Systèmes concurrents

Philippe Quéinnec

14 septembre 2012

Dixième partie

Synchronisation non bloquante

Plan

Objectifs de la synchronisation non bloquante

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

Principes généraux

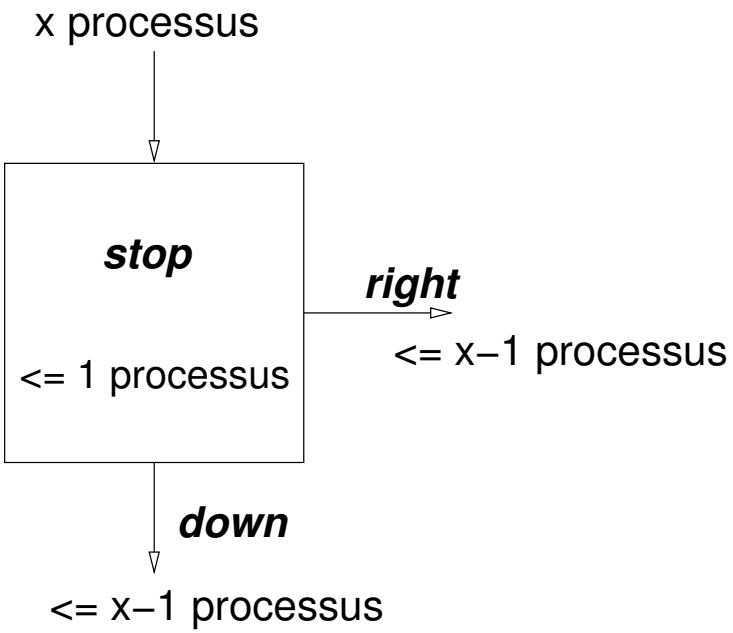
On se concentre sur le partage des structures de données.
Problème habituellement résolu par l'exclusion mutuelle.

- Plus de verrous !
- Instructions atomiques processeur suffisamment évoluées
- Boucle active en cas de conflit d'accès
- Limiter le plus possible la zone de conflit

Note : wait-free $\neq$ lock-free (mais on simplifie ici)

Plan

Splitter



- x (indéterminé) processus appellent concurremment (ou pas) le splitter
- au plus un processus termine avec *stop*
- si $x = 1$, le processus termine avec *stop*
- au plus $(x - 1)$ processus terminent avec *right*
- au plus $(x - 1)$ processus terminent avec *down*

Splitter

« Registres »

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

Implantation non bloquante

Deux « registres » partagés : X (init $\forall$) et Y (init faux)
Chaque processus a un identifiant unique id_i.

direction(id_i)
  $X \leftarrow id_i$;
  if Y **then** $dir_i \leftarrow$ *right*;
  else $Y \leftarrow$ *true*;
    if $(X = id_i)$ **then** $dir_i \leftarrow$ *stop*;
    else $dir_i \leftarrow$ *down*; **endif**
  endif
  return dir_i;

Renommage

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

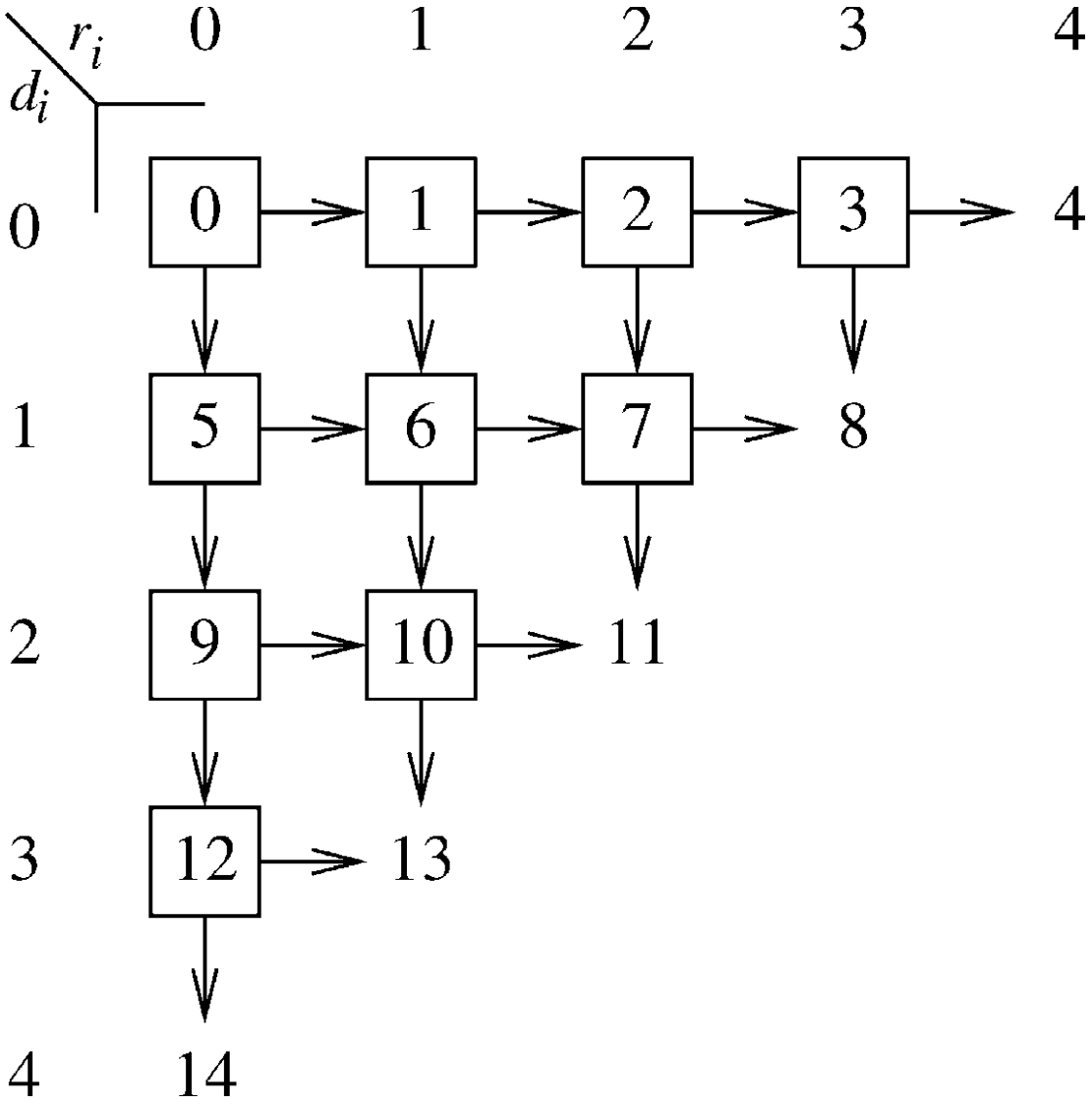
Solution à base de verrous

- Distributeur de numéro accédé en exclusion mutuelle
- $M = n$
- Complexité temporelle : O(1) pour un numéro, O(n) pour tous

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



Renommage non bloquant

get_name(id_i)

$d_i \leftarrow 0$; $r_i \leftarrow 0$; $term_i \leftarrow false$;
while $((d_i + r_i < n - 1) \wedge \neg term_i)$ **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 $(n * d_i + r_i - (d_i(d_i - 1)/2))$
% *le nom en position* d_i, r_i *de la grille*

File basique

```
class Nœud<T> {
   Nœud<T> suiv;
   T item;
}

class File<T> {
  Nœud<T> tête;
  Nœud<T> queue;

  File() {
      // nœud bidon en tête
      tête = queue = new Nœud<T>();
  }
```

File basique

```
void enfiler (T item) {
    Nœud<T> n = new Nœud<T>();
    n.item = item;
    queue.suiv = n;
    queue = n;
}
```

```
T défiler () {
    T rés = null;
    if (tête != queue) then
        tête = tête.suiv;
        rés = tête.item;
    endif
    return rés;
}
```

Synchronisation classique

- Conflit enfiler/enfiler (queue est utilisé en deux endroits)
- Conflit retirer/retirer (tête est utilisé en deux endroits)
- Conflit enfiler/retirer (file avec 1 seul élément)
- $\Rightarrow$ tout en exclusion mutuelle

File basique avec verrou

```
void enfiler (T item) {
    Nœud<T> n = new Nœud<T>();
    n.item = item;
    verrou.lock();
    queue.suiv = n;
    queue = n;
    verrou.unlock();
}
```

```
T défiler () {
    T rés = null;
    verrou.lock();
    if (tête != queue) then
        tête = tête.suiv;
        rés = tête.item;
    endif
    verrou.unlock();
    return rés;
}
```

- Bloquant définitivement si un processus s'arrête en plein milieu
- Tous les processus sont ralentis par un unique lent
- Compétition systématique enfiler/défiler

Compare-and-set

Hypothèses

- Lectures et écritures atomiques (« registres »)
- Pas d'interférence due aux caches en multi-processeurs
- Une instruction atomique évoluée

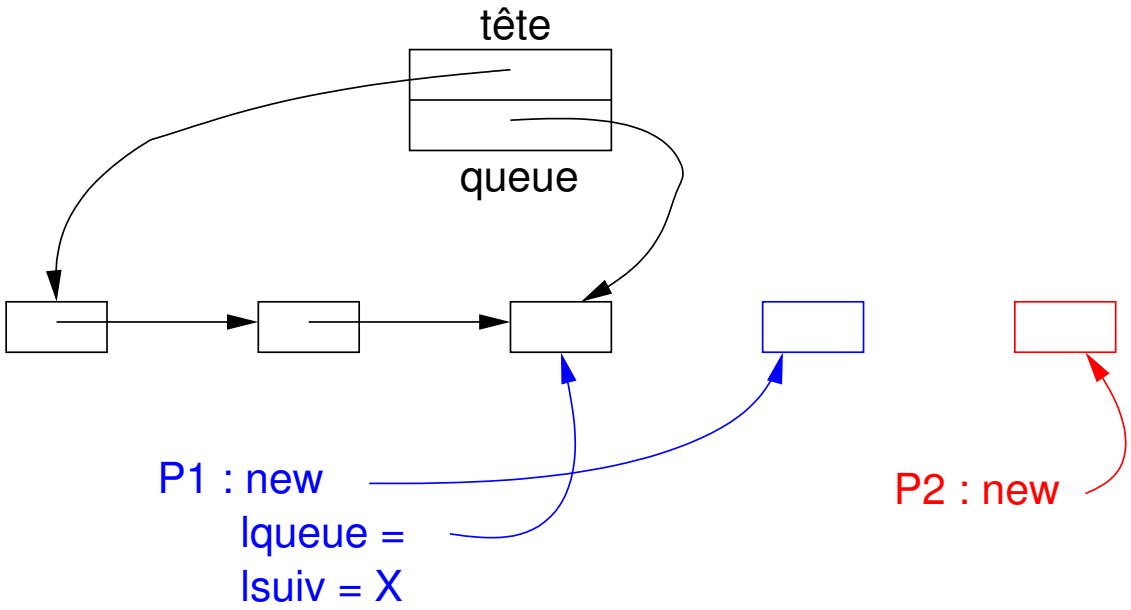
compare-and-set

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

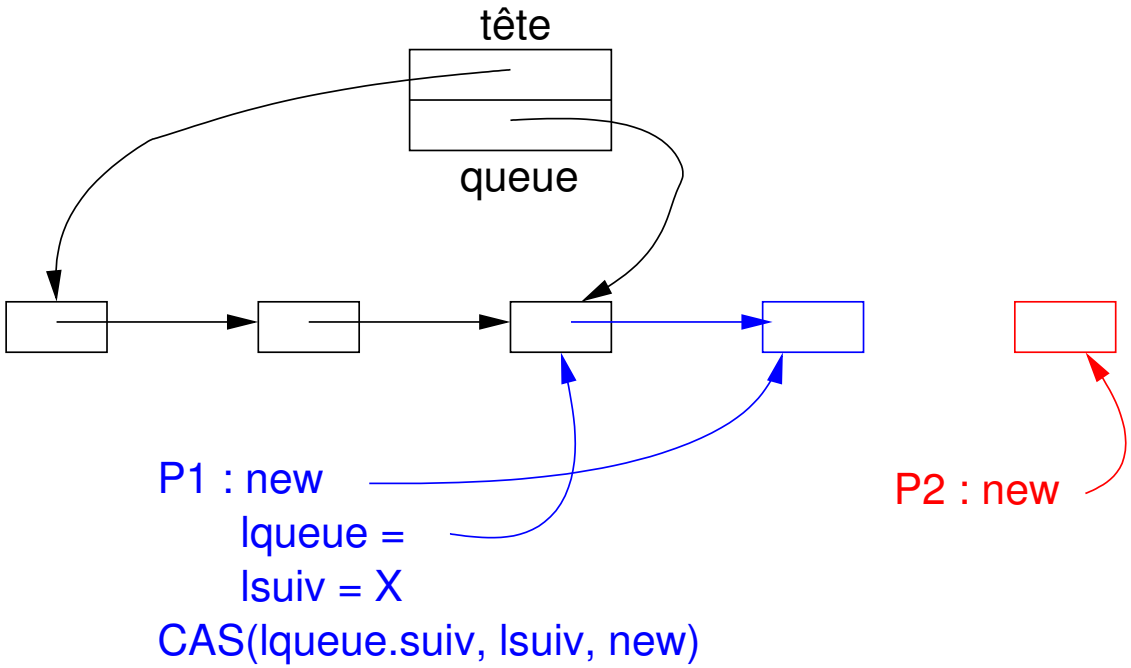
enfiler non bloquant

```
Nœud<T> n = new Nœud<T>;
n.item = item;
loop
  Nœud<T> lqueue = queue;
  Nœud<T> lsuiv = lqueue.suiv;
  if lqueue == queue then      lqueue et lsuiv cohérents ?
    if lsuiv == null then      queue vraiment dernier ?
      if CAS(lqueue.suiv, lsuiv, n) essai lien nouveau nœud
        break;                 succès !
      endif
    else              queue n'était pas le dernier nœud
      CAS(queue, lqueue, lsuiv); essai m-à-j queue
    endif
  endif
endloop
CAS(queue, lqueue, n); insertion réussie, essai m-à-j queue
```

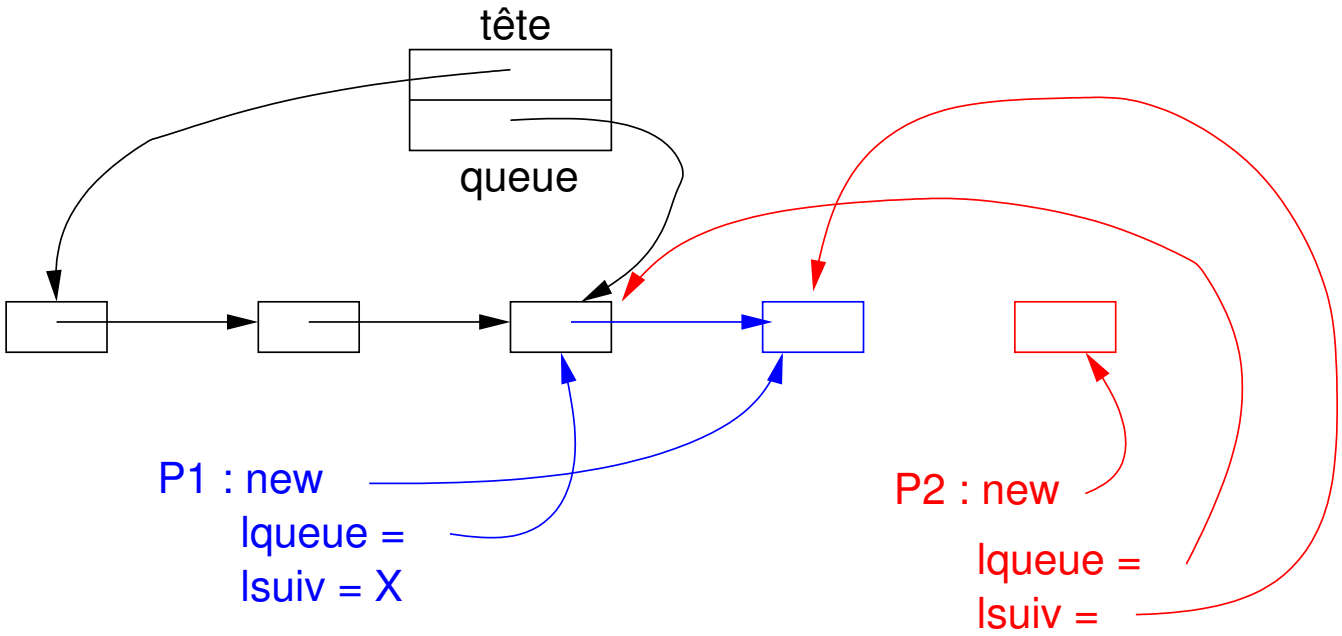
Exemple : deux `enfiler` concurrents



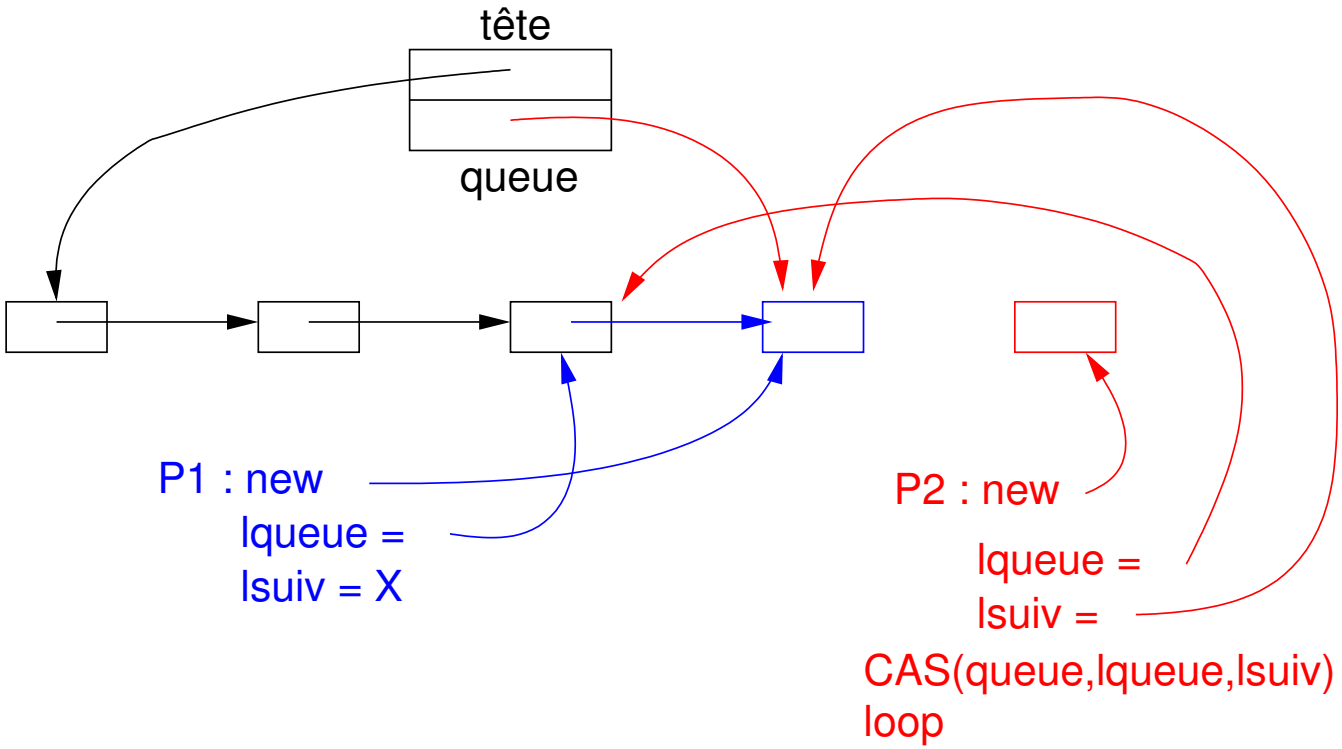
Exemple : deux `enfiler` concurrents



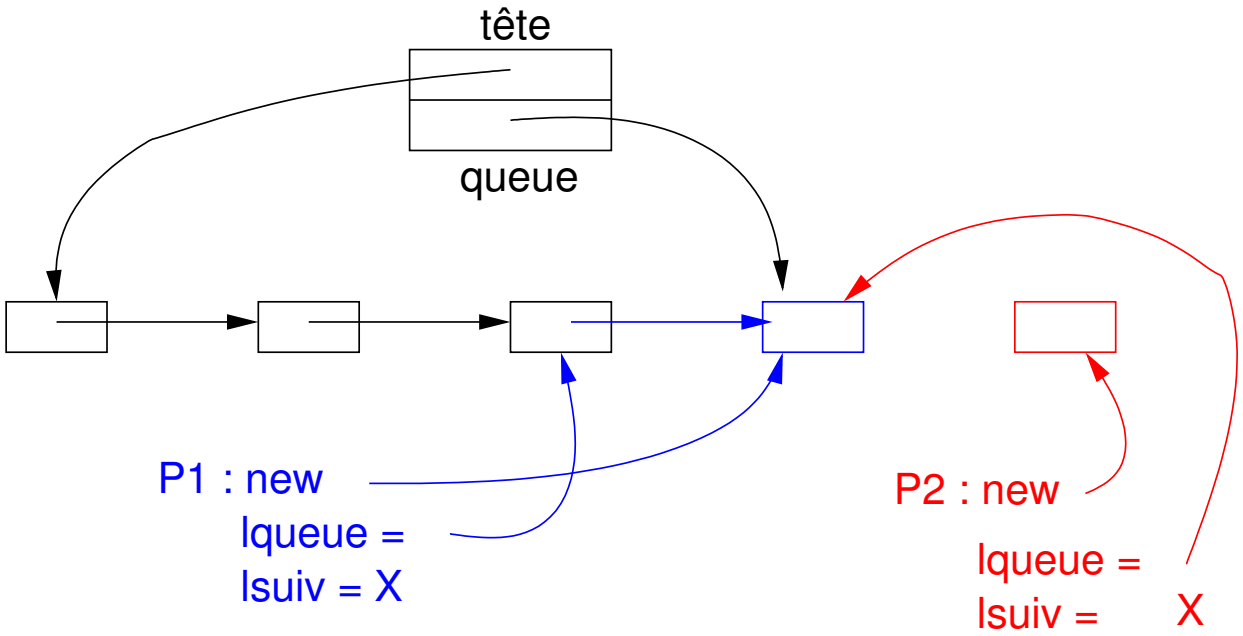
Exemple : deux `enfiler` concurrents



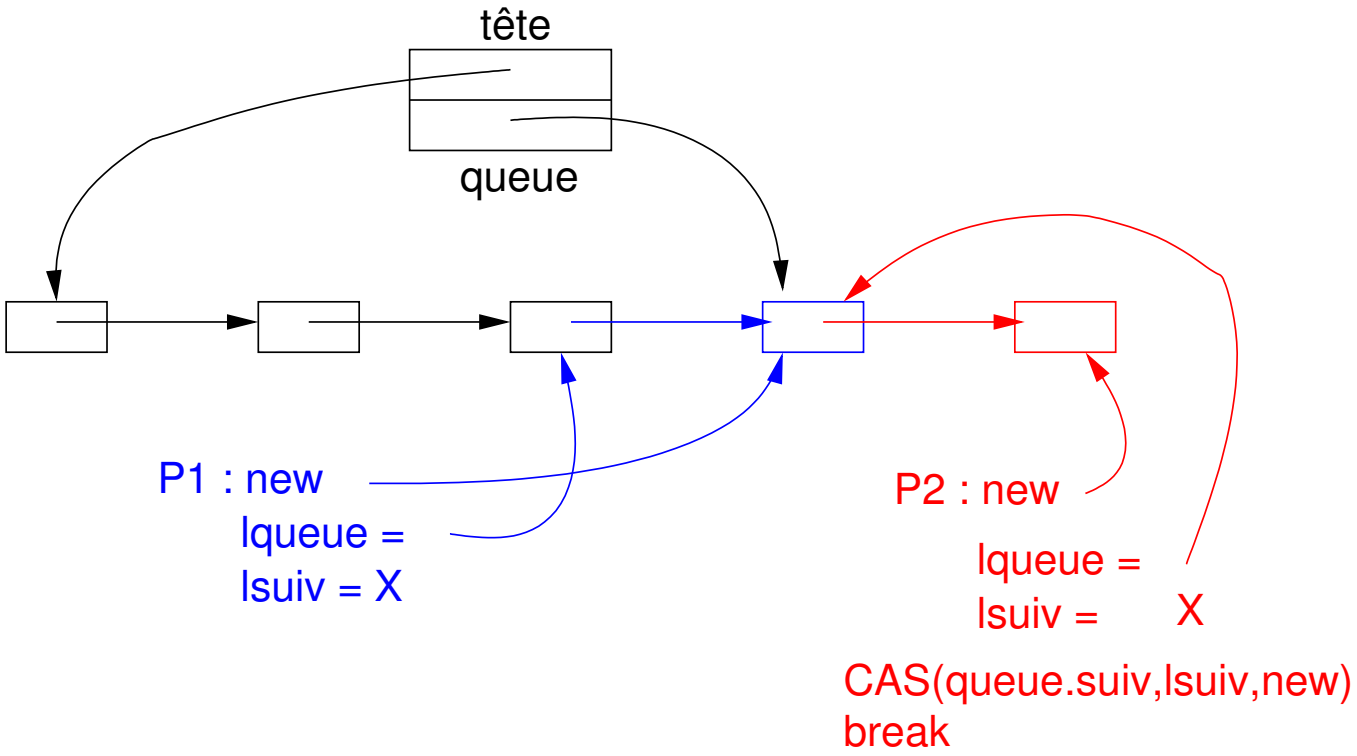
Exemple : deux `enfiler` concurrents



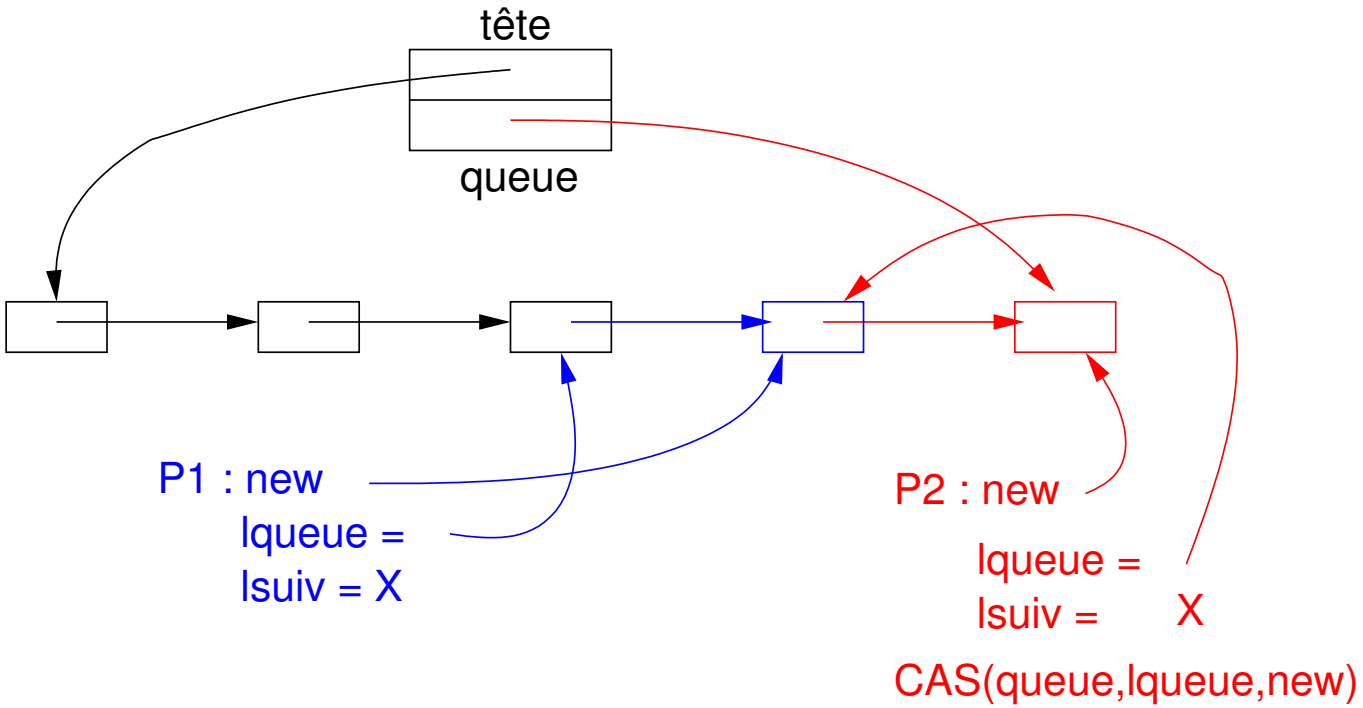
Exemple : deux `enfiler` concurrents



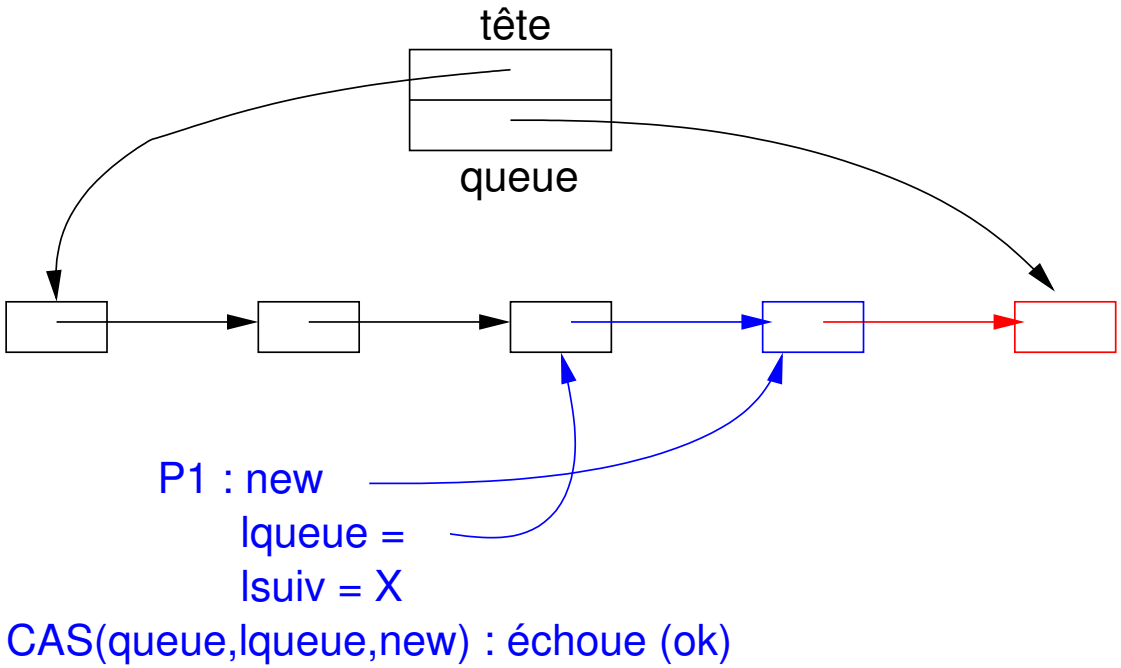
Exemple : deux `enfiler` concurrents



Exemple : deux `enfiler` concurrents



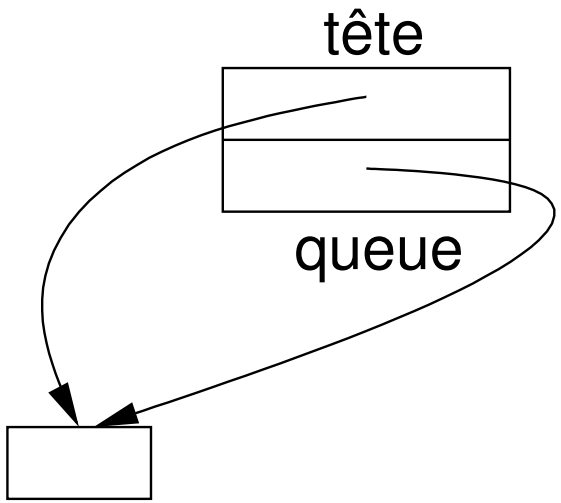
Exemple : deux `enfiler` concurrents



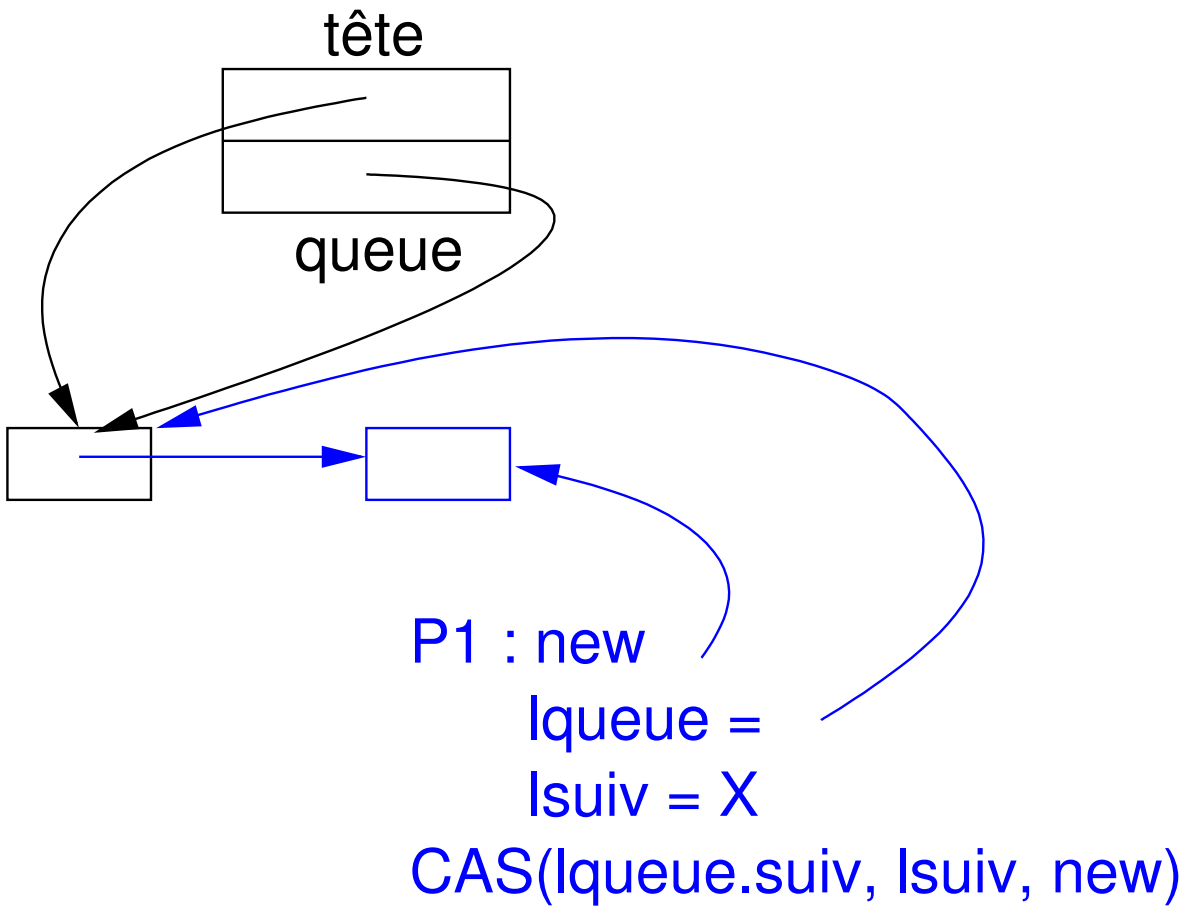
défiler non bloquant

```
loop
  Nœud<T> ltête = tête; Nœud<T> lqueue = queue;
  Nœud<T> lsuiv = ltête.suiv;
  if ltête == tête then  lqueue, ltête, lsuiv cohérents ?
    if ltête == lqueue then  file vide ou queue à la traîne ?
      if (lsuiv == null) then
          return null;        file vide
      endif
      CAS(queue, lqueue, lsuiv); essai m-à-j queue
    else
      res = lsuiv.item;
      if CAS(tête, ltête, lsuiv) then essai m-à-j tête
          break;                          succès !
      endif
    endif
  endif
endloop sinon (queue ou tête à la traîne) on recommence
return res;
```

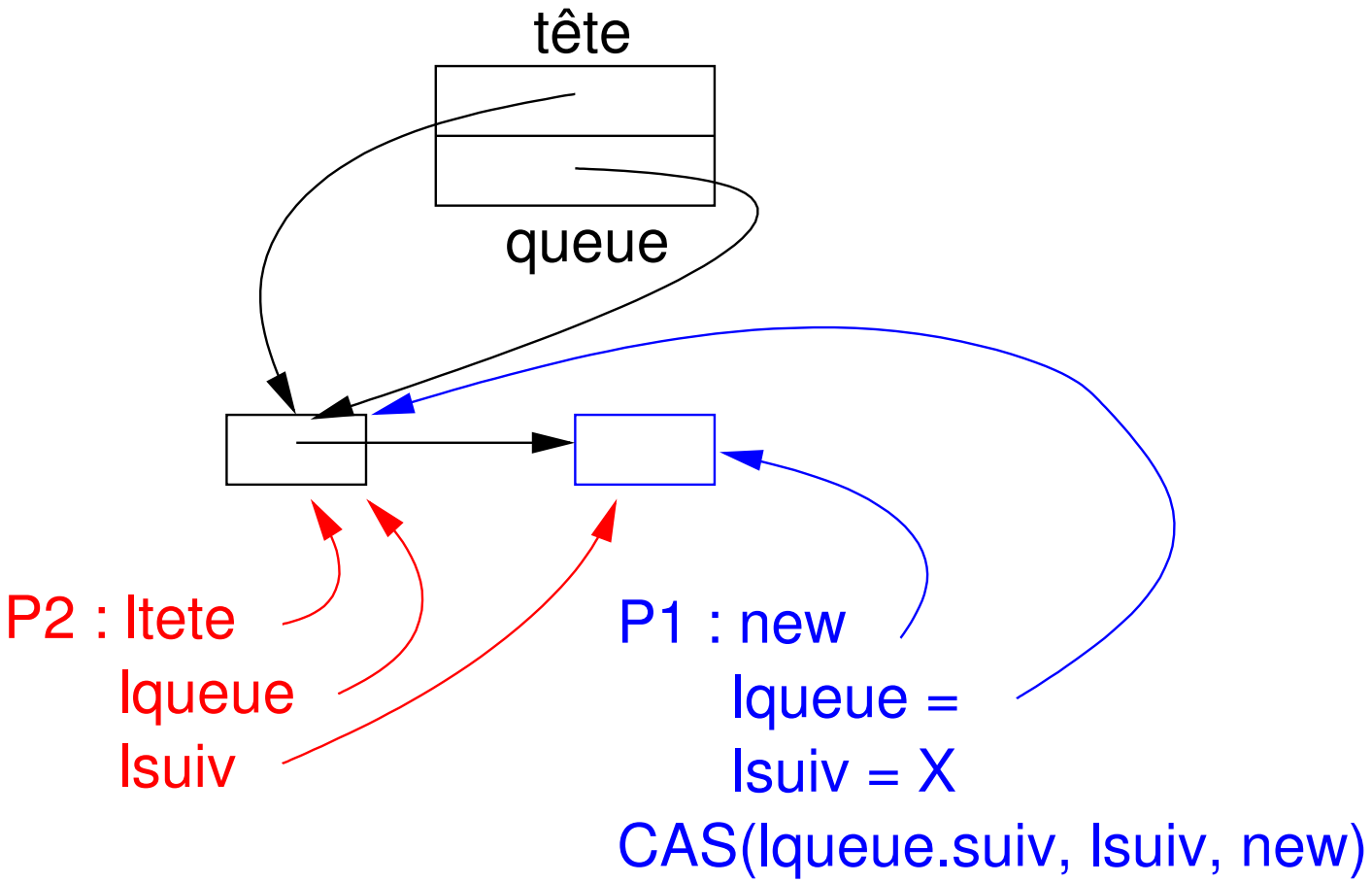
Exemple : `défiler` et `enfiler` concurrents



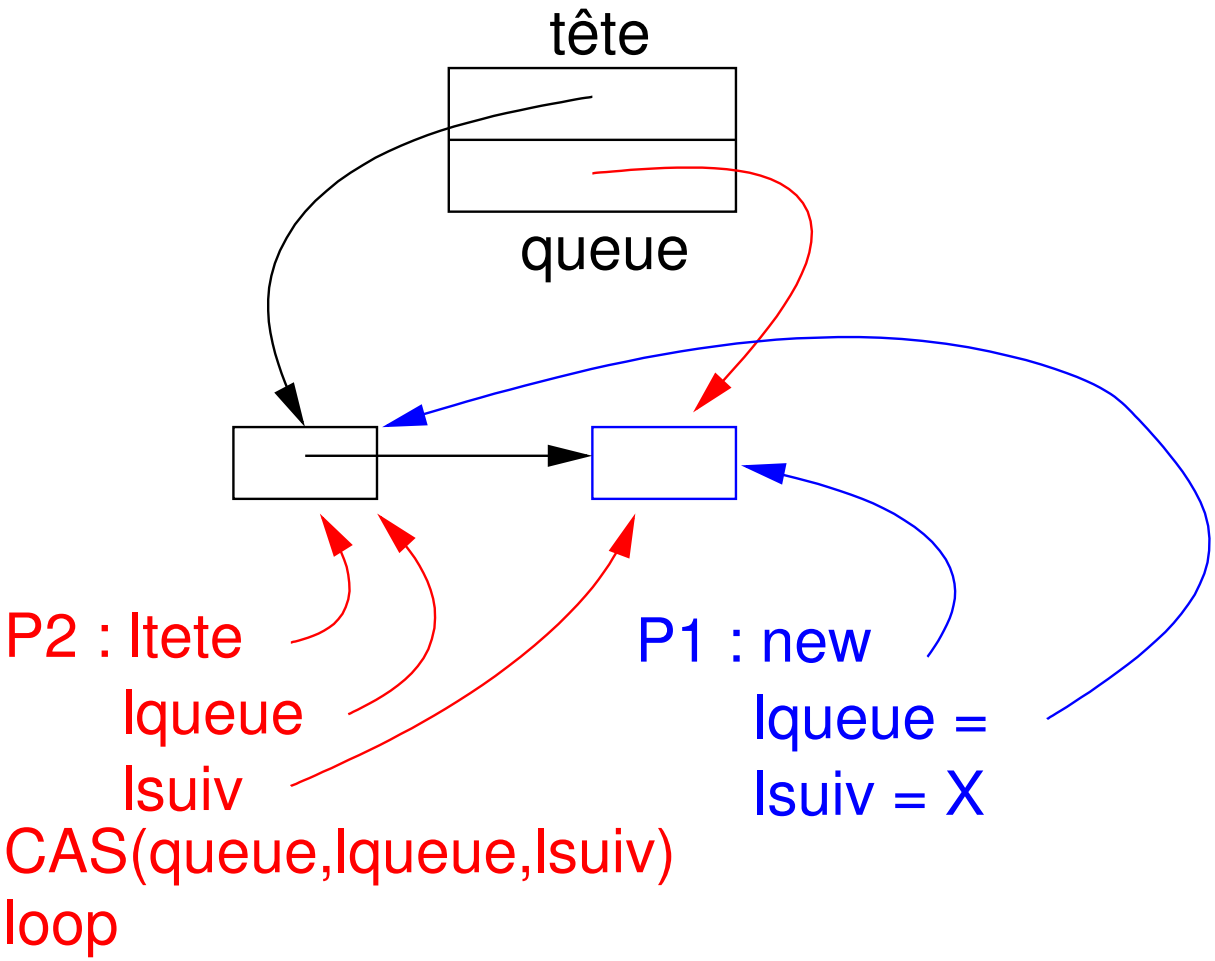
Exemple : `défiler` et `enfiler` concurrents



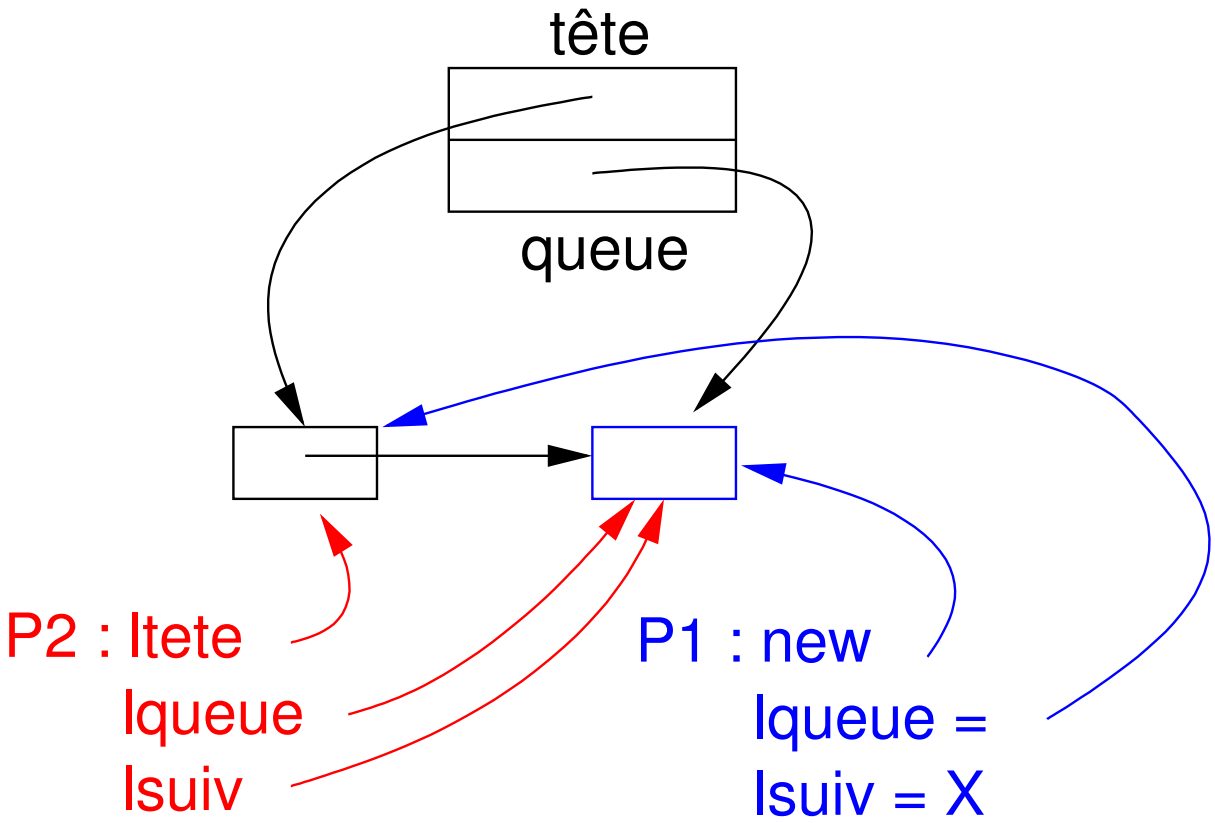
Exemple : `défiler` et `enfiler` concurrents



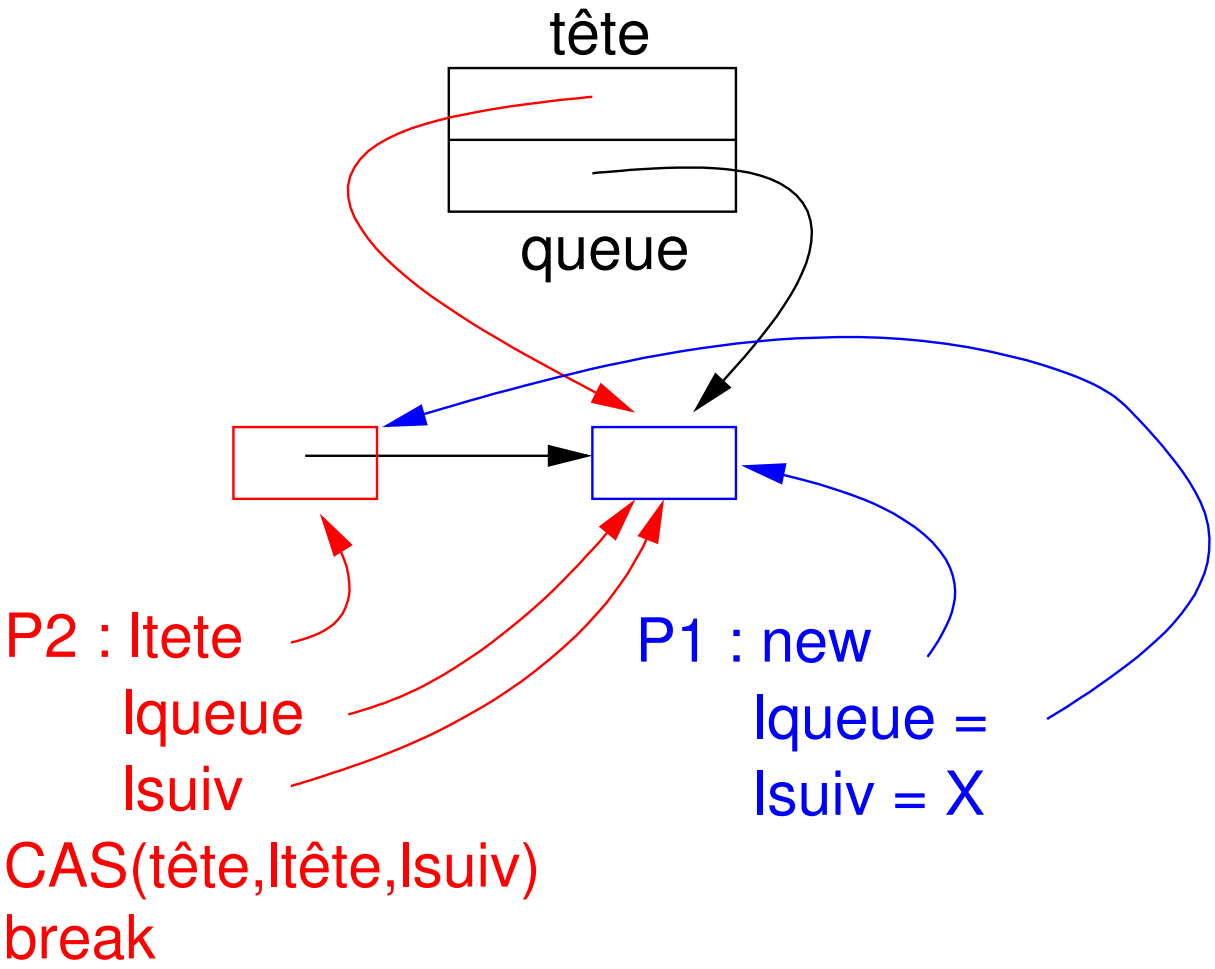
Exemple : `défiler` et `enfiler` concurrents



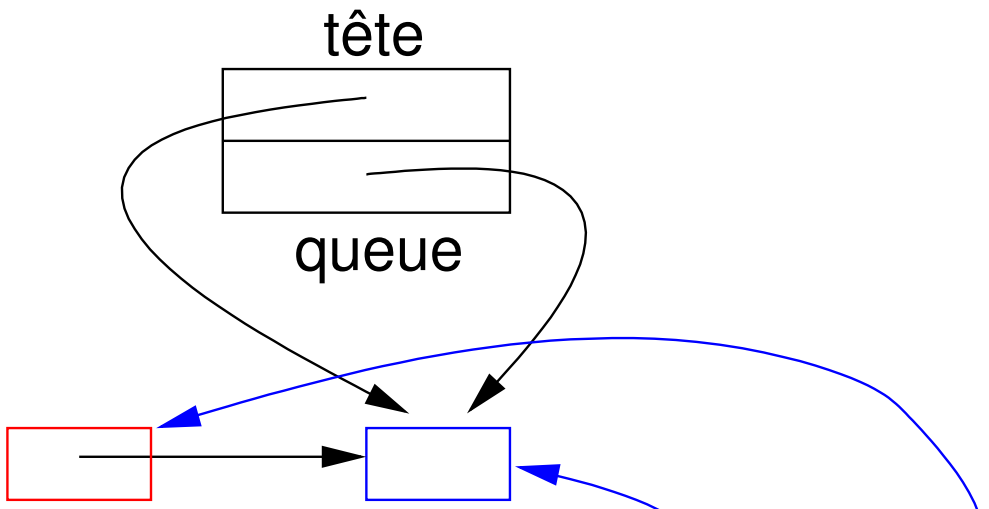
Exemple : `défiler` et `enfiler` concurrents



Exemple : `défiler` et `enfiler` concurrents



Exemple : `défiler` et `enfiler` concurrents



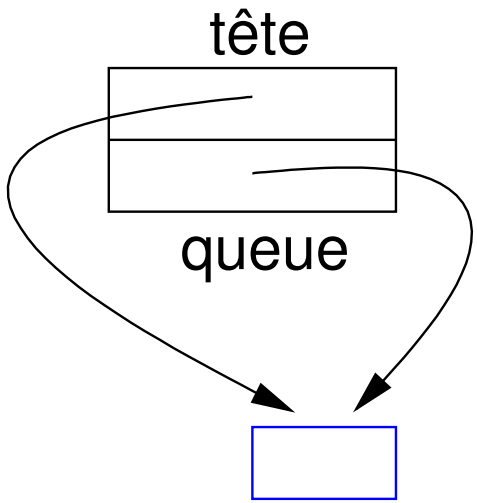
P1 : new

lqueue =

lsuiv = X

CAS(queue,lqueue,new) : échoue (ok)

Exemple : `défiler` et `enfiler` concurrents



Problème A-B-A

- L'algorithme précédent n'est correct qu'en absence de recyclage des cellules libérées par `défiler`
- Problème A-B-A :
  - P_1 lit x et obtient a ;
  - P_2 change x en b et libère a ;
  - P_3 change x en a ;
  - P_1 effectue CAS(x,a,...), qui réussit et lui laisse croire que x n'a pas changé depuis sa lecture.
- Solutions
  - Compteur de générations, incrémenté à chaque modification
    $\langle x, \text{gen } i\rangle \neq \langle x, \text{gen } i+2\rangle$
    Nécessite un CAS2(x,a,gen,i,...)
  - 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 de processus
+ résistant
– structure de données ad-hoc
– implantation fragile, peu réutilisable, pas extensible
– implantation très complexe, à réserver aux experts
– lié à une architecture matérielle
– nécessité de prouver la correction
+ bibliothèques spécialisées
  `java.util.concurrent.ConcurrentLinkedQueue`
  `j.u.concurrent.atomic.AtomicReference.compareAndSet`