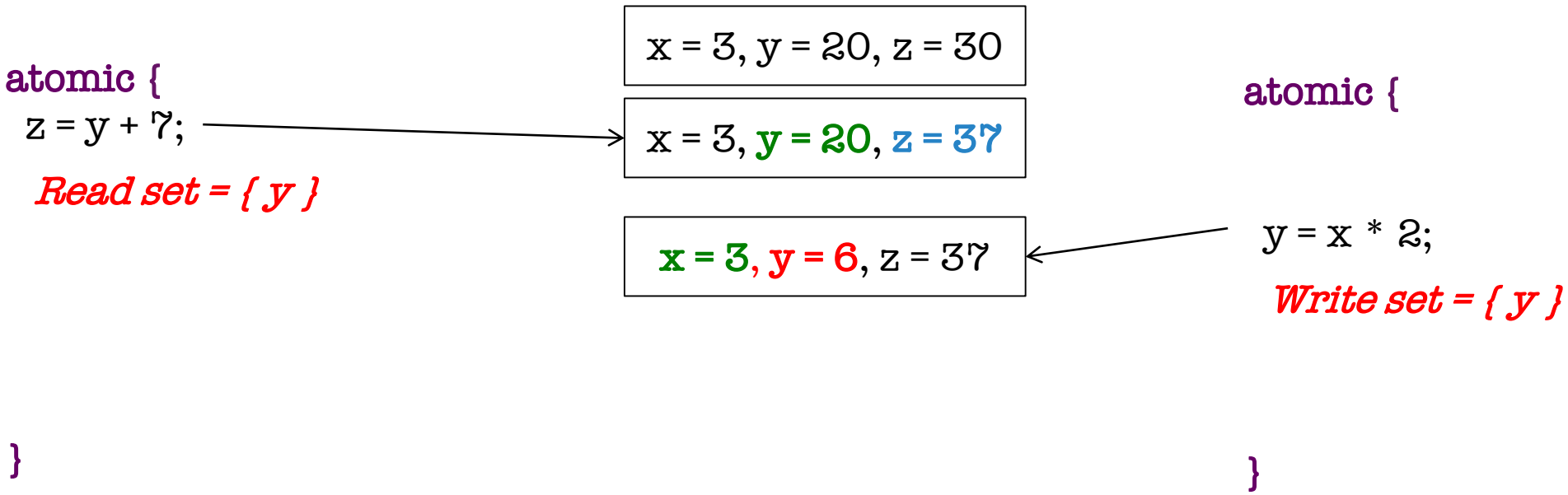
MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)



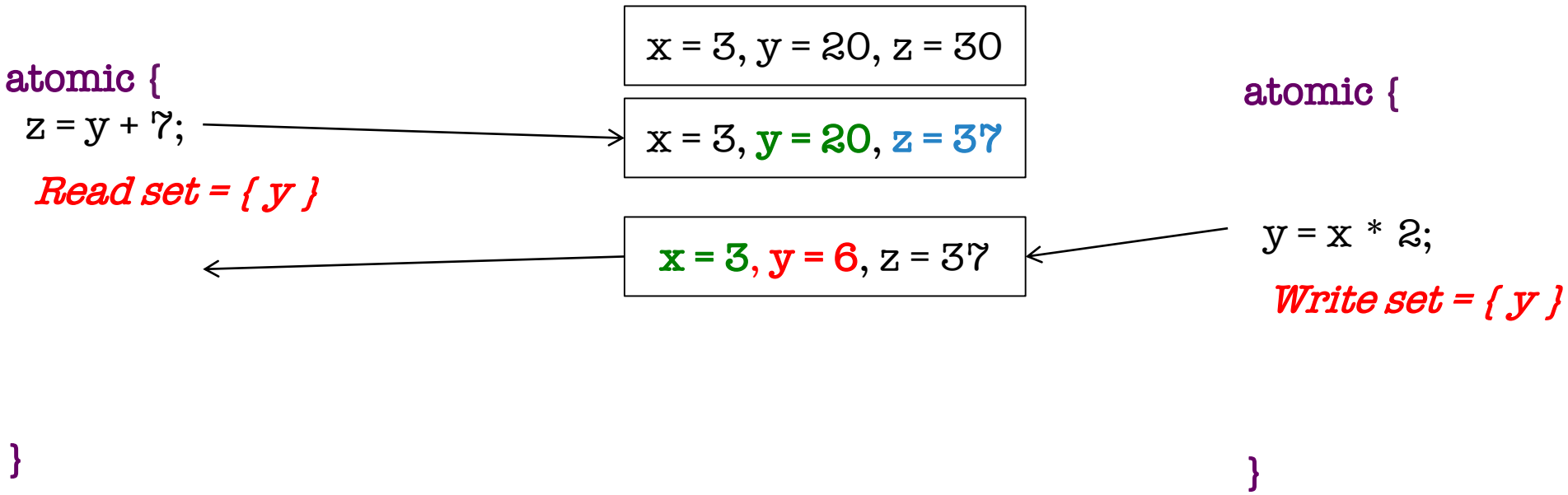
MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)



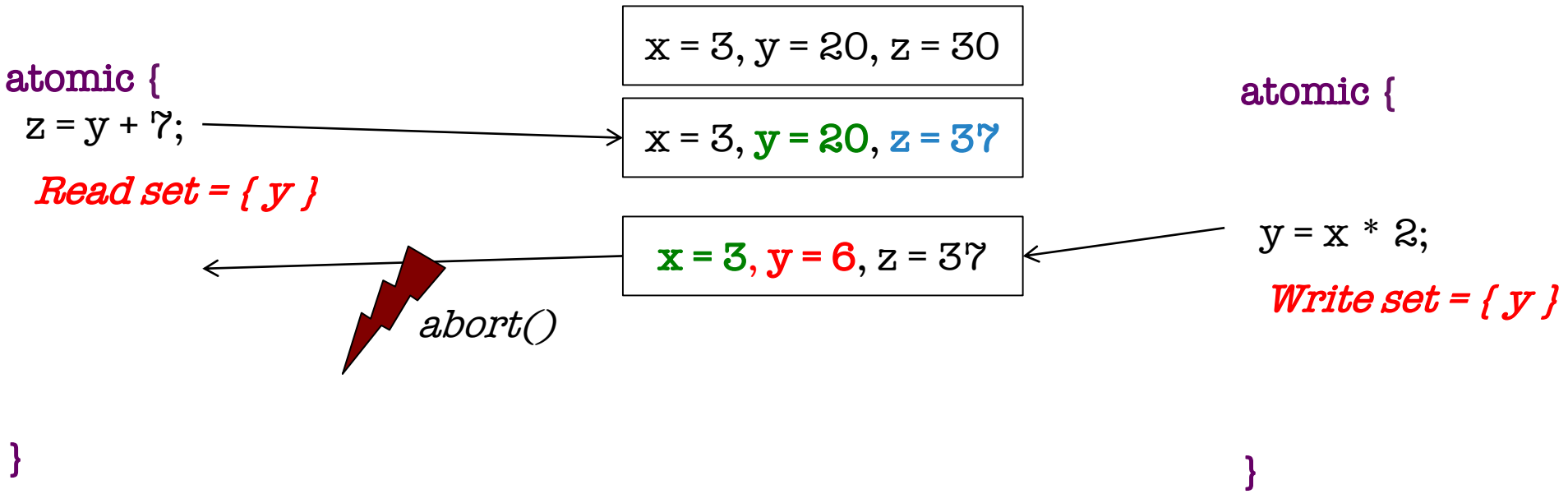
MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)



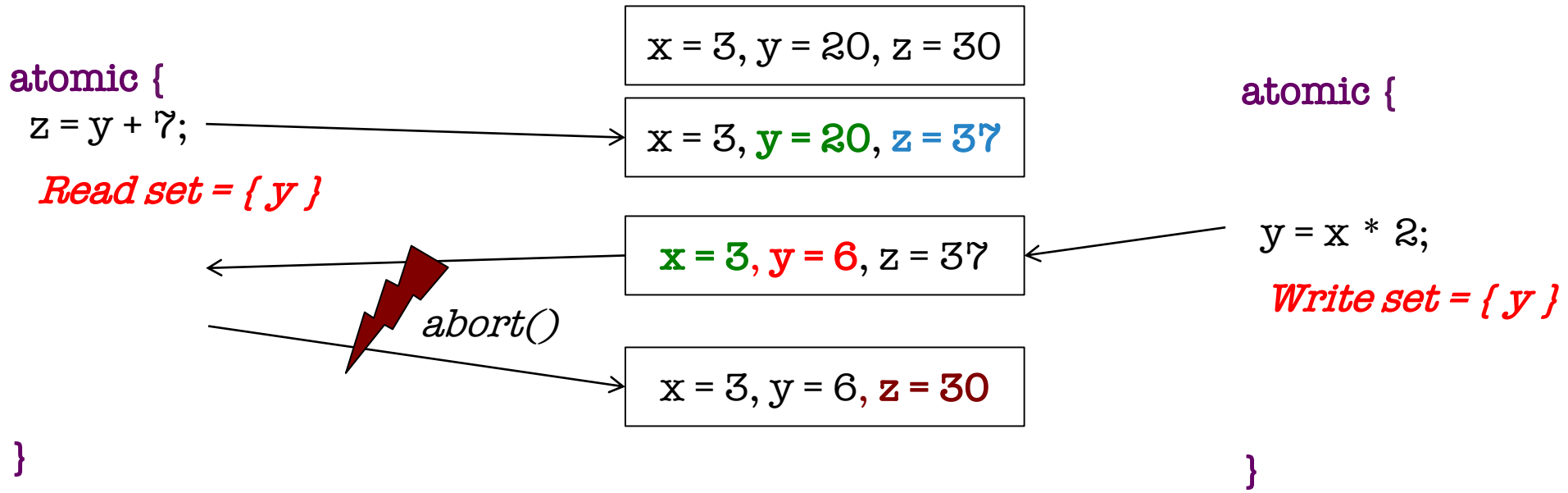
MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle immédiate (deferred update) :** aborts coûteux (on passe par un log…)



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle matérielle (HTM) :**
  - Utilise le cache du processeur pour construire une mémoire transactionnelle retardée
  - À la fin d'une transaction, soit lignes de cache propagées, soit invalidées si conflit
  - Au niveau du protocole de cohérence de cache
  - **Avantage :** très rapide, **inconvénient :** taille limitée
    - Au cache L1 sous Haswell par exemple, donc pas plus de 32Ko, limité pour e.g. une transaction dans une BDD

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle matérielle (HTM) :**
  - Utilise le cache du processeur pour construire une mémoire transactionnelle retardée
  - À la fin d'une transaction, soit lignes de cache propagées, soit invalidées si conflit
  - Au niveau du protocole de cohérence de cache
  - **Avantage :** très rapide, **inconvénient :** taille limitée
    - Au cache L1 sous Haswell par exemple, donc pas plus de 32Ko, limité pour e.g. une transaction dans une BDD
  - Géré dans les CPUs Intel dernière génération (Haswell) ! Mais buggy dans certains (désactivé…)

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle matérielle (HTM) :**
  - Utilise le cache du processeur pour construire une mémoire transactionnelle retardée
  - À la fin d'une transaction, soit lignes de cache propagées, soit invalidées si conflit
  - Au niveau du protocole de cohérence de cache
  - **Avantage :** très rapide, **inconvénient :** taille limitée
    - Au cache L1 sous Haswell par exemple, donc pas plus de 32Ko, limité pour e.g. une transaction dans une BDD
  - Géré dans les CPUs Intel dernière génération (Haswell) ! Mais buggy dans certains (désactivé…)
- **Mémoire transactionnelle logicielle (STM) :**
  - Le compilateur modifie les accès pour ajouter des appels aux fonctions de la STM
  - **Avantage :** taille de transactions quelconques, **inconvénient :** lent…

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Mémoire transactionnelle matérielle (HTM) :**
  - Utilise le cache du processeur pour construire une mémoire transactionnelle retardée
  - À la fin d'une transaction, soit lignes de cache propagées, soit invalidées si conflit
  - Au niveau du protocole de cohérence de cache
  - **Avantage :** très rapide, **inconvénient :** taille limitée
    - Au cache L1 sous Haswell par exemple, donc pas plus de 32Ko, limité pour e.g. une transaction dans une BDD
  - Géré dans les CPUs Intel dernière génération (Haswell) ! Mais buggy dans certains (désactivé…)
- **Mémoire transactionnelle logicielle (STM) :**
  - Le compilateur modifie les accès pour ajouter des appels aux fonctions de la STM
  - **Avantage :** taille de transactions quelconques, **inconvénient :** lent…
- **Mémoire transactionnelle hybride (HyTM) :**
  - Matérielle tant que le cache suffit, passe en logiciel sinon !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- Que se passe-t-il en cas d'I/O ?

```
atomic {
  if (x > 42)
    launchMissile();
}
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- Que se passe-t-il en cas d'I/O ?

```
atomic {
  if (x > 42)
    launchMissile();
}
```

- Les entrées/sorties se prêtent très mal aux transactions…

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- Que se passe-t-il en cas d'I/O ?

```
atomic {
 if (x > 42)
  launchMissile();
}
```

- Les entrées/sorties se prêtent très mal aux transactions…
- Cas spécifiques :
  - **Ecriture :** difficile, voire impossible à mettre dans une transaction
  - **Lecture :** on peut généralement tricher (en jouant avec des tampons par exemple…)

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

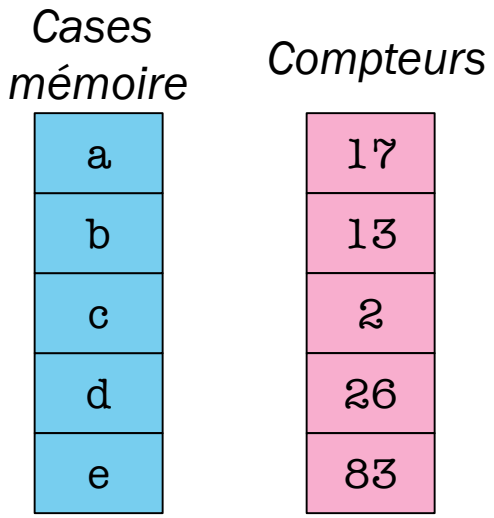
- Que se passe-t-il en cas d'I/O ?

```
atomic {
 if (x > 42)
  launchMissile();
}
```

- Les entrées/sorties se prêtent très mal aux transactions…
- Cas spécifiques :
  - **Ecriture :** difficile, voire impossible à mettre dans une transaction
  - **Lecture :** on peut généralement tricher (en jouant avec des tampons par exemple…)
- Des techniques existent, mais coûteuses et complexifient beaucoup les algos.

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

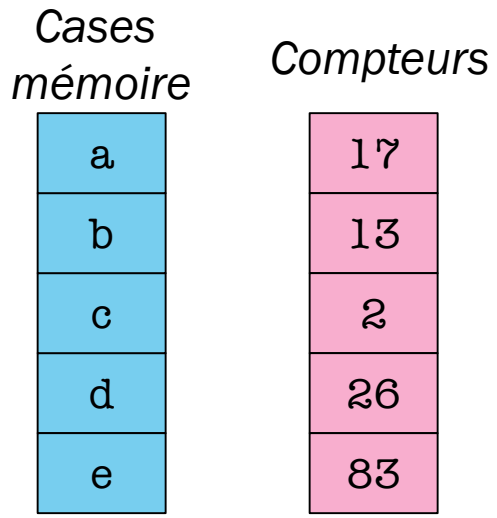
- **Exemple d'implémentation :** une STM retardée avec verrou au commit



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Exemple d'implémentation :** une STM retardée avec verrou au commit

- **Principe :**
  - Chaque case mémoire possède un compteur
  - **Lecture :** mémorise le compteur
  - **Ecriture :** mémorise la case (nouvelle valeur) et le compteur
  - **Lors d'une mise à jour :**
    - Vérifie que les compteurs n'ont pas changé
    - Incrémente les compteurs des cases écrites



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Fonctionnement :** un petit exemple

Mémoire globale

Cases	Valeur	Compteurs
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Copie locale

a
b
c
d
e

```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```


Université Nice Sophia Antipolis

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Fonctionnement :** un petit exemple

Mémoire globale

Cases	Valeur	Compteurs
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

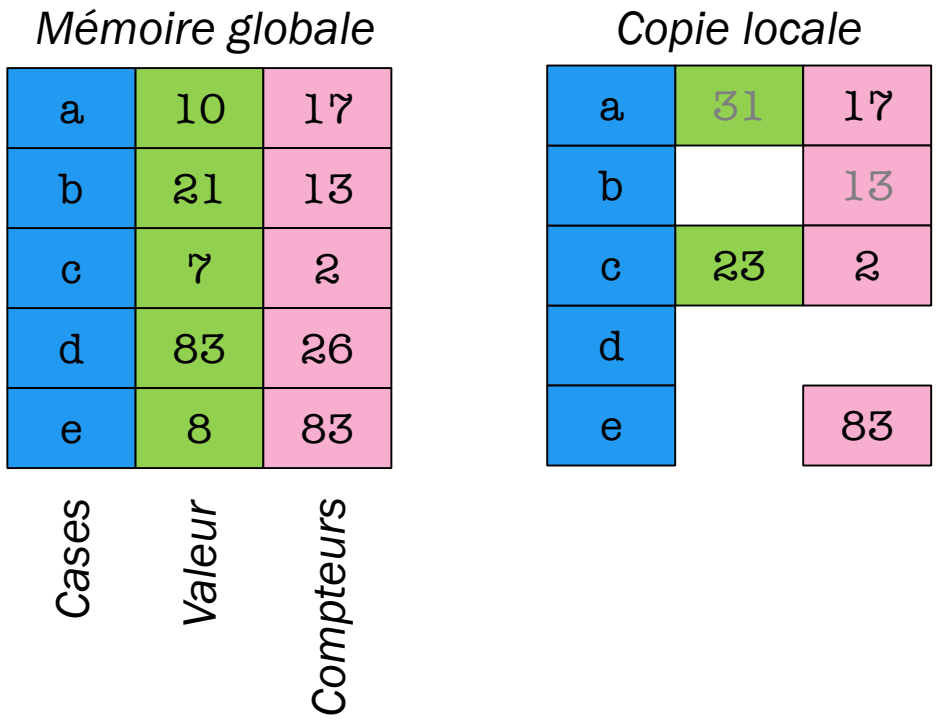
Copie locale

a	31	17
b		13
c		
d		
e		

```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

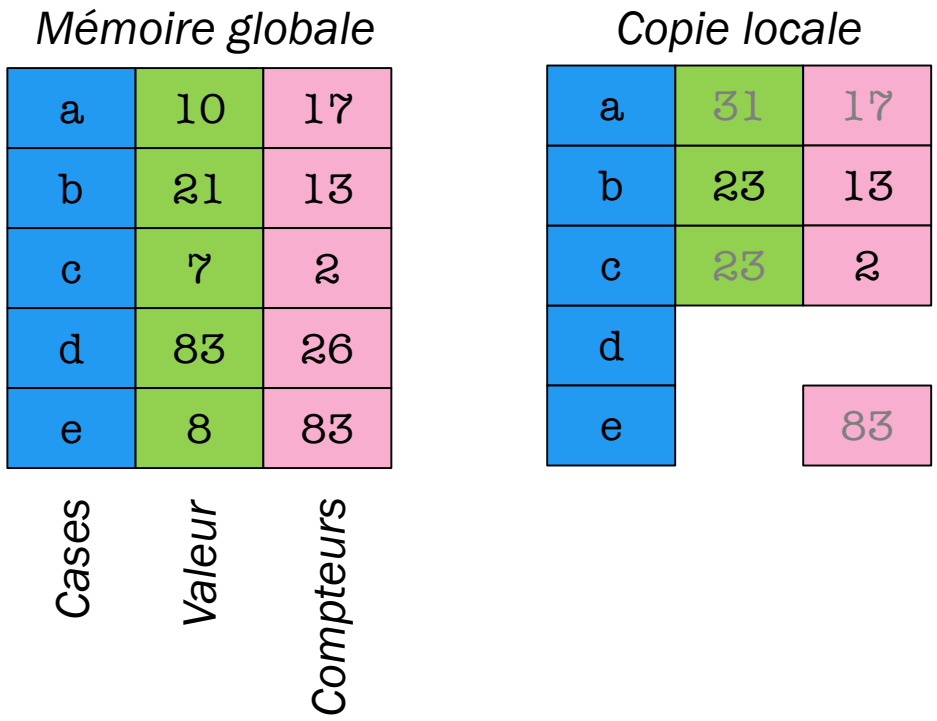
- **Fonctionnement :** un petit exemple




```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Fonctionnement :** un petit exemple




```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Fonctionnement :** un petit exemple

Mémoire globale

Cases	Valeur	Compteurs
a	10	17
b	21	13
c	7	2
d	83	26
e	8	83

Copie locale

a	31	17
b	23	13
c	23	2
d		
e		83

```
a. atomic {
b.   a = a + b;
c.   c = a - e;
d.   b = c;
e. }
```

Mémoire global au commit

a	31	18
b	23	14
c	23	3
d	83	26
e	8	83

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

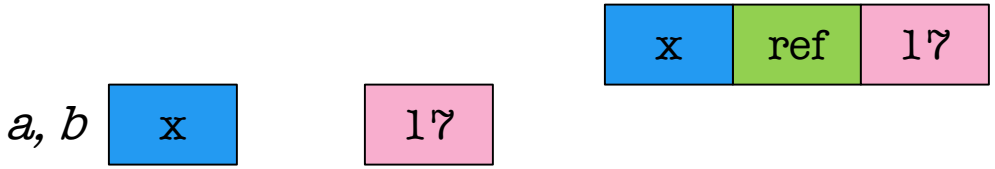
```
e. atomic {
f.   x = null;
g. }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a. atomic {
b.  if(x != null)
c.     x.f();
d.  }
```

```
e. atomic {
f.    x = null;
g.  }
```

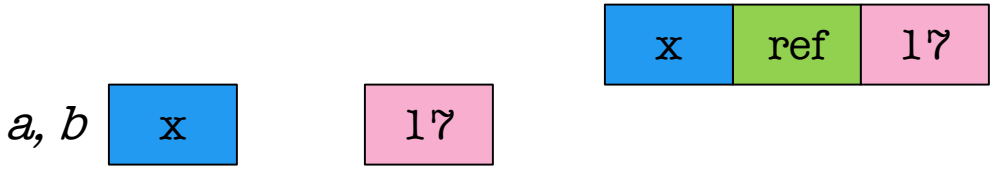


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a. atomic {
b.  if(x != null)
c.     x.f();
d.  }
```

```
e. atomic {
f.    x = null;
g.  }
```

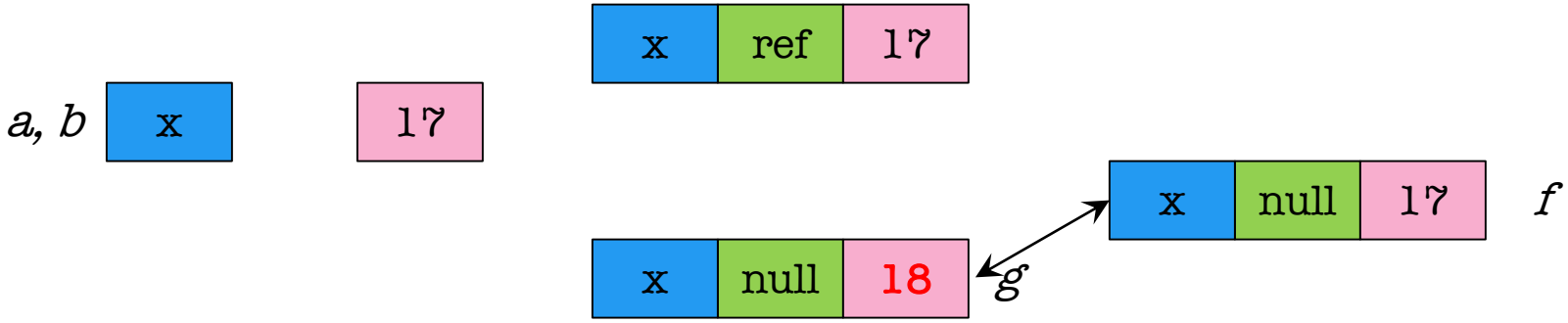


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a.  atomic {
b.   if(x != null)
c.      x.f();
d.  }
```

```
e.  atomic {
f.     x = null;
g.  }
```

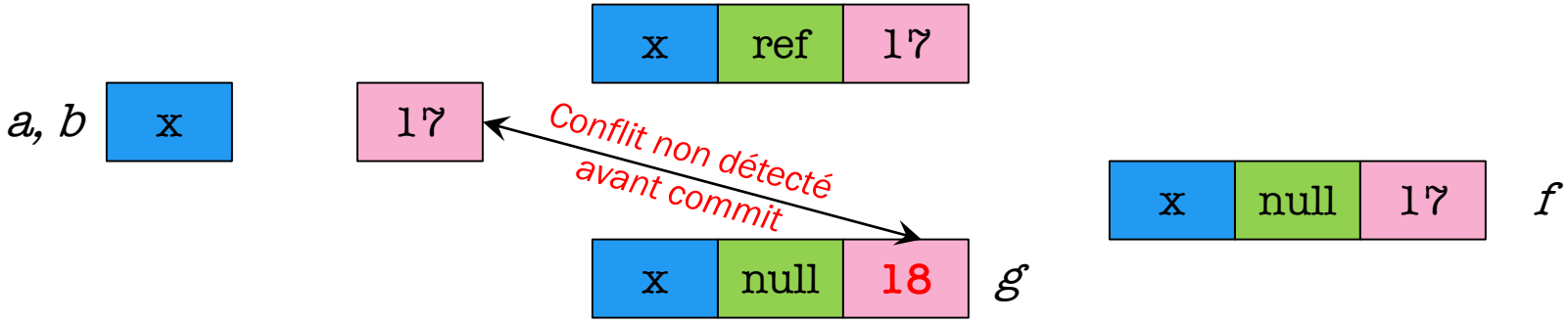


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a. atomic {
b.  if(x != null)
c.     x.f();
d.  }
```

```
e. atomic {
f.    x = null;
g.  }
```

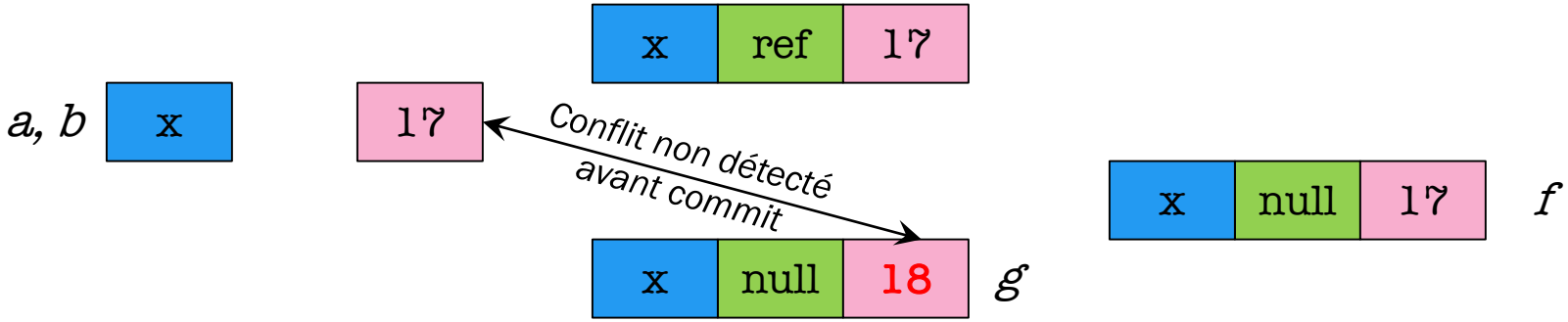


MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

```
e. atomic {
f.   x = null;
g. }
```



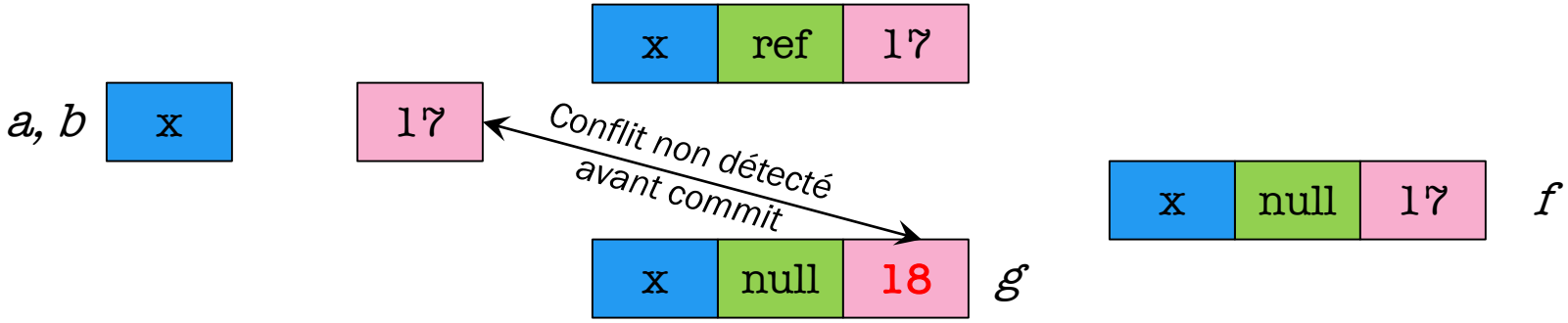
c: NullPointerException

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Problème classique :** les transactions zombies

```
a. atomic {
b.   if(x != null)
c.     x.f();
d. }
```

```
e. atomic {
f.   x = null;
g. }
```



c: NullPointerException

- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

Invariant de transaction : x != y

(Sinon division par zéro)

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

x	4	17
y	5	83

- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

Invariant de transaction : x != y

(Sinon division par zéro)

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

x	4	17
y	5	83

a,b : t1 = 4

- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

Invariant de transaction : `x != y`

(Sinon division par zéro)

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

Initialement : x = 4, y = 5

x	4	17
y	5	83

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

a,b : t1 = 4

x

17

x	217	17
y	4	83

f, g, h

- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

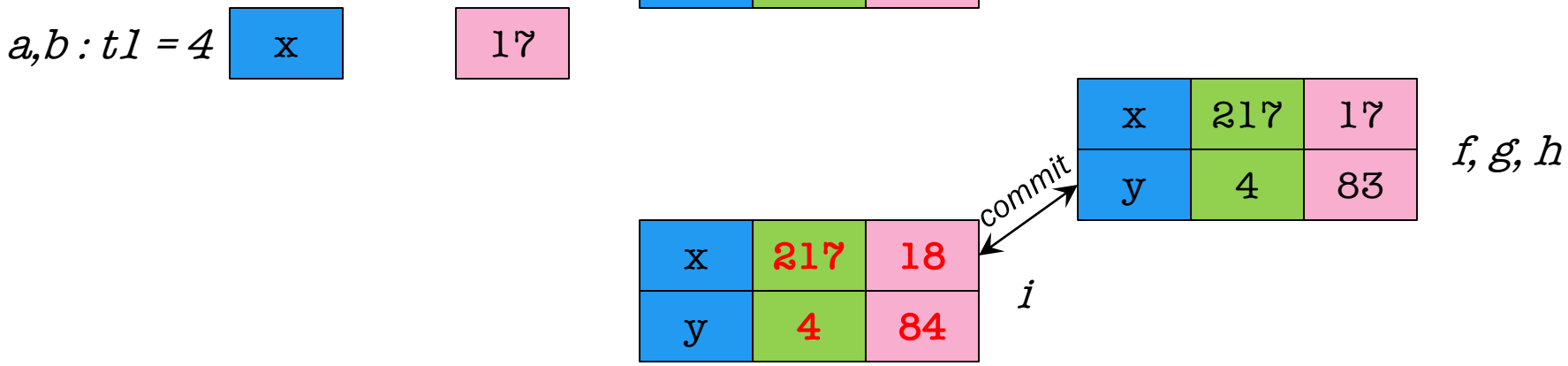
Invariant de transaction : x != y

(Sinon division par zéro)

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```



- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

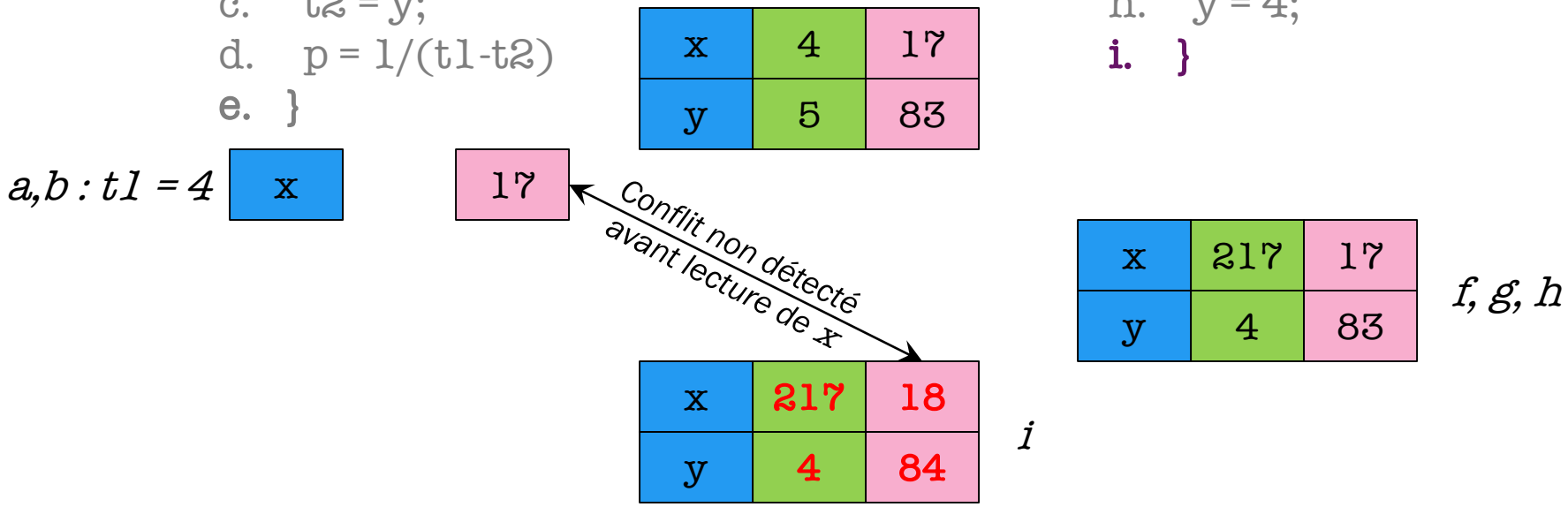
Invariant de transaction : x != y

(Sinon division par zéro)

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```



- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

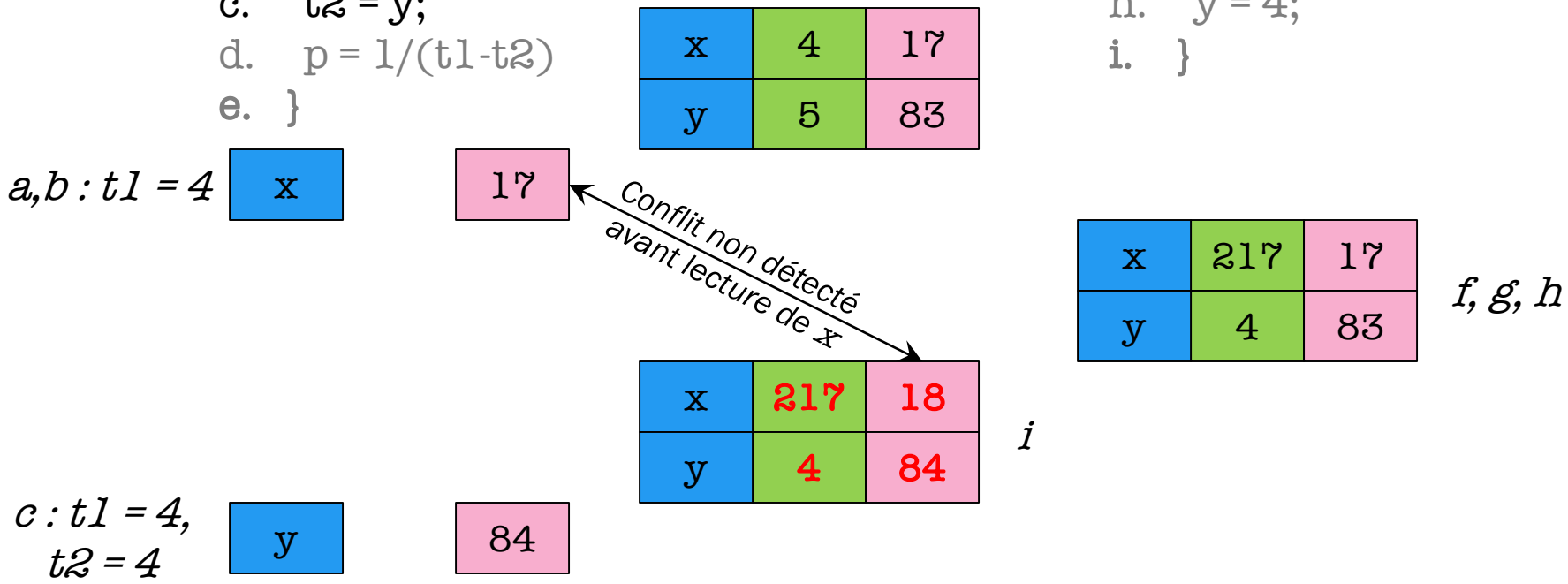
Invariant de transaction : x != y

(Sinon division par zéro)

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```



- **Solution intuitive :** vérifier le compteur d'une variable à chacune de ses lectures…
  - Ne suffit pas !

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

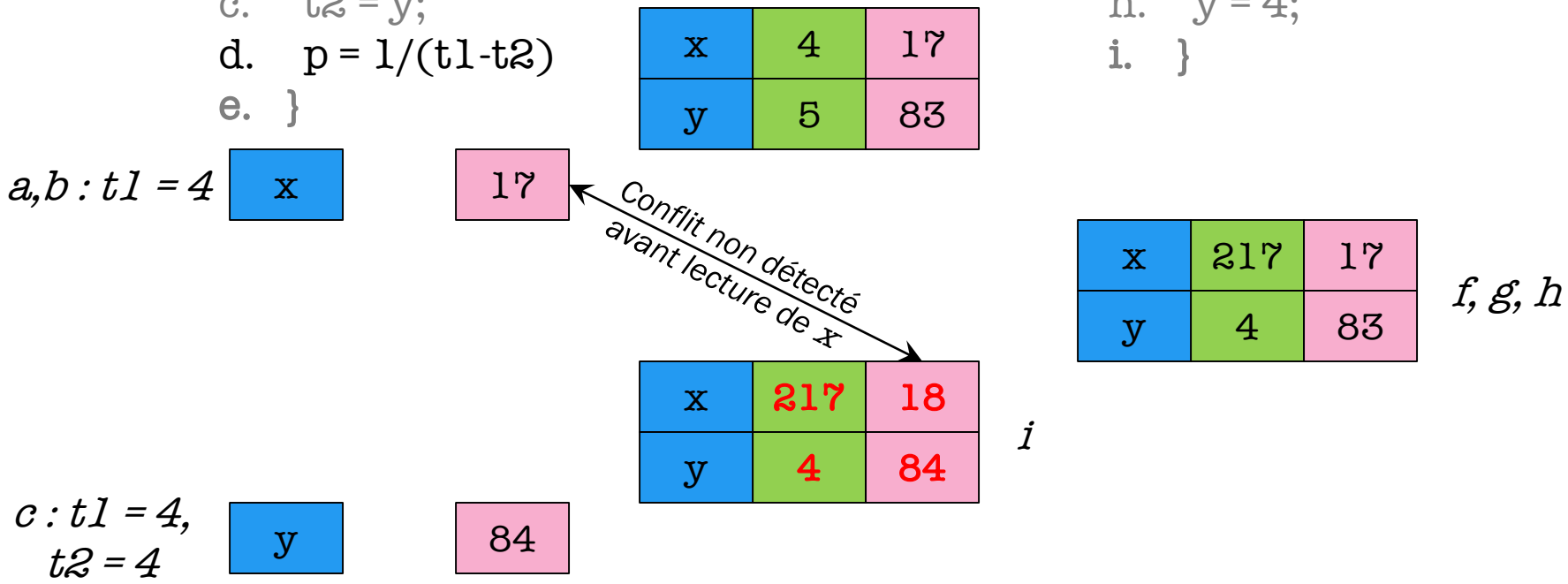
Invariant de transaction : x != y

(Sinon division par zéro)

Initialement : x = 4, y = 5

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```



Problème : *pas de lecture de x avant le commit, donc on ne regarde pas son compteur*

d ⇒ division par zéro!

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- Vérifier le compteur à chaque lecture pas suffisant
  - On ne peut pas non plus vérifier tous les compteurs du read set à chaque lecture, trop lent…

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Vérifier le compteur à chaque lecture pas suffisant**
  - On ne peut pas non plus vérifier tous les compteurs du read set à chaque lecture, trop lent…
- **Solution possible :** horloge globale pour la mémoire + horloge locale pour chaque transaction.

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Vérifier le compteur à chaque lecture pas suffisant**
  - On ne peut pas non plus vérifier tous les compteurs du read set à chaque lecture, trop lent…
- **Solution possible :** horloge globale pour la mémoire + horloge locale pour chaque transaction.
- **Début transaction :** copie horloge globale dans horloge locale

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Vérifier le compteur à chaque lecture pas suffisant**
  - On ne peut pas non plus vérifier tous les compteurs du read set à chaque lecture, trop lent…
- **Solution possible :** horloge globale pour la mémoire + horloge locale pour chaque transaction.
- **Début transaction :** copie horloge globale dans horloge locale
- **À chaque lecture :** vérifie que compteur de la variable < horloge locale
  - Prouve que toutes valeurs lues dans un état valide avant le début de la transaction
    - Pas de mélange avec nos modifs temporaires
  - On ne stocke pas le compteur

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Vérifier le compteur à chaque lecture pas suffisant**
  - On ne peut pas non plus vérifier tous les compteurs du read set à chaque lecture, trop lent…
- **Solution possible :** horloge globale pour la mémoire + horloge locale pour chaque transaction.
- **Début transaction :** copie horloge globale dans horloge locale
- **À chaque lecture :** vérifie que compteur de la variable < horloge locale
  - Prouve que toutes valeurs lues dans un état valide avant le début de la transaction
    - Pas de mélange avec nos modifs temporaires
  - On ne stocke pas le compteur
- **À chaque écriture :** pareil qu'avant, sauf qu'on ne stocke pas le compteur

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

- **Vérifier le compteur à chaque lecture pas suffisant**
  - On ne peut pas non plus vérifier tous les compteurs du read set à chaque lecture, trop lent…
- **Solution possible :** horloge globale pour la mémoire + horloge locale pour chaque transaction.
- **Début transaction :** copie horloge globale dans horloge locale
- **À chaque lecture :** vérifie que compteur de la variable < horloge locale
  - Prouve que toutes valeurs lues dans un état valide avant le début de la transaction
    - Pas de mélange avec nos modifs temporaires
  - On ne stocke pas le compteur
- **À chaque écriture :** pareil qu'avant, sauf qu'on ne stocke pas le compteur
- **Commit :**
  - S'il existe un compteur du RS/WS > horloge locale, annule transaction (conflit)
  - Pour toute variable écrite, met son compteur à horloge globale
  - Incrémente horloge globale

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

```
a.  atomic {
b.    t1 = x;
c.    t2 = y;
d.    p = 1/(t1-t2)
e.  }
```

90

```
f.  atomic {
g.    x = 217;
h.    y = 4;
i.  }
```

f 90

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

90

Autres transactions en //

100

x	4	17
y	5	83

```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

f 90

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

a 100

90

Autres transactions en //

100

x	4	17
y	5	83

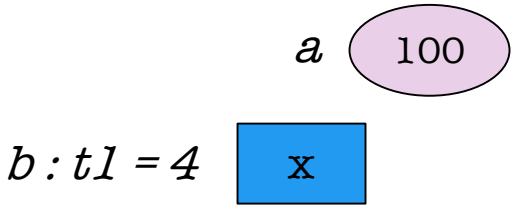
```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

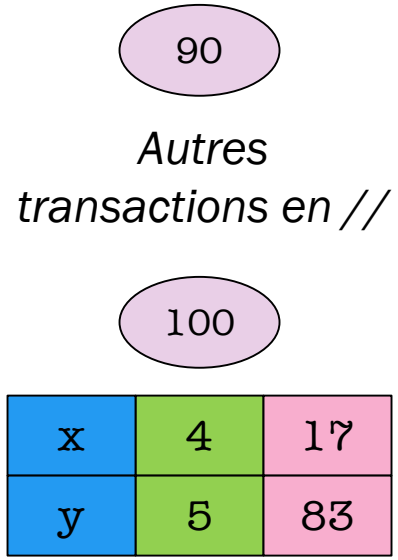
f 90

Université Nice Sophia Antipolis

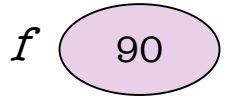
MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```






```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION

```
a. atomic {
b.   t1 = x;
c.   t2 = y;
d.   p = 1/(t1-t2)
e. }
```

90

Autres transactions en //

100

x	4	17
y	5	83

a 100

b : t1 = 4 x

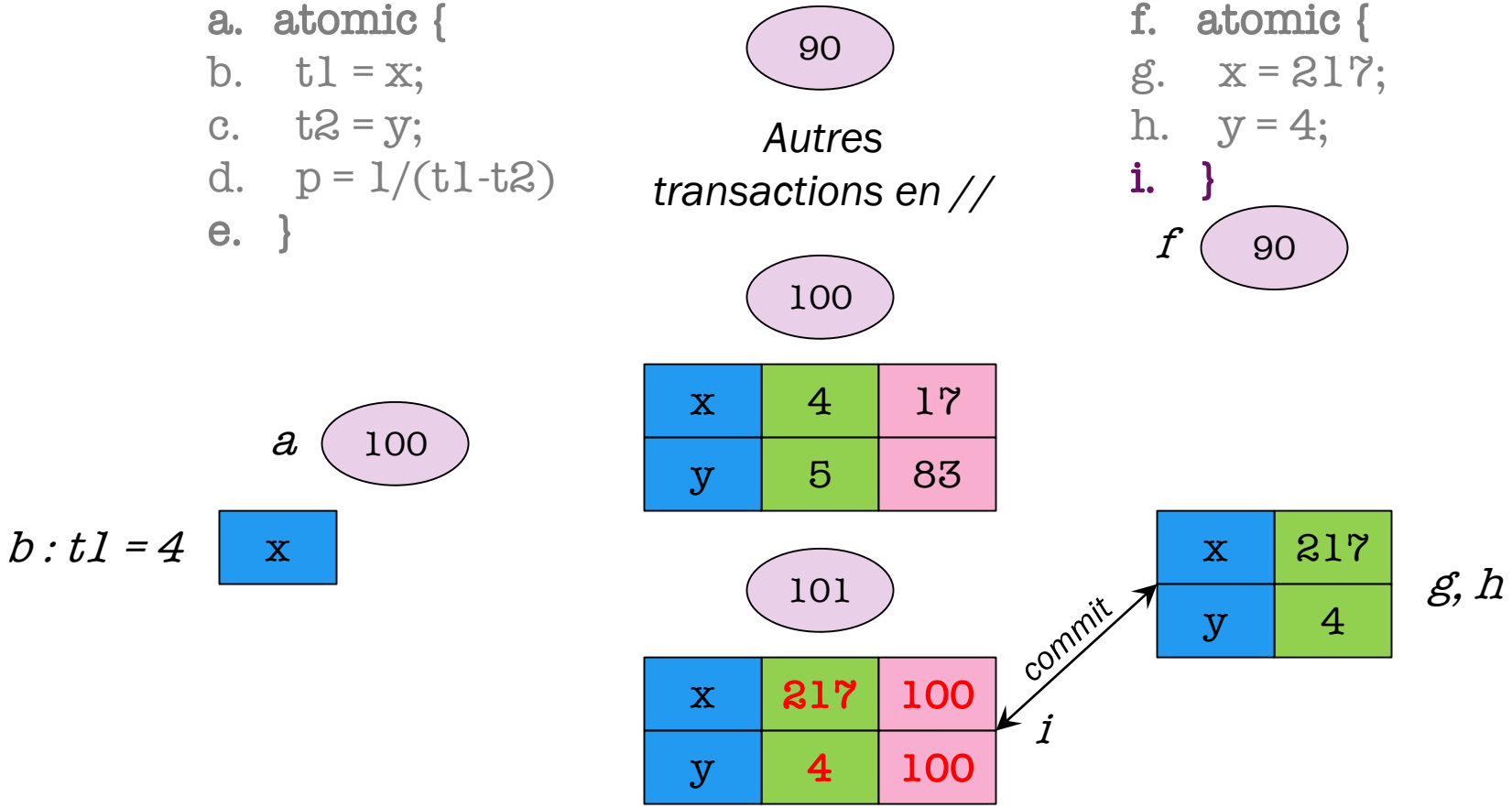
```
f. atomic {
g.   x = 217;
h.   y = 4;
i. }
```

f 90

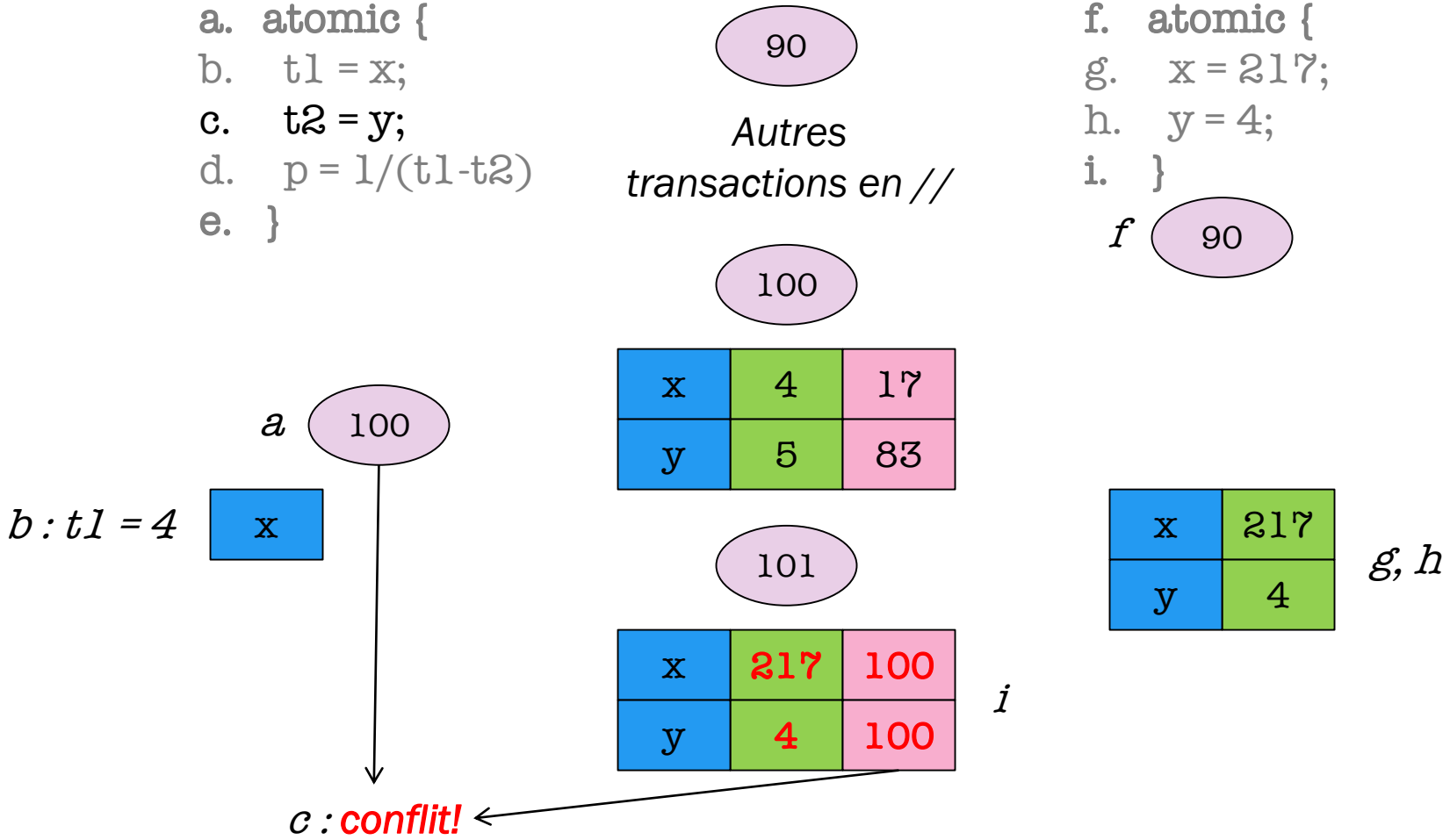
x	217
y	4

g, h

MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION



MÉMOIRES TRANSACTIONNELLES : IMPLÉMENTATION



MÉMOIRES TRANSACTIONNELLES : CONCLUSION

- **Implémentation : pas si simple…**
  - On a vu un exemple d'un algorithme naïf, nombreuses implémentations possibles

MÉMOIRES TRANSACTIONNELLES : CONCLUSION

- **Implémentation : pas si simple…**
  - On a vu un exemple d'un algorithme naïf, nombreuses implémentations possibles

- **Simplification de la programmation concurrente**
  - Concept simple
  - Plus de problème de deadlock, de famine
  - Transactions composables

MÉMOIRES TRANSACTIONNELLES : CONCLUSION

- **Implémentation : pas si simple…**
  - On a vu un exemple d'un algorithme naïf, nombreuses implémentations possibles

- **Simplification de la programmation concurrente**
  - Concept simple
  - Plus de problème de deadlock, de famine
  - Transactions composables

- **Et niveau performances ?**
  - STM lentes.
  - HTM peuvent être plus rapides que verrous…

MÉMOIRES TRANSACTIONNELLES : CONCLUSION

- **Implémentation : pas si simple…**
  - On a vu un exemple d'un algorithme naïf, nombreuses implémentations possibles

- **Simplification de la programmation concurrente**
  - Concept simple
  - Plus de problème de deadlock, de famine
  - Transactions composables

- **Et niveau performances ?**
  - STM lentes.
  - HTM peuvent être plus rapides que verrous…
  - Vient d'apparaître dans les CPUs Intel Haswell. Le futur ?