Systèmes concurrents

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

16 septembre 2020

Neuvième partie

Synchronisation non bloquante

Plan

Limitation des verrous

Limites des verrous (et plus généralement de la synchronisation par blocage/attente) :

- Interblocage : ensemble de processus se bloquant mutuellement
- Inversion de priorité : un processus de faible priorité bloque un processus plus prioritaire
- Convoi : une ensemble d'actions avance à la vitesse de la plus lente
- Interruption : quelles actions dans un gestionnaire de signal ?
- Arrêt involontaire d'un processus
- Tuer un processus ?
- Granularité des verrous $\rightarrow$ performance

Objectifs de la synchronisation non bloquante ♪♪♪

Problème

Garantir la cohérence d'accès à un objet partagé sans blocage

- Résistance à l'arrêt (crash) d'une activité : une activité donnée n'est jamais empêchée de progresser, quel que soit le comportement des autres activités
- Vitesse de progression indépendante de celle des autres activités
- Passage à l'échelle
- Surcoût négligeable de synchronisation en absence de conflit (notion de *fast path*)
- Compatible avec la programmation événementielle (un gestionnaire d'interruption ne doit pas être bloqué par la synchronisation)

Synchronisation non bloquante

Non-blocking synchronization

Obstruction-free Si à tout point, une activité en isolation parvient à terminer en temps fini (en un nombre fini de pas).

Lock-free Synchronisation et protection garantissant la *progression du système* même si une activité s'arrête arbitrairement. Peut utiliser de l'attente active mais (par exemple) pas de verrous.
Absence d'interblocage et d'inversion de priorité mais risque de famine individuelle (vivacité faible).

Wait-free Une sous-classe de lock-free où *toute activité* est certaine de compléter son action en temps fini, indépendamment du comportement des autres activités (arrêtées ou agressivement interférantes).
Absence de famine individuelle (vivacité forte).

Mécanismes matériels

♪♪♪

Mécanismes matériels utilisés

- Registres : protocoles permettant d'abstraire la gestion de la concurrence d'accès à la mémoire partagée (caches. . .).
  - registres sûrs : toute lecture fournit une valeur écrite ou en cours d'écriture
  - registres réguliers : toute lecture fournit la dernière valeur écrite ou une valeur en cours d'écriture
  - registres atomiques : toute lecture fournit la dernière valeur écrite
- Instructions processeur atomiques combinant lecture(s) et écriture(s) (exemple : test-and-set)

Principes généraux

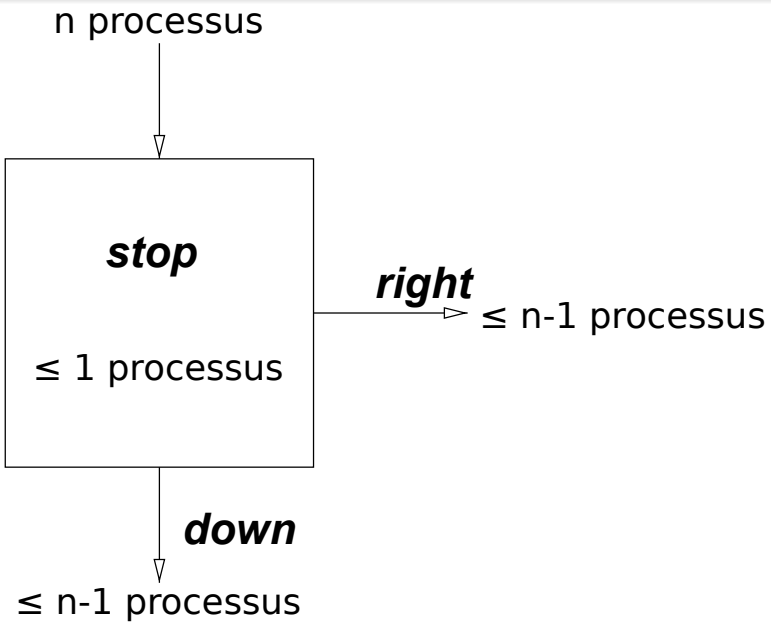
Principes

- Chaque activité travaille à partir d'une copie locale de l'objet partagé
- Un conflit est détecté lorsque la copie diffère de l'original
- Boucle active en cas de conflit d'accès non résolu
  $\rightarrow$ limiter le plus possible la zone de conflit
- Entraide : si un conflit est détecté, une activité peut exécuter des opérations pour le compte d'une autre activité (p.e. finir la mise à jour de l'objet partagé)

Plan

Splitter

Moir, Anderson 1995



- x (indéterminé) activités appellent concurremment (ou pas) le splitter
- au plus une activité termine avec *stop*
- si $x = 1$, l'activité termine avec *stop*
- au plus $(x - 1)$ activités terminent avec *right*
- au plus $(x - 1)$ activités terminent avec *down*

Splitter

Registres

- Lectures et écritures atomiques
- Pas d'interférence due aux caches en multiprocesseur

Implantation non bloquante

Deux registres partagés : X (init $\forall$) et Y (init faux)
Chaque activité a un identifiant unique id_i et un résultat dir_i.

```
function direction (id_i)
    X := id_i;
    if Y then dir_i := right;
    else Y := true;
        if (X = id_i) then dir_i := stop;
        else dir_i := down;
        end if
    end if
    return dir_i;
```

Schéma de preuve

Validité les seules valeurs retournées sont right, stop et down.

Vivacité ni boucle ni blocage

stop si $x = 1$ évident (une seule activité exécute *direction()*)

au plus $x - 1$ right les activités obtenant right trouvent Y, qui a nécessairement été positionné par une activité obtenant down ou stop

au plus $x - 1$ down soit p_i la dernière activité ayant écrit X. Si p_i trouve Y, elle obtiendra right. Sinon son test $X = id_i$ lui fera obtenir stop.

au plus 1 stop soit p_i la *première* activité trouvant $X = id_i$. Alors aucune activité n'a modifié X depuis que p_i l'a fait. Donc toutes les activités suivantes trouveront Y et obtiendront right (car p_i a positionné Y), et les activités en cours qui n'ont pas trouvé Y ont vu leur écriture de X écrasée par p_i (puisqu'elle n'a pas changé jusqu'au test par p_i). Elles ne pourront donc trouver X égal à leur identifiant et obtiendront donc down.

Renommage

♪♪♪

- Soit n activités d'identité $id_1, \ldots, id_n \in [0..N]$ où $N \gg n$
- On souhaite renommer les activités pour qu'elles aient une identité prise dans [0..M] où $M \ll N$
- Deux activités ne doivent pas avoir la même identité

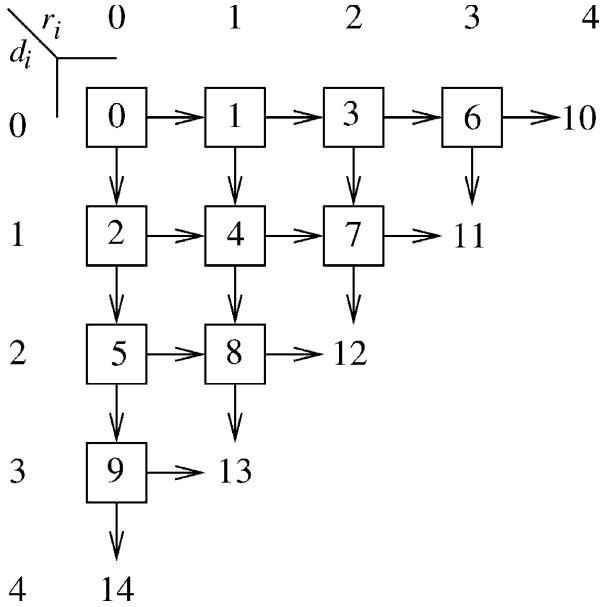
Solution à base de verrous

- Distributeur de numéro accédé en exclusion mutuelle
- $M = n$
- Complexité temporelle : O(1) pour un numéro, O(n) pour tous
- Une activité lente ralentit les autres

Solution non bloquante

- Grille de splitters
- $M = \frac{n(n+1)}{2}$
- Complexité temporelle : O(n) pour un numéro, O(n) pour tous

Grille de splitters



Étiquettes uniques : un splitter renvoie stop à une activité au plus

Vivacité : traversée d'un nombre fini de splitters, chaque splitter est non bloquant

Toute activité obtient une étiquette :

- stop si $x = 1$,
- un splitter ne peut orienter toutes les activités dans la même direction,
- les bords de la grille sont à distance $n - 1$ de l'origine.

Renommage non bloquant

get_name(id_i)

$d_i \leftarrow 0$; $r_i \leftarrow 0$; $term_i \leftarrow false$;
while $(\neg term_i)$ **do**

$X[d_i, r_i] \leftarrow id_i$;
if $Y[d_i, r_i]$ **then** $r_i \leftarrow r_i + 1$; % *right*
else $Y[d_i, r_i] \leftarrow true$;
if $(X[d_i, r_i] = id_i)$ **then** $term_i \leftarrow true$; % *stop*
else $d_i \leftarrow d_i + 1$; % *down*
endif
endif

endwhile
return $\frac{1}{2}(r_i + d_i)(r_i + d_i + 1) + d_i$
% *le nom en position* d_i, r_i *de la grille*

Pile chaînée basique

♪♪♪

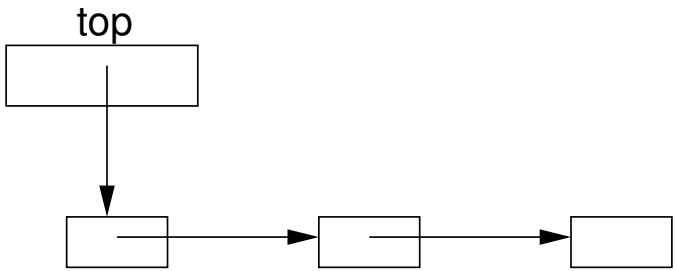
Objet avec opérations `push` et `pop`

```
class Stack<T> {
  class Node<T> { Node<T> next; T item; }

  Node<T> top;

  public void push(T item) {
    Node<T> newTop
            = new Node<>(item);
    Node<T> oldTop = top;
    newTop.next = oldTop;
    top = newTop;
  }
}
```

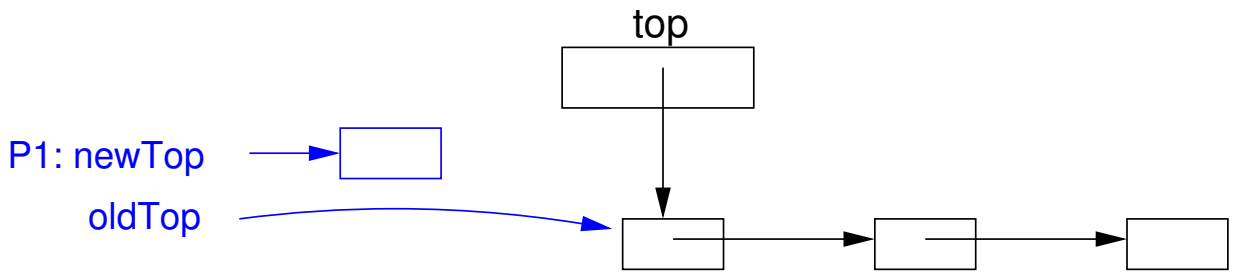
```
public T pop() {
  Node<T> oldTop = top;
  if (oldTop == null)
    return null;
  top = oldTop.next;
  return oldTop.item;
}
```

Non résistant à une utilisation concurrente par plusieurs activités

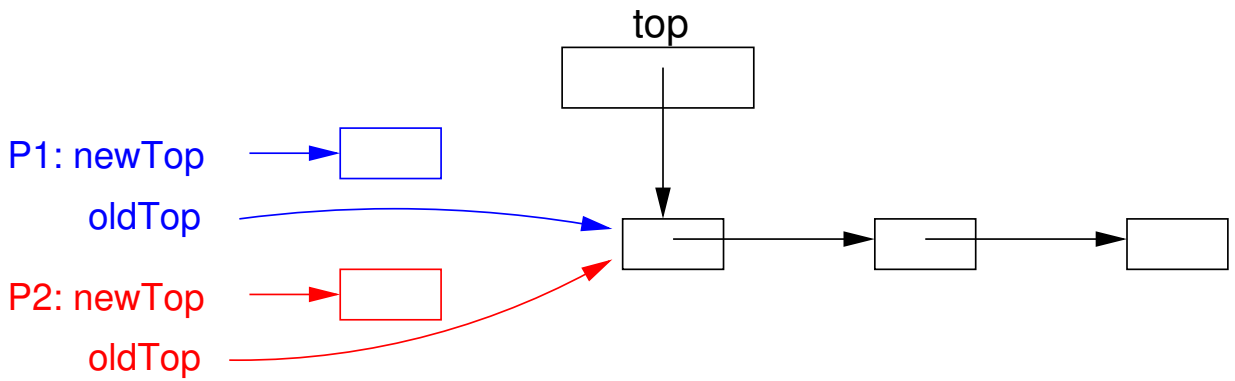
Pile chaînée basique : conflit push/push



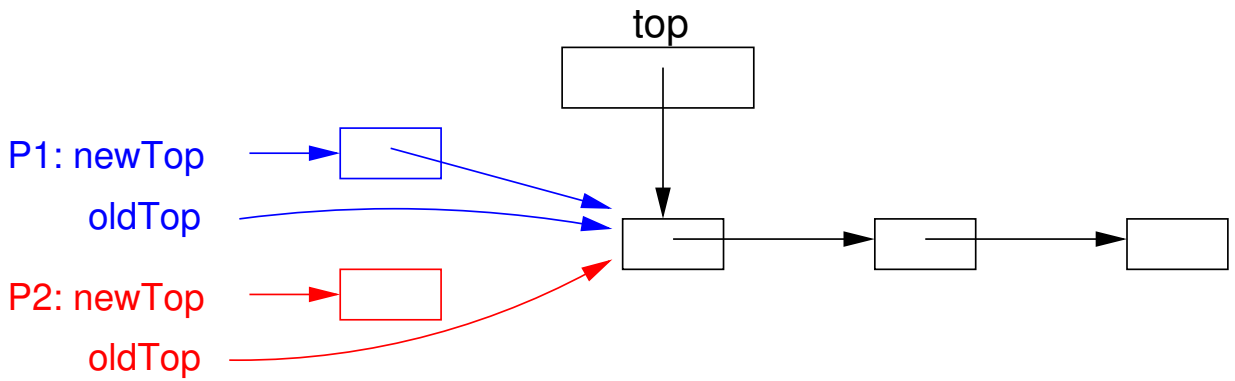
Pile chaînée basique : conflit push/push



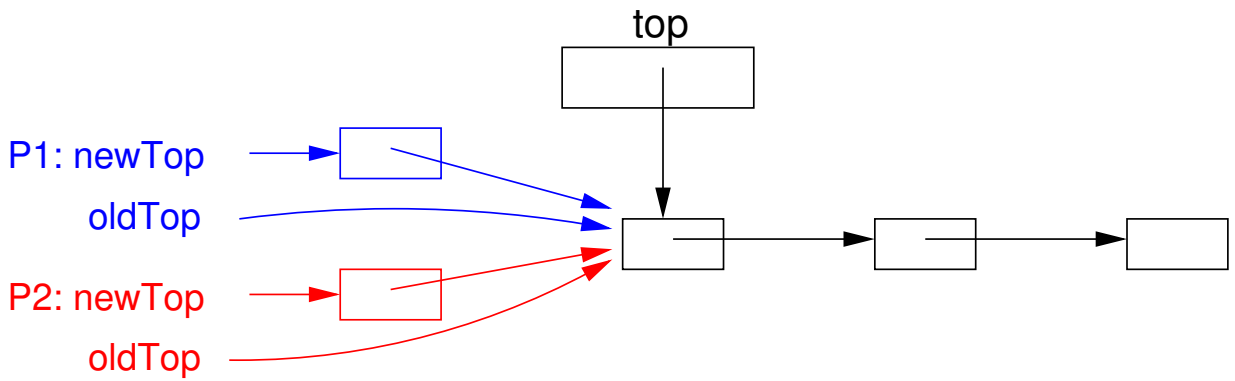
Pile chaînée basique : conflit push/push



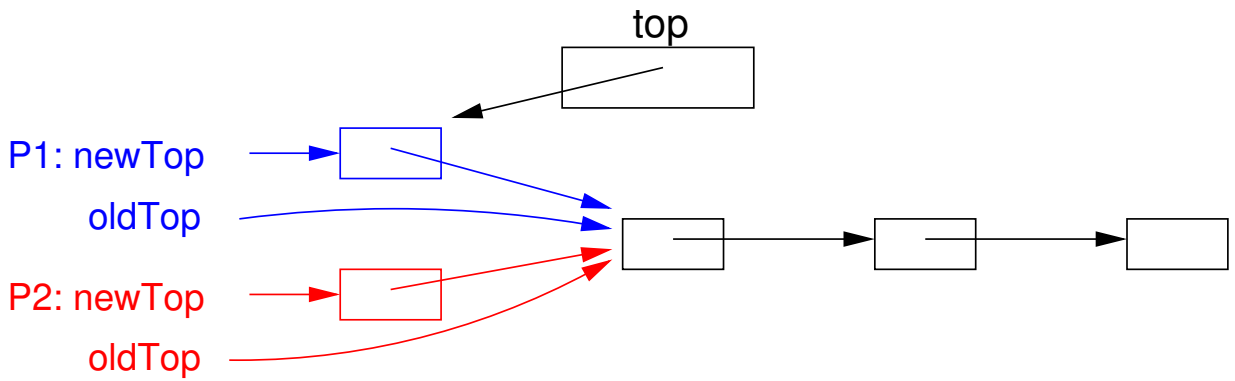
Pile chaînée basique : conflit push/push



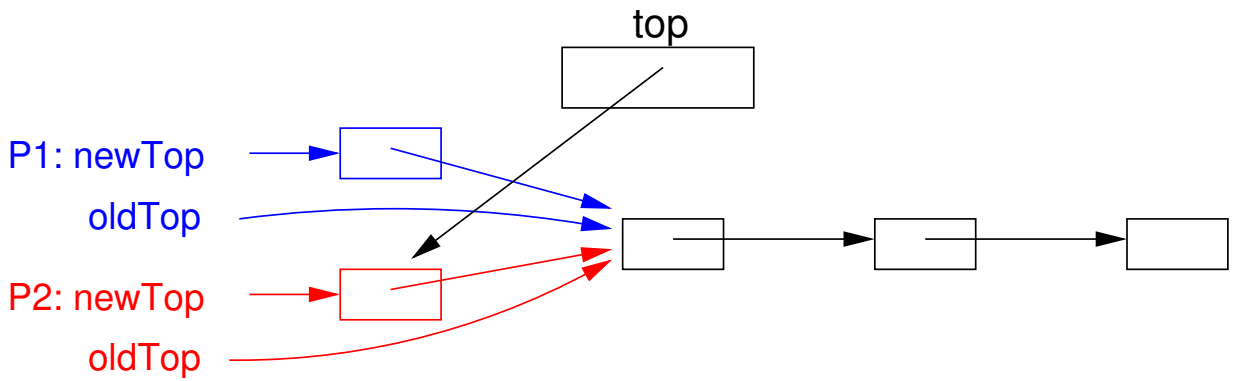
Pile chaînée basique : conflit push/push



Pile chaînée basique : conflit push/push



Pile chaînée basique : conflit push/push



Le problème est que pour P2, `top` a changé entre sa lecture et sa mise à jour.

De même conflit push/pop et pop/pop

Synchronisation classique

Conflit push/push, pop/pop, push/pop ⇒ exclusion mutuelle

```
public void push(T item) {
    verrou.lock();
    Node<T> newTop
            = new Node<>(item);
    Node<T> oldTop = top;
    newTop.next = oldTop;
    top = newTop;
    verrou.unlock();
}
```

```
public T pop() {
    verrou.lock();
    try {
        Node<T> oldTop = top;
        if (oldTop == null)
            return null;
        top = oldTop.next;
        return oldTop.item;
    } finally {
        verrou.unlock();
    }
}
```

- Bloquant définitivement si une activité s'arrête en plein milieu
- Toutes les activités sont ralenties par un unique lent

Pile chaînée non bloquante

Principe du `push`

1. Préparer une nouvelle cellule (valeur à empiler)
2. Chaîner cette cellule avec le sommet actuel
3. Si le sommet n'a pas changé, le mettre à jour avec la nouvelle cellule. *cette action doit être atomique !*
4. Sinon, recommencer à l'étape 2

Principe du `pop`

1. Récupérer la cellule au sommet
2. Récupérer la cellule suivante celle au sommet
3. Si le sommet n'a pas changé, le mettre à jour avec celle-ci. *cette action doit être atomique !*
4. Sinon, recommencer à l'étape 1
5. Retourner la valeur dans l'ancien sommet

Registres et Compare-and-set

`java.util.concurrent.atomic.AtomicReference`

- Lectures et écritures atomiques (registres atomiques), sans interférence due aux caches en multiprocesseur
- Une instruction atomique évoluée : `compareAndSet`

```
public class AtomicReference<V> { /* simplified */
  private volatile V value; /* la valeur contenue dans le registre */
  public V get() { return value; }
  public boolean compareAndSet(V expect, V update) {
      atomically {
          if (value == expect) { value = update; return true; }
          else { return false; }
      }
  }
}
```

Push/pop lock free

```
class Stack<T> {
    class Node<T> { Node<T> next; T item; }
    AtomicReference<Node<T>> top = new AtomicReference<>();

    public void push(T item) {
        Node<T> oldTop, newTop = new Node<>();
        newTop.item = item;
        do {
            oldTop = top.get();
            newTop.next = oldTop;
        } while (! top.compareAndSet(oldTop, newTop));
    }

    public T pop() {
        Node<T> oldTop, newTop;
        do {
            oldTop = top.get();
            if (oldTop == null)
              return null;
            newTop = oldTop.next;
        } while (! top.compareAndSet(oldTop, newTop));
        return oldTop.item;
    }
}
```

File chaînée basique

```
class Node<T> { Node<T> next; T item; }

class File<T> {
  Node<T> head, queue;
  File() { // noeud bidon en tête
      head = queue = new Node<T>();
  }

void enqueue (T item) {
    Node<T> n = new Node<T>();
    n.item = item;
    queue.next = n;
    queue = n;
}
```

```
T dequeue () {
    T res = null;
    if (head != queue) {
        head = head.next;
        res = head.item;
    }
    return res;
}
```

Non résistant à une utilisation concurrente par plusieurs activités

Synchronisation classique

Conflit enfiler/enfiler, retirer/retirer, enfiler/retirer

⇒ tout en exclusion mutuelle

```
void enqueue (T item) {
    Node<T> n = new Node<T>();
    n.item = item;
    verrou.lock();
    queue.next = n;
    queue = n;
    verrou.unlock();
}
```

```
T dequeue () {
    T res = null;
    verrou.lock();
    if (head != queue) {
        head = head.next;
        res = head.item;
    }
    verrou.unlock();
    return res;
}
```

- Bloquant définitivement si une activité s'arrête en plein milieu
- Toutes les activités sont ralenties par un unique lent
- Compétition systématique enfiler/défiler

File non bloquante

- Toute activité doit s'attendre à trouver une opération *enqueue* à moitié finie, et aider à la finir
- Invariant : l'attribut queue est toujours soit le dernier nœud, soit l'avant-dernier nœud.
- Présent dans la bibliothèque java (`java.util.concurrent.ConcurrentLinkedQueue`)

Par lisibilité, on utilise CAS (compareAndSet) défini ainsi :

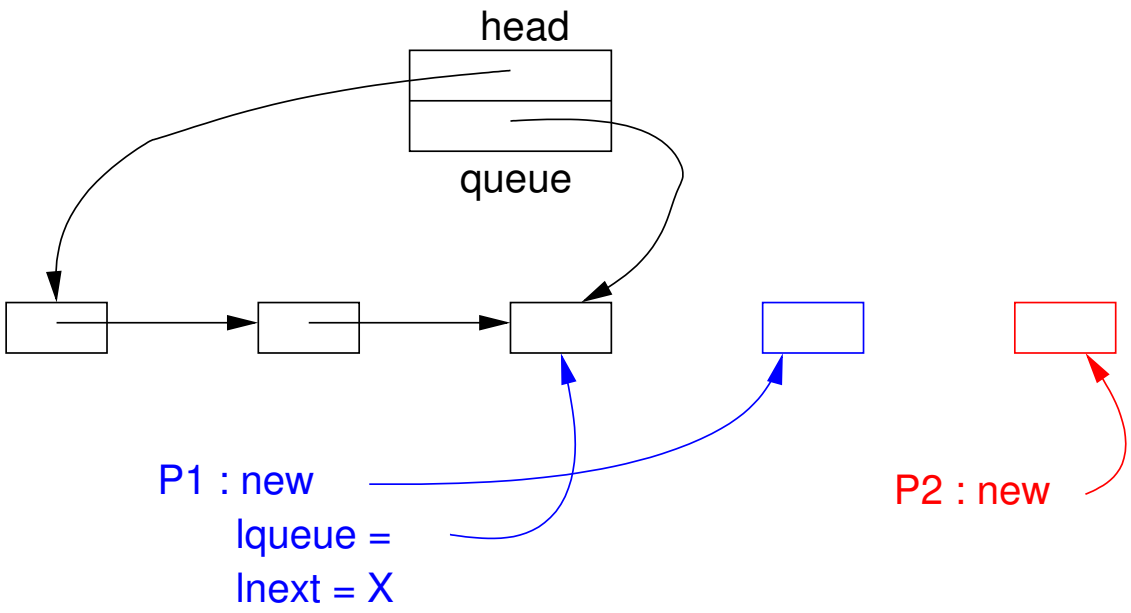
```
boolean CAS(*add, old, new) {
    atomically {
        if (*add == old ) { *add = new; return true; }
        else { return false ; }
    }
}
```

Enfiler non bloquant

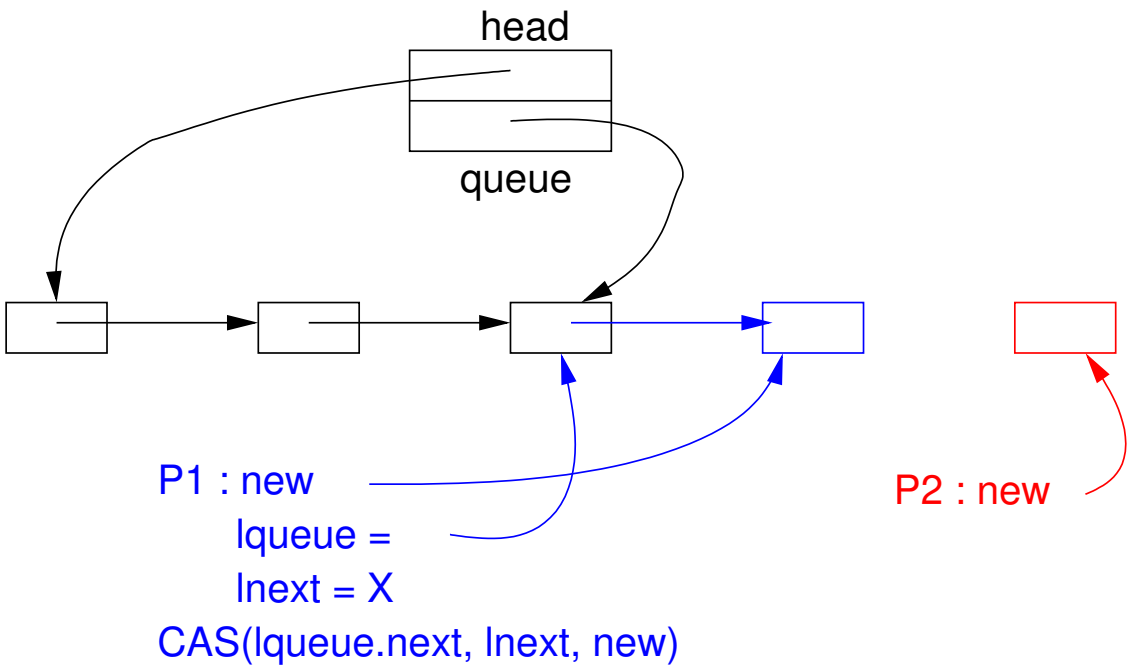
enqueue non bloquant

```
Node<T> n = new Node<T>;
n.item = item;
do {
    Node<T> lqueue = queue;
    Node<T> lnext = lqueue.next;
    if (lqueue == queue) {      // lqueue et lnext cohérents ?
        if (lnext == null) {    // queue vraiment dernier ?
            if CAS(lqueue.next, lnext, n) // essai lien nouveau noeud
                break;                    // succès !
        } else {                          // queue n'était pas le dernier noeud
            CAS(queue, lqueue, lnext); // essai mise à jour queue
        }
    }
} while (1);
CAS(queue, lqueue, n); // insertion réussie, essai m. à j. queue
```

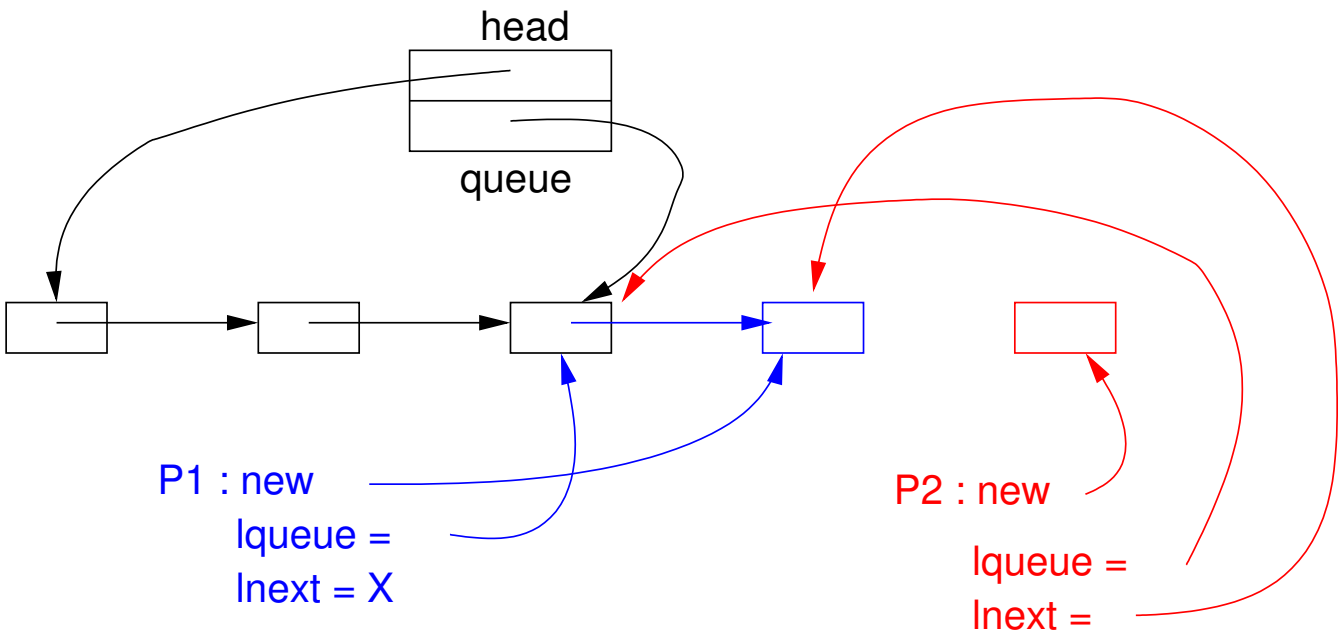
Exemple : deux enqueue concurrents



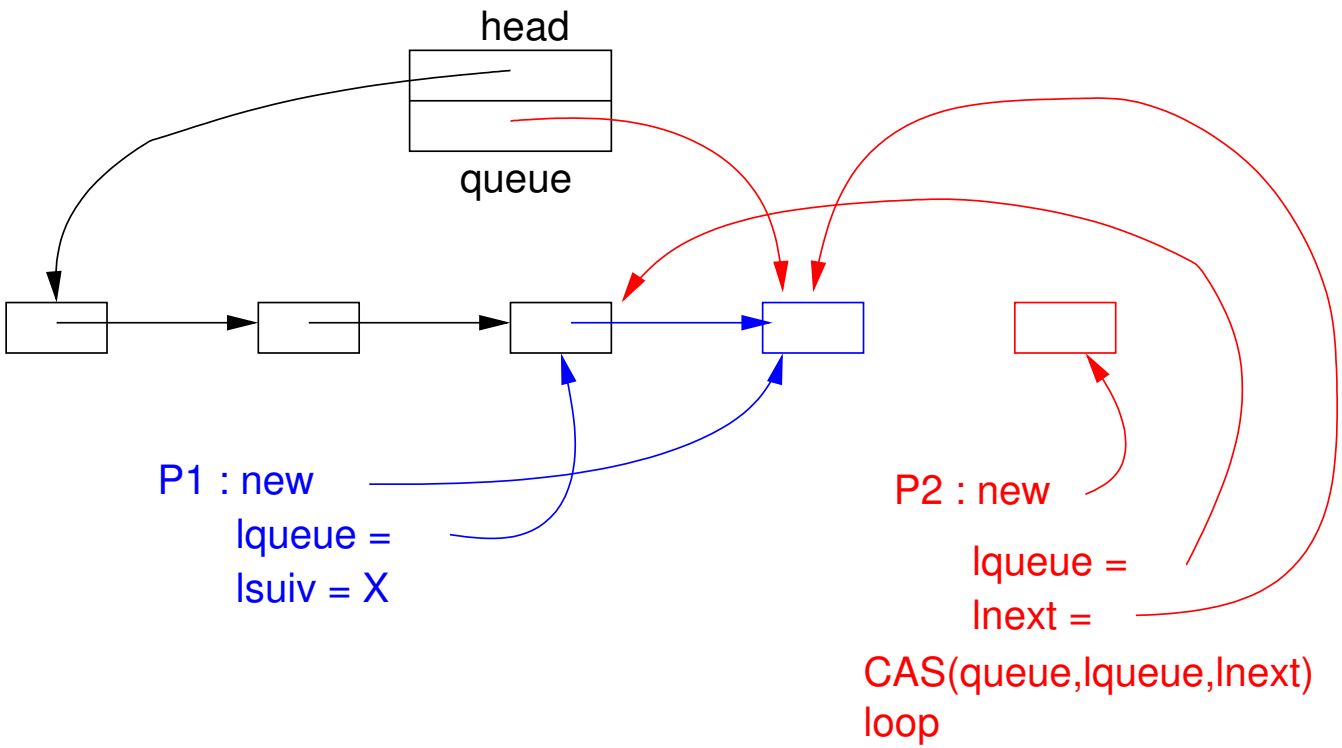
Exemple : deux enqueue concurrents



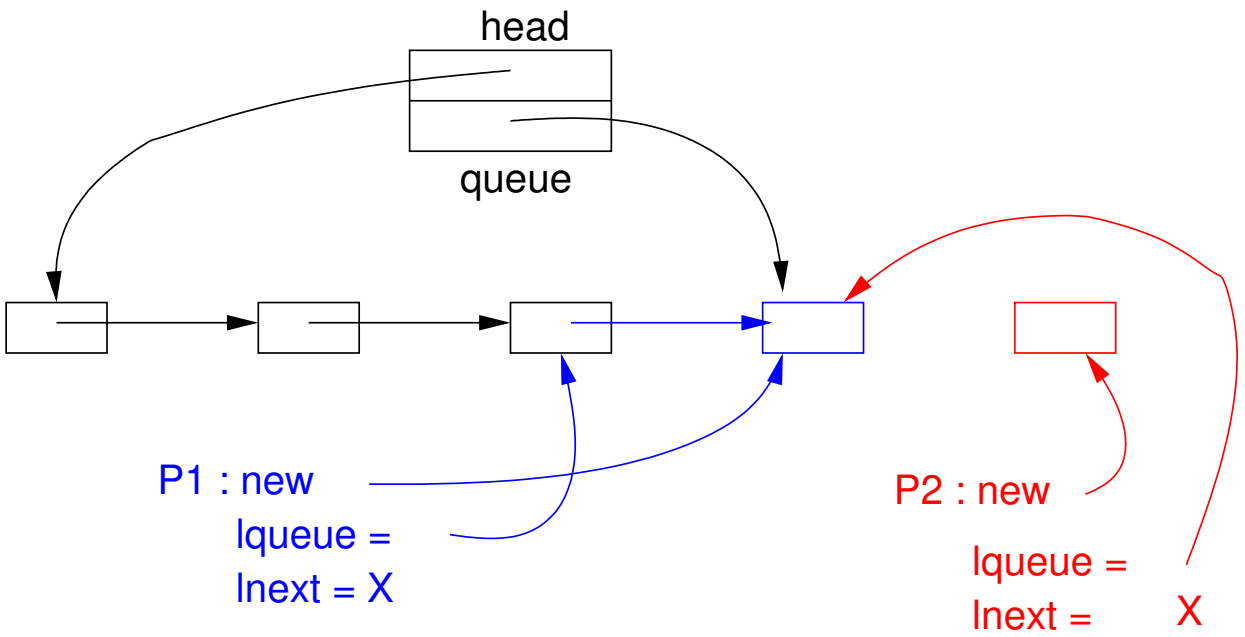
Exemple : deux enqueue concurrents



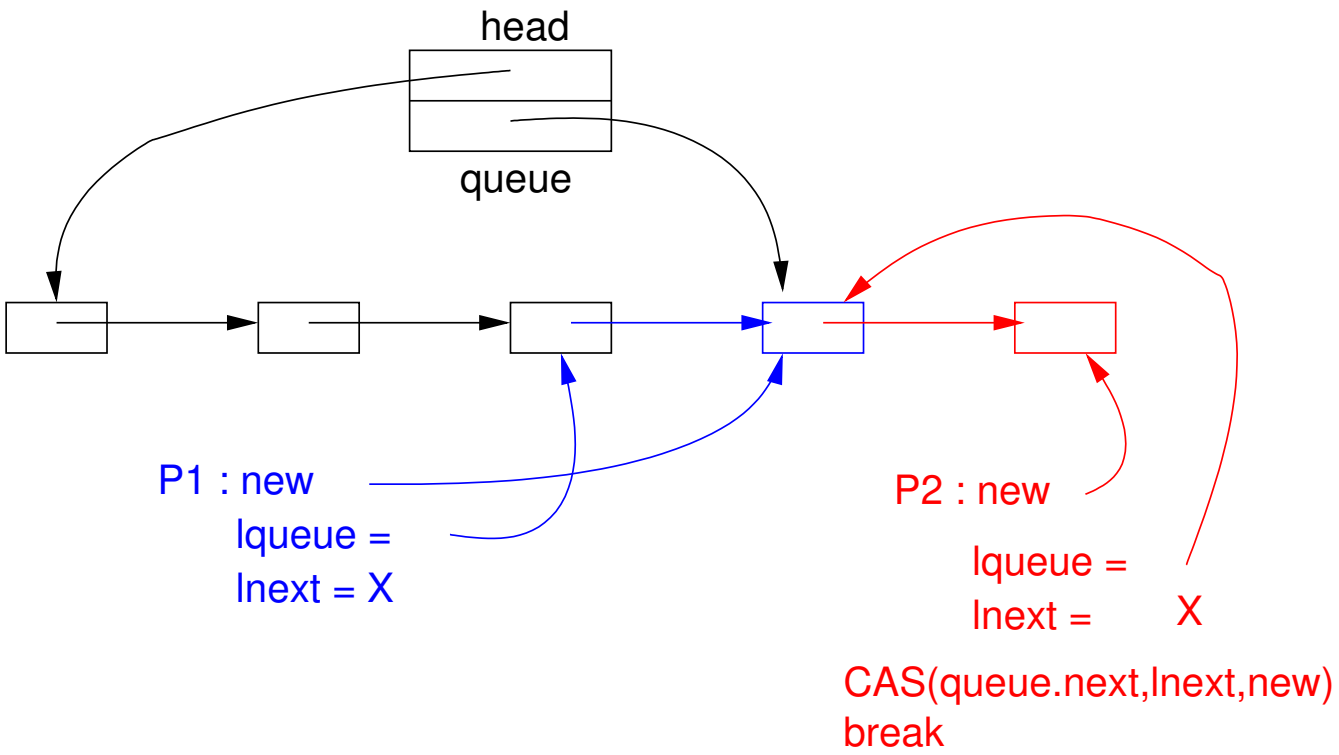
Exemple : deux enqueue concurrents



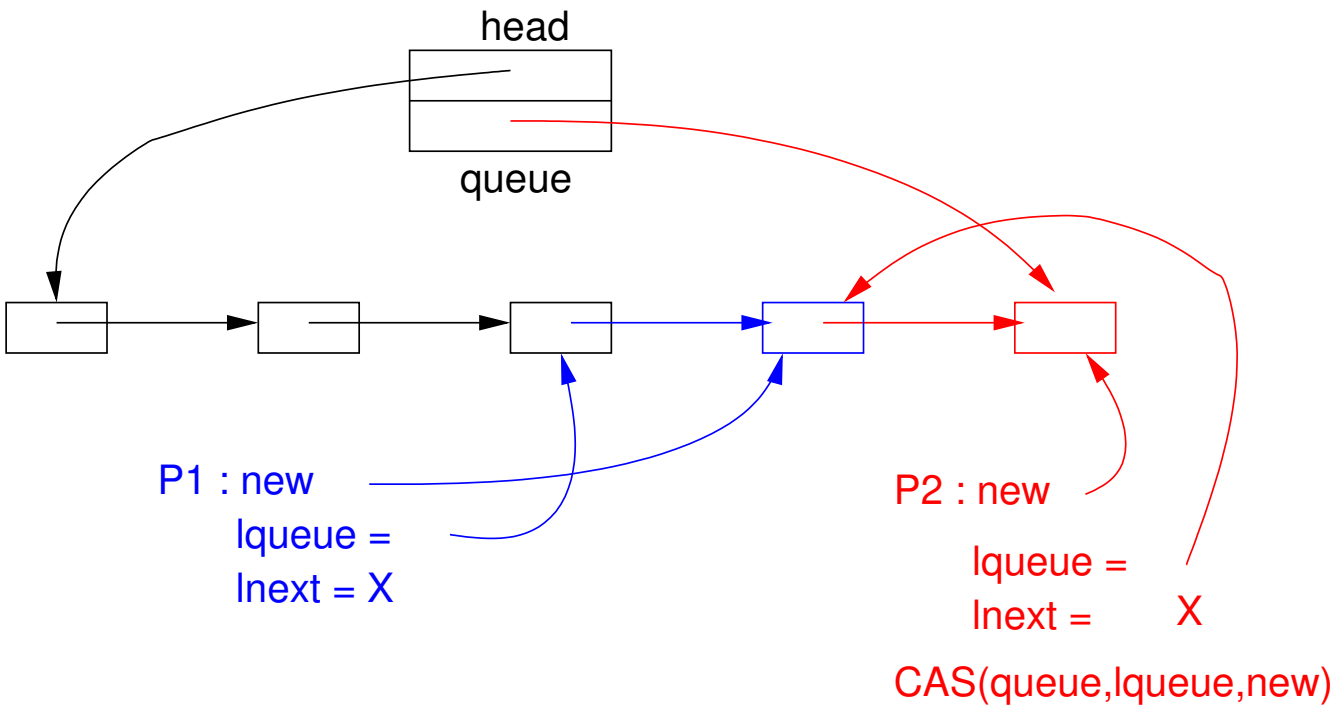
Exemple : deux enqueue concurrents



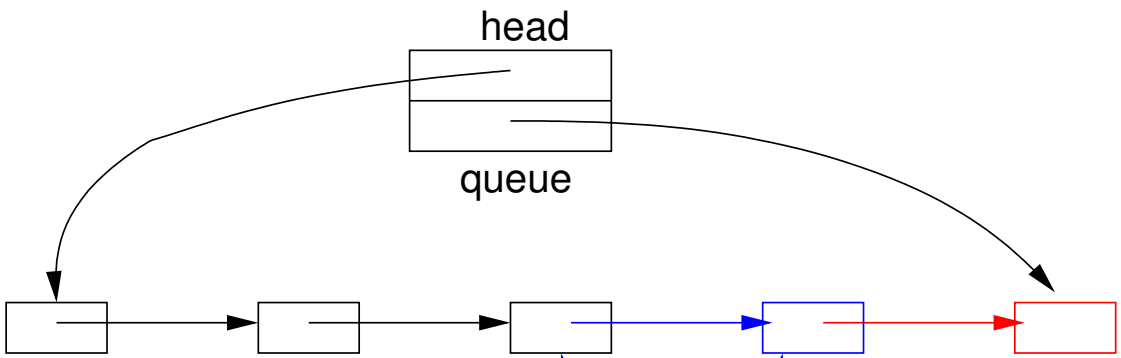
Exemple : deux enqueue concurrents



Exemple : deux enqueue concurrents



Exemple : deux enqueue concurrents

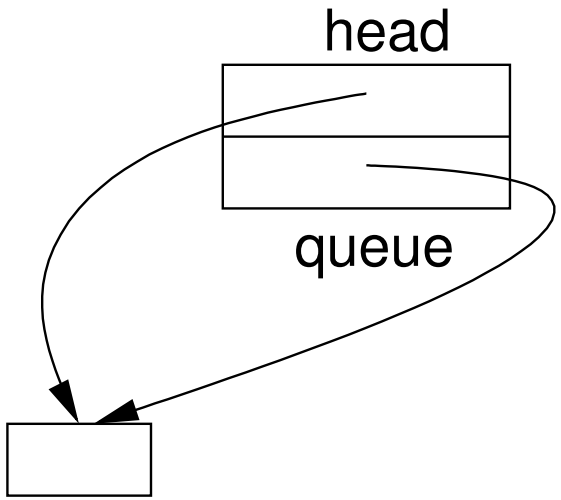


P1 : new
lqueue =
lnext = X
CAS(queue,lqueue,new) : échoue (ok)

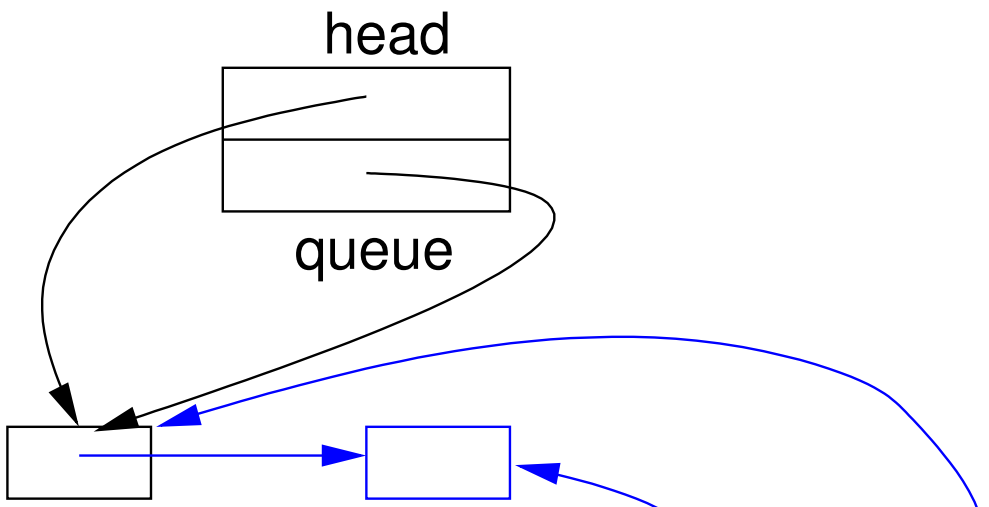
dequeue non bloquant

```
do {
    Node<T> lhead = head;
    Node<T> lqueue = queue;
    Node<T> lnext = lhead.next;
    if (lhead == head) { // lqueue, lhead, lnext cohérents ?
        if (lhead == lqueue) { // file vide ou queue à la traîne ?
            if (lnext == null)
                return null;          // file vide
            CAS(queue, lqueue, lnext); // essai mise à jour queue
        } else {                      // file non vide, prenons la tête
            res = lnext.item;
            if CAS(head, lhead, lnext)  // essai mise à jour tête
                break;                  // succès !
        }
    }
} while (1); // sinon (queue ou tête à la traîne) on recommence
return res;
```

Exemple : dequeue et enqueue concurrents



Exemple : dequeue et enqueue concurrents



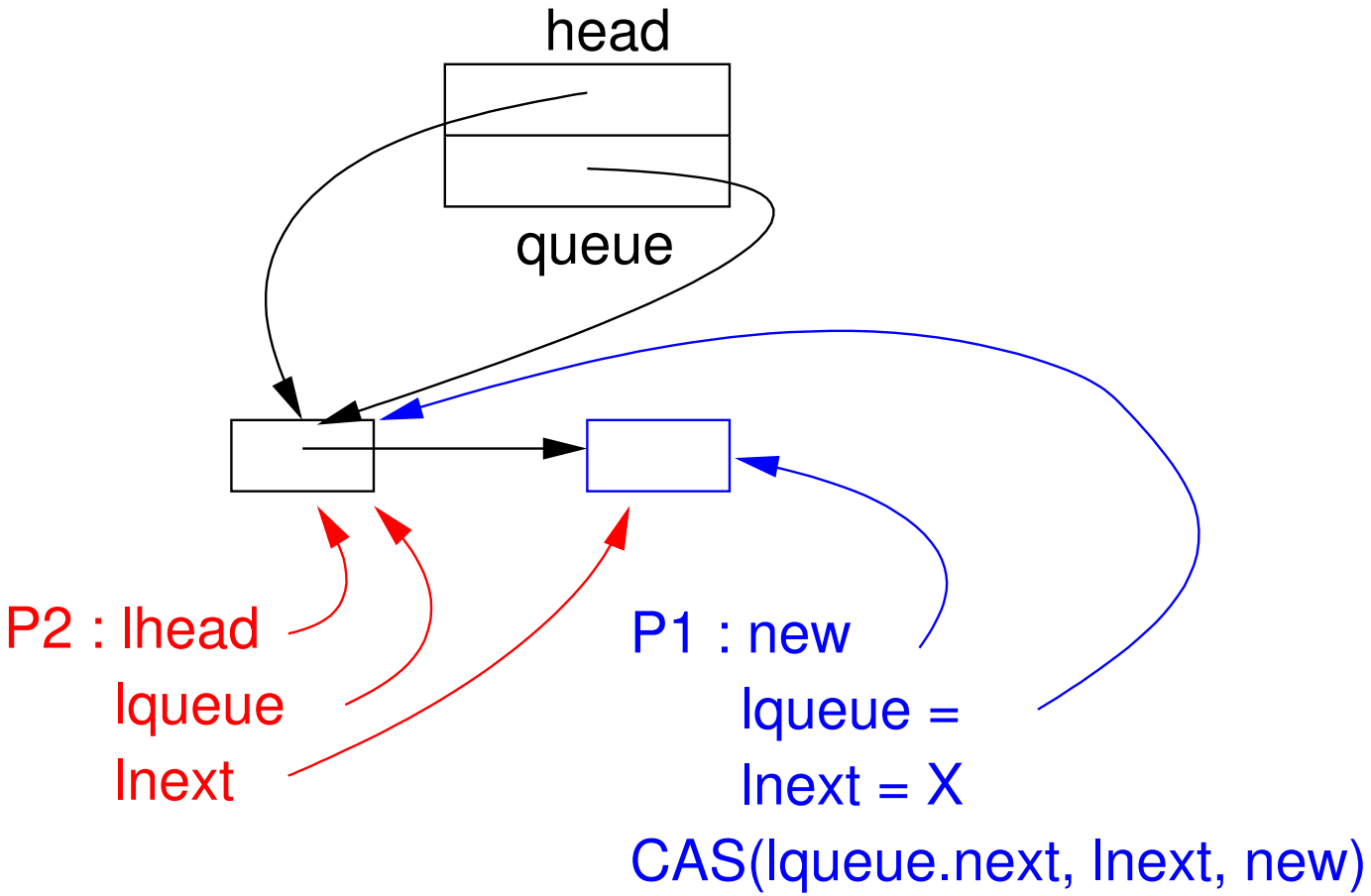
P1 : new

lqueue =

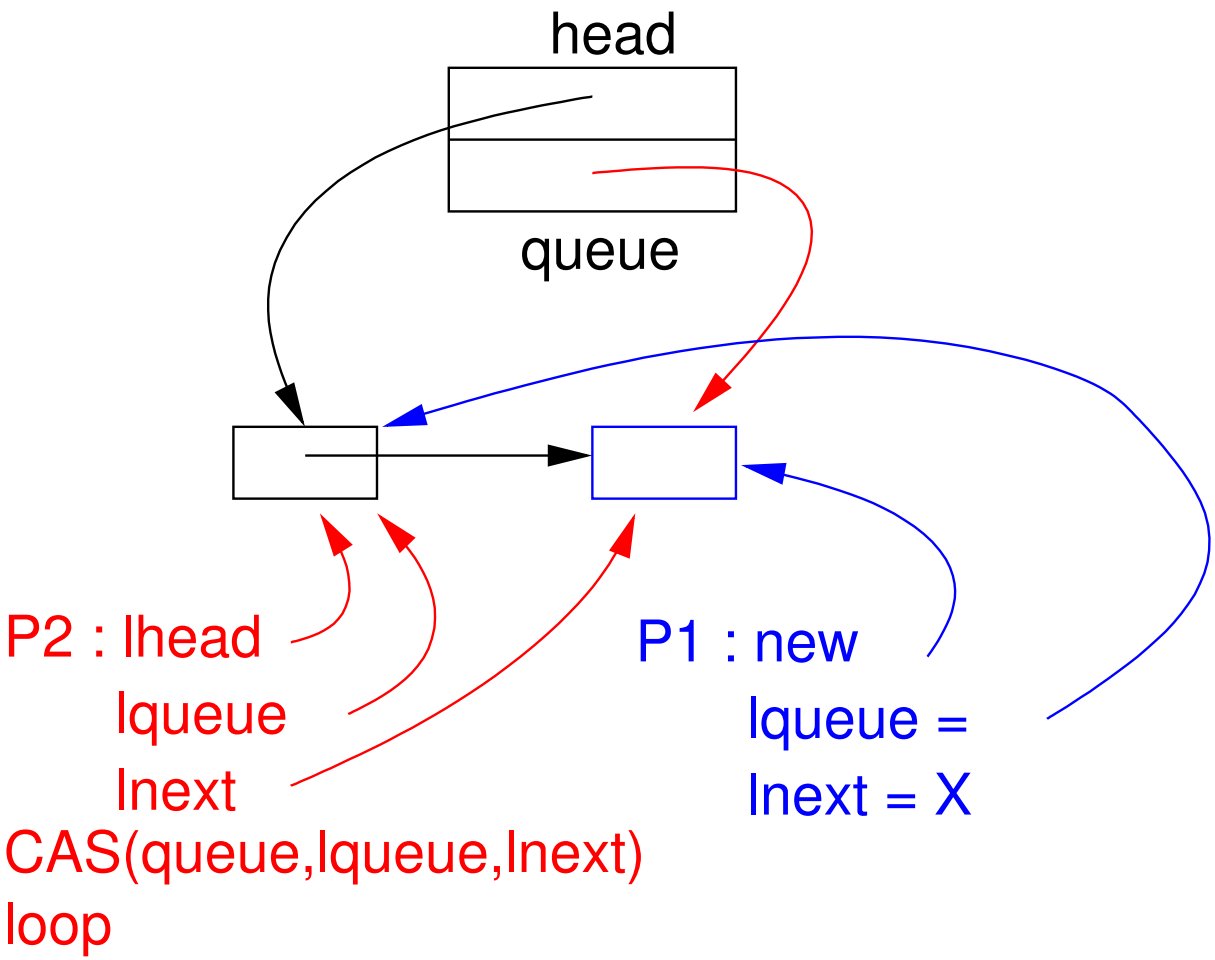
lnext = X

CAS(lqueue.next, lnext, new)

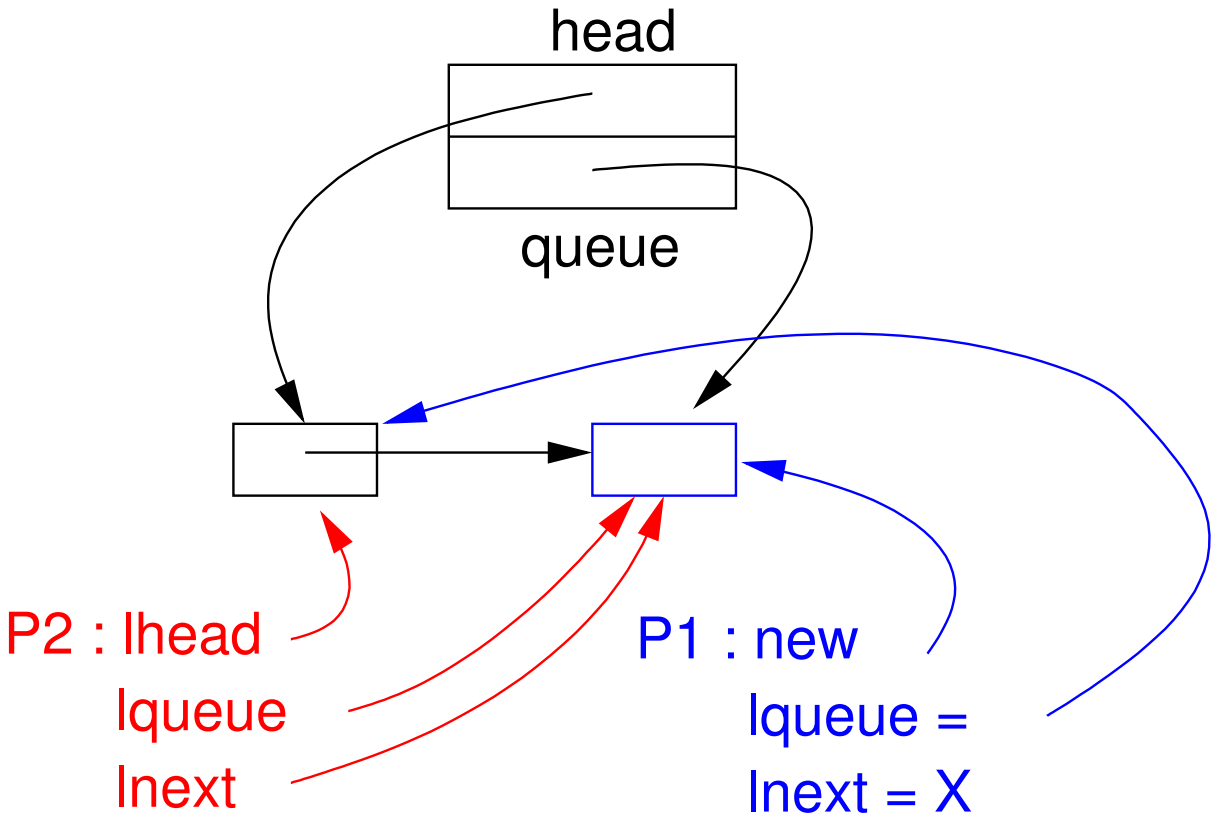
Exemple : dequeue et enqueue concurrents



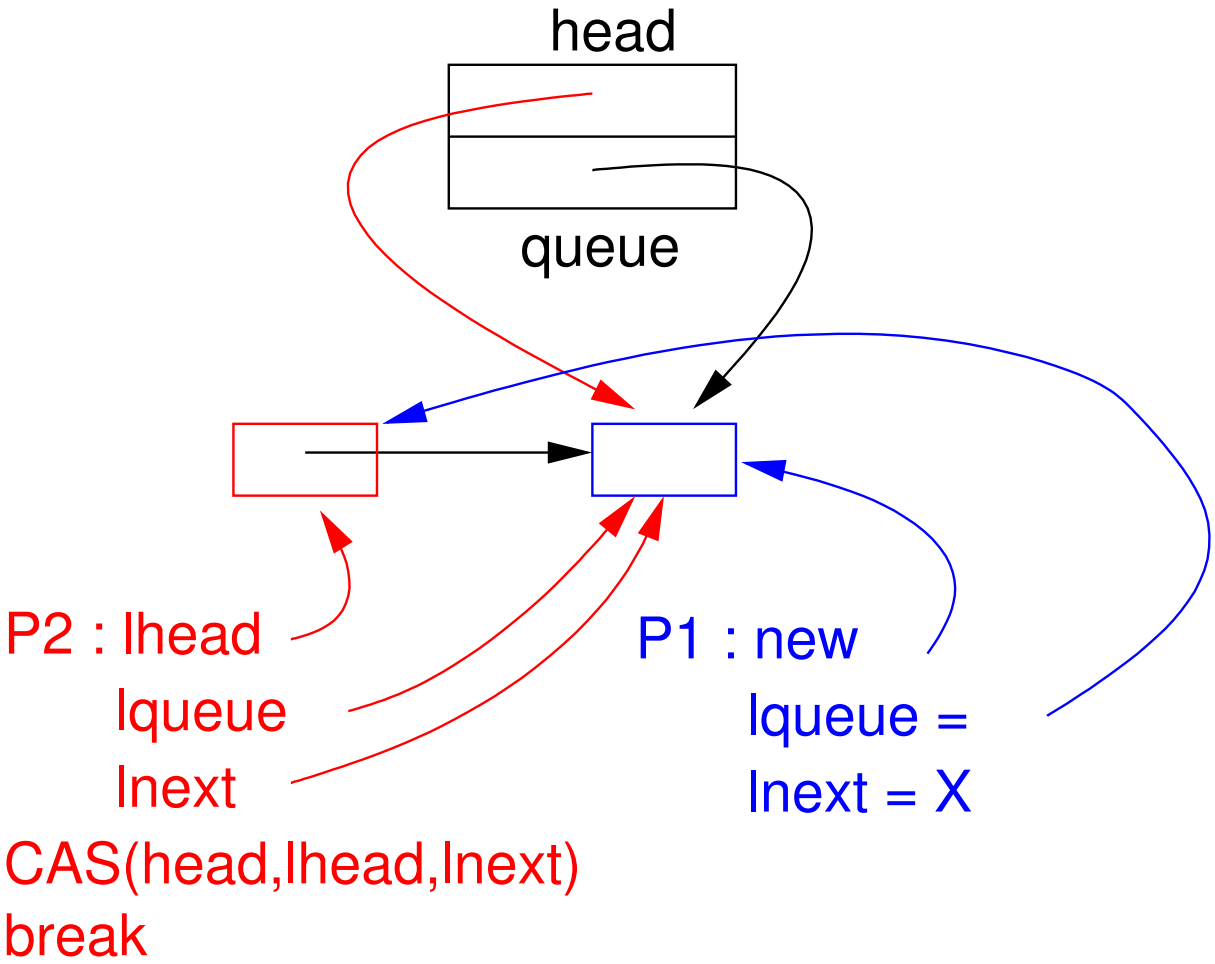
Exemple : dequeue et enqueue concurrents



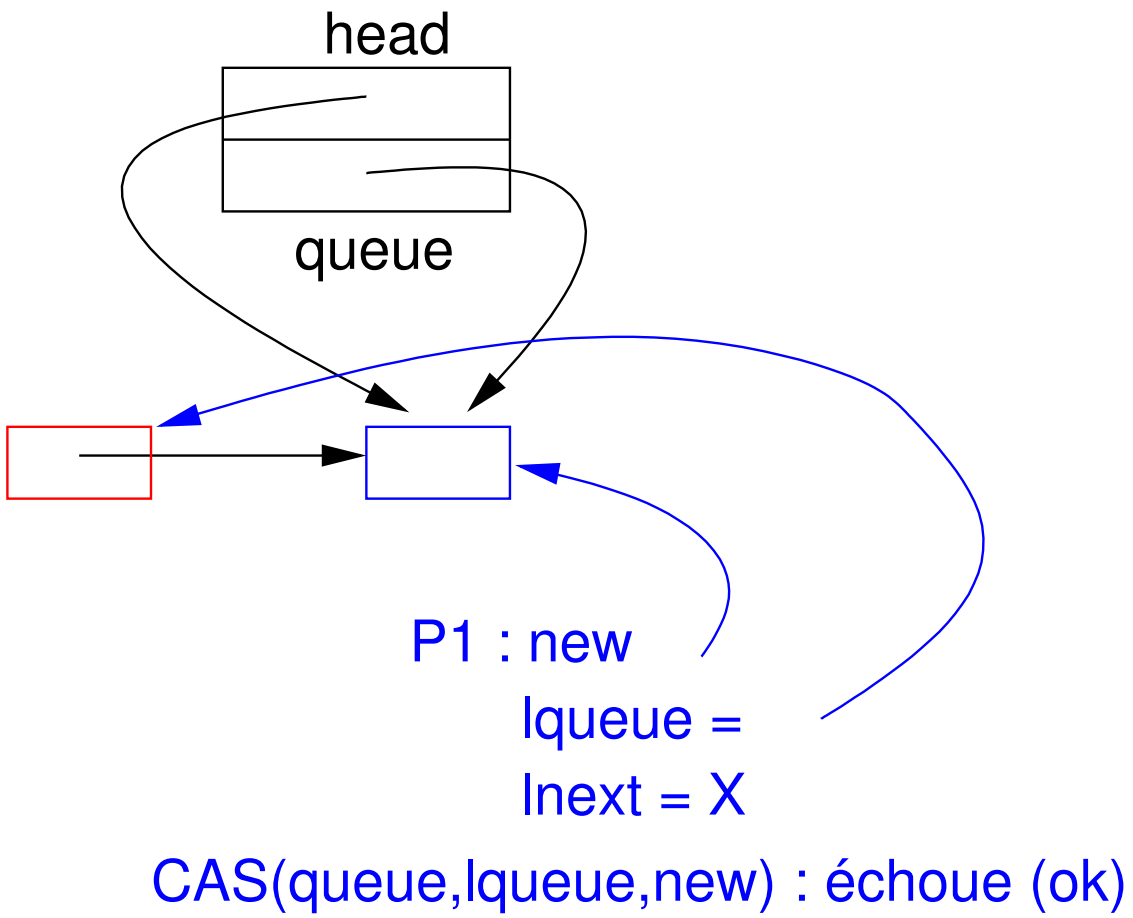
Exemple : dequeue et enqueue concurrents



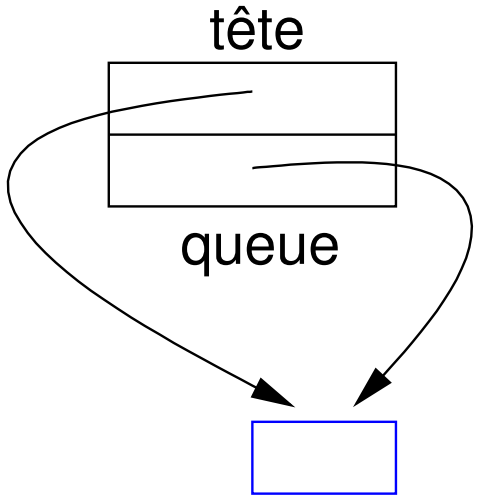
Exemple : dequeue et enqueue concurrents



Exemple : dequeue et enqueue concurrents



Exemple : dequeue et enqueue concurrents



Problème A-B-A

- L'algorithme précédent n'est correct qu'en absence de recyclage des cellules libérées par dequeue
- Problème A-B-A :
  1. A_1 lit x et obtient a
  2. A_2 change x en b et libère a
  3. A_3 demande un objet libre et obtient a
  4. A_3 change x en a
  5. A_1 effectue CAS(x,a,...), qui réussit et lui laisse croire que x n'a pas changé depuis sa lecture

Solutions au problème A-B-A

- Compteur de générations, incrémenté à chaque modification
  $\langle a, \text{gen } i\rangle \neq \langle a, \text{gen } i+1\rangle$
  Nécessite un CAS2(x,a,gen,i,...)
  (`java.util.concurrent.atomic.AtomicStampedReference`)
- Instructions load-link / store-conditional (LL/SC) :
  - Load-link renvoie la valeur courante d'une case mémoire
  - Store-conditional écrit une nouvelle valeur à condition que la case mémoire n'a pas été écrite depuis le dernier load-link.
  - (les implantations matérielles imposent souvent des raisons supplémentaires d'échec de SC : imbrication de LL, écriture sur la ligne de cache voire écriture quelconque...)
- Ramasse-miette découplé : retarder la réutilisation d'une cellule (*Hazard pointers*). L'allocation/libération devient alors le facteur limitant de l'algorithme.

Plan

Conclusion

+ performant, même avec beaucoup d'activités
+ résistant à l'arrêt temporaire ou définitif d'une activité
– structure de données ad-hoc
– implantation fragile, peu réutilisable, pas extensible
– implantation très complexe, à réserver aux experts
– implantation liée à une architecture matérielle
– nécessité de prouver la correction
+ bibliothèques spécialisées
  `java.util.concurrent.ConcurrentLinkedQueue`
  `j.u.concurrent.atomic.AtomicReference.compareAndSet`
  `j.u.concurrent.atomic.AtomicInteger`