Systèmes concurrents

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

26 août 2024

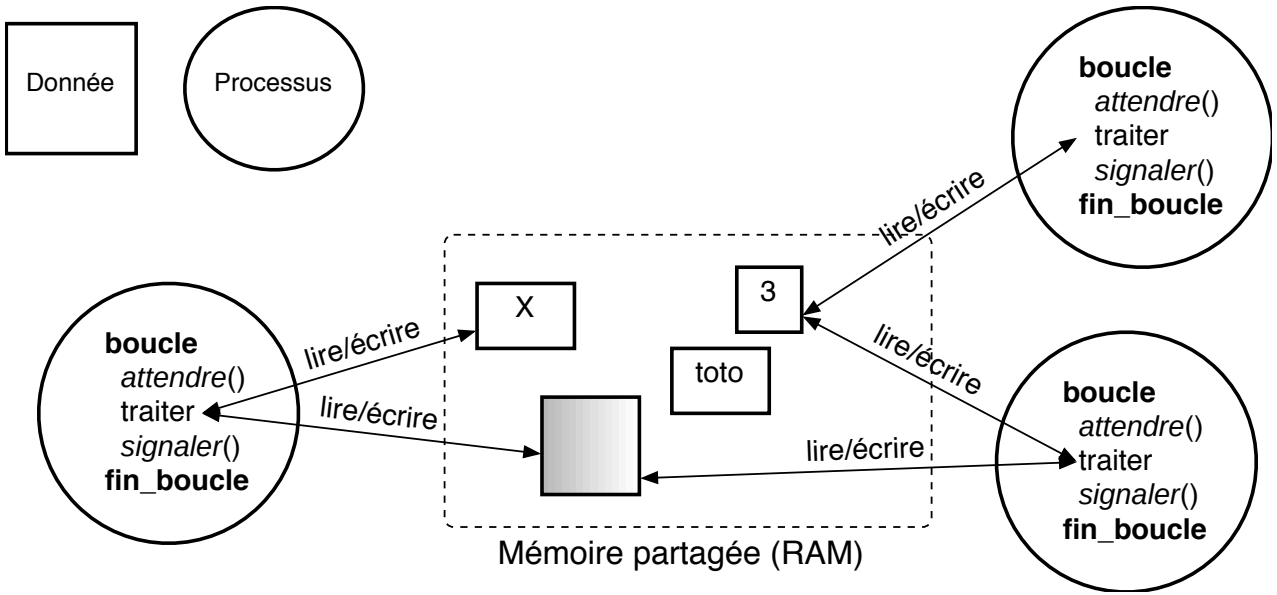
Septième partie

Processus communicants

Contenu de cette partie

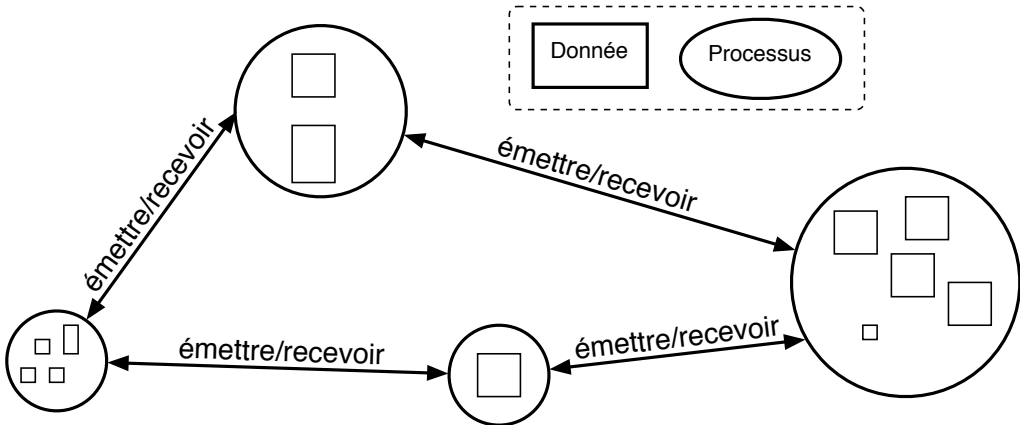
- Modèles de programmation concurrente
- Modèle des processus communicants
- Approche CSP/Go pour la programmation concurrente
  - Goroutine et canaux
  - Communiquer explicitement plutôt que partager implicitement

Modèles d'interaction : mémoire partagée



- Données partagées
- Communication implicite
  - résulte de l'accès et de la manipulation des variables partagées
  - l'identité des activités n'intervient pas dans l'interaction
- Synchronisation explicite (et nécessaire)
- Architectures/modèles cibles
  - multiprocesseurs à mémoire partagée,
  - programmes multiactivités

Modèles d'interaction : processus communicants



- Données encapsulées par les processus
- Communication nécessaire, explicite : échange de messages
  - Programmation et interactions plus lourdes
  - Visibilité des interactions → possibilité de trace/supervision
  - Isolation des données
- Synchronisation implicite : attente de message
- Architectures/modèles cibles
  - systèmes répartis : sites distants, reliés par un réseau
  - acteurs, CSP/Erlang/Go, tâches Ada

Plan

1. Processus communicants
   - Principes
   - Désignation, alternatives
   - Architecture d'une application parallèle
2. Communication synchrone – CSP/CCS/Go
   - Principes
   - Exemple : recherche concurrente
3. Méthodologie
   - Synchronisation pure, approche condition
   - Synchronisation pure, approche par automate
   - Synchronisation avec transfert de données

Processus communicants

Principes

- Communication inter-processus avec des opérations explicites d'envoi / réception de messages
- Synchronisation via ces primitives de communication bloquantes : envoi (bloquant) de messages / réception bloquante de messages

Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π-calcul / Erlang / Go

Les principes détaillés des échanges et leur utilisation pour développer des applications sont vus dans le module « intergiciels ». On ne s'intéresse ici qu'à la synchronisation.

Quelle synchronisation ?

Réception

Réception bloquante : attendre un message

Émission

- Émission non bloquante ou asynchrone
- Émission bloquante ou synchrone : bloque jusqu'à la réception du message = rendez-vous élémentaire entre l'activité émettrice et l'activité destinataire
- Rendez-vous étendu : bloquant jusqu'à réception + réaction + réponse ≈ appel de procédure

- Émission asynchrone ⇒ buffers (messages émis non reçus)
- Synchrone ⇒ 1 case suffit

Désignation du destinataire et de l'émetteur

Nommage direct

Désignation de l'activité émettrice/destinataire

```
SEND message TO processName
RECV message FROM processName
```

Nommage indirect

Désignation d'une boîte à lettres ou d'un canal de communication

```
SEND message TO channel
RECV message FROM channel
```

Multiplicité

$1-1$

Désignation de l'activité : 1 émetteur / 1 récepteur désignés

$n-1$

Canal réservé en lecture (consommation) : envoi par n'importe quelle activité ; réception par une seule, propriétaire du canal

$n-m$

Canal avec envoi par n'importe qui, réception par n'importe qui :

- pas de duplication : un seul destinataire consomme le message
- ou duplication à tous les destinataires (diffusion)

En mode synchrone, la diffusion est complexe et coûteuse à mettre en œuvre (nécessite une synchronisation globale entre tous les récepteurs)

Alternative

Alternative en émission ou en réception = choix parmi un ensemble de communications possibles :

```
RECV msg FROM channel1 OR channel2
(SEND msg1 TO pid1) OR (SEND msg2 TO pid2)
(RECV msg1 FROM channel1) OR (SEND msg2 TO channel2)
```

- Si aucun choix n'est faisable ⇒ attendre
- Si un seul des choix est faisable ⇒ le faire
- Si plusieurs choix sont faisables ⇒ sélection non-déterministe (arbitraire)

Divers

Émission asynchrone ⇒ risque de buffers pleins

- perte de messages ?
- ou l'émission devient bloquante si plein ?

Émission non bloquante → émission bloquante

introduire un acquittement

```
   (SEND m TO ch; RECV _ FROM ack)
 ‖ (RECV m FROM ch; SEND _ TO ack)
```

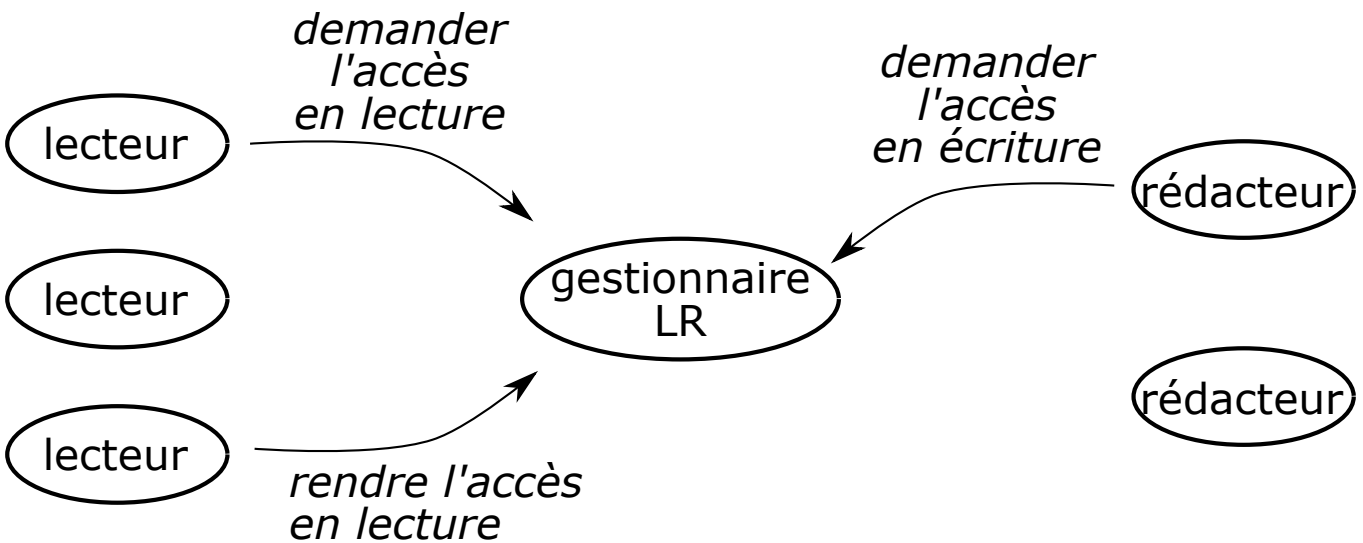
Émission bloquante → émission non bloquante

introduire une boîte intermédiaire qui accepte immédiatement tout message et le stocke dans une file.

```
   (SEND m TO ch1)
 ‖ boucle (RECV m FROM ch1; insérer m dans file)
 ‖ boucle (si file non vide alors extraire et SEND TO ch2)
 ‖ (RECV FROM ch2)
```

Architecture

La résolution des problèmes de synchronisation classiques (producteurs/consommateurs...) ne se fait plus en synchronisant directement les activités via des données partagées, mais indirectement via une activité de synchronisation.



Activité gestionnaire d'un objet partagé

Interactions avec l'objet partagé

Pour chaque opération faisable sur l'objet :

- émettre un message de requête vers le gestionnaire
- attendre le message de réponse de gestionnaire

Schéma de fonctionnement du gestionnaire

- L'arbitre exécute une boucle infinie contenant une alternative
- Cette alternative possède une branche par opération fournie
- Chaque branche est gardée par la condition d'acceptation de l'opération (suivie de l'attente du message correspondant)

Note : en communication synchrone, on peut se passer du message de réponse s'il n'y a pas de contenu à fournir.

Intérêt

+ découplage entre les activités clientes : l'interface partagée est celle de l'activité de synchronisation
+ réalisation centralisée et répartie
+ transfert explicite d'information : traçage
+ pas de données partagées $\Rightarrow$ pas de protection nécessaire
+ contrôle fin des interactions
+ schéma naturel côté client : question/réponse = appel de fonction
− multiples recopies (mais optimisations possibles)
− parallélisation du service : au cas par cas

Plan

1. Processus communicants
   - Principes
   - Désignation, alternatives
   - Architecture d'une application parallèle
2. Communication synchrone – CSP/CCS/Go
   - Principes
   - Exemple : recherche concurrente
3. Méthodologie
   - Synchronisation pure, approche condition
   - Synchronisation pure, approche par automate
   - Synchronisation avec transfert de données

Go language

Principes de conception

- Syntaxe légère inspirée du C
- Typage statique fort avec inférence
- Interfaces avec extension et polymorphisme (typage structurel / duck typing à la Smalltalk)
- Ramasse-miettes

Concepts pour la concurrence

- Descendant de CSP (Hoare 1978), cousin d'Erlang
- Goroutine $\sim$ activité/thread
  - une fonction s'exécutant indépendant (avec sa pile)
  - très léger (plusieurs milliers sans problème)
  - gérée par le noyau Go qui alloue les ressources processeurs
- Canaux pour la communication et la synchronisation

Go – canaux

Canaux

- Création : `make(chan type)` ou `make(chan type, 10)` (synchrone / asynchrone avec capacité)
- Envoi d'une valeur sur le canal `chan` : `chan <- valeur`
- Réception d'une valeur depuis `chan` : `<- chan`
- Canal transmissible en paramètre ou dans un canal : `chan chan int` est un canal qui transporte des canaux (transportant des entiers)

Go – canaux

Alternative en réception et émission

```
select {
 case v1 := <- chan1:
        fmt.Printf("received %v from chan1\n", v1)
 case v2 := <- chan2:
        fmt.Printf("received %v from chan2\n", v2)
 case chan3 <- 42:
        fmt.Printf("sent %v to chan3\n", 42)
 default:
        fmt.Printf("no one ready to communicate\n")
}
```

Exemple élémentaire

```
func boring(msg string, c chan string) {
  for i := 0; ; i++ {
    c <- fmt.Sprintf("%s %d", msg, i)
    time.Sleep(time.Duration(rand.Intn(4)) * time.Second)
  }
}
```

```
func main() {
  c := make(chan string)
  go boring("boring!", c)
  for i := 0; i < 5; i++ {
    fmt.Printf("You say: %q\n", <- c)
  }
  fmt.Println("You're boring; I'm leaving.")
}
```

Moteur de recherche

Objectif : agrégation de la recherche dans plusieurs bases

func Web(query **string**) Result
func Image(query **string**) Result
func Video(query **string**) Result

Moteur séquentiel

```
func Google(query string) ( results []Result ) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```

exemple tiré de `https://talks.golang.org/2012/concurrency.slide`

Recherche concurrente

Moteur concurrent

```
func Google(query string) ( results  []Result ) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <- c
        results = append(results, result )
    }
    return
}
```

Le temps sans interruption

Crée un canal sur lequel un message sera envoyé après la durée spécifiée.

time.After

```
func After(d time.Duration) <-chan bool {
    // Returns a receive-only channel
    // A message will be sent on it after the duration
    c := make(chan bool)
    go func() {
        time.Sleep(d)
        c <- true
    }()
  return c
}
```

Recherche concurrente en temps borné

Moteur concurrent avec timeout

```
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

Recherche répliquée

Utiliser plusieurs serveurs répliqués et garder la réponse du premier qui répond.

Recherche en parallèle

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Recherche répliquée

Moteur concurrent répliqué avec timeout

```
c := make(chan Result)
go func() { c <- First(query, Web1, Web2, Web3) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

Bilan

- Création ultra-légère de goroutine : penser concurrent
- Pas besoin de variables partagées
  $\Rightarrow$ Pas de verrous d'exclusion mutuelle
- Pas besoin de variable condition pour synchroniser explicitement
- Pas besoin de callback ou d'interruption

> Don't communicate by sharing memory, share memory by communicating.

(la bibliothèque Go contient *aussi* les objets usuels de synchronisation pour travailler en mémoire partagée : verrous, sémaphores, moniteur...)

Plan

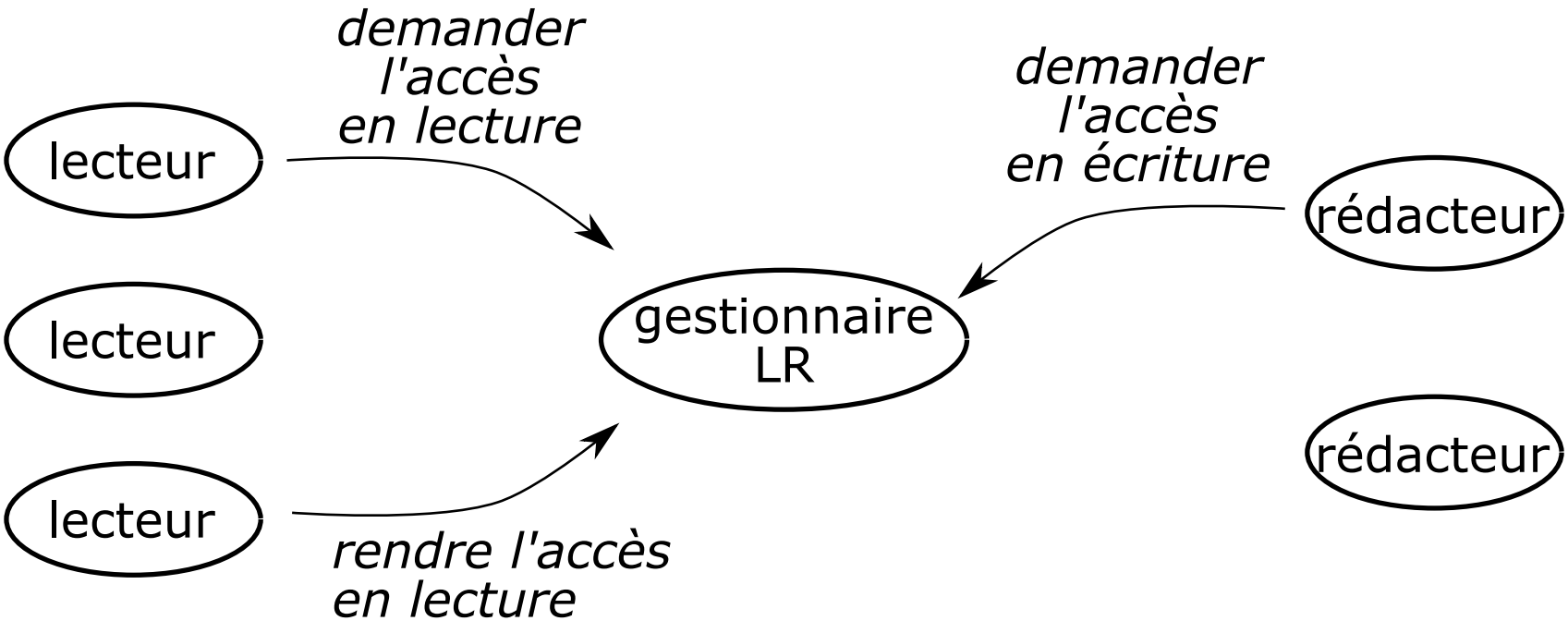
1. Processus communicants
   - Principes
   - Désignation, alternatives
   - Architecture d'une application parallèle
2. Communication synchrone – CSP/CCS/Go
   - Principes
   - Exemple : recherche concurrente
3. Méthodologie
   - Synchronisation pure, approche condition
   - Synchronisation pure, approche par automate
   - Synchronisation avec transfert de données

Synchronisation pure, approche condition

Pour un problème de synchronisation pure (pas d'échange de données) :

1. Identifier l'interface = les requêtes recevables → un canal par requête
2. Identifier les conditions d'acceptation pour chaque canal
3. Activité serveur qui boucle, lisant un message parmi ceux dont la condition d'acceptation est vraie (et bloquant s'il n'y a aucun tel message).

Exemple : lecteurs/rédacteurs (1/3)



- Un canal pour chaque type de requête : DL, TL, DE, TE
- Émission bloquante ⇒ accepter un message (une requête) uniquement si l'état l'autorise

Exemple : lecteurs/rédacteurs (2/3)

Utilisateur

```
func Utilisateur () {
  nothing := struct{}{}
  for {
    DL <- nothing; // demander lecture
     ...
    TL <- nothing; // terminer  lecture
     ...
    DE <- nothing; // demander écriture
     ...
    TE <- nothing; // terminer  é criture
  }
}
```

Exemple : lecteurs/rédacteurs (3/3)

Goroutine de synchronisation

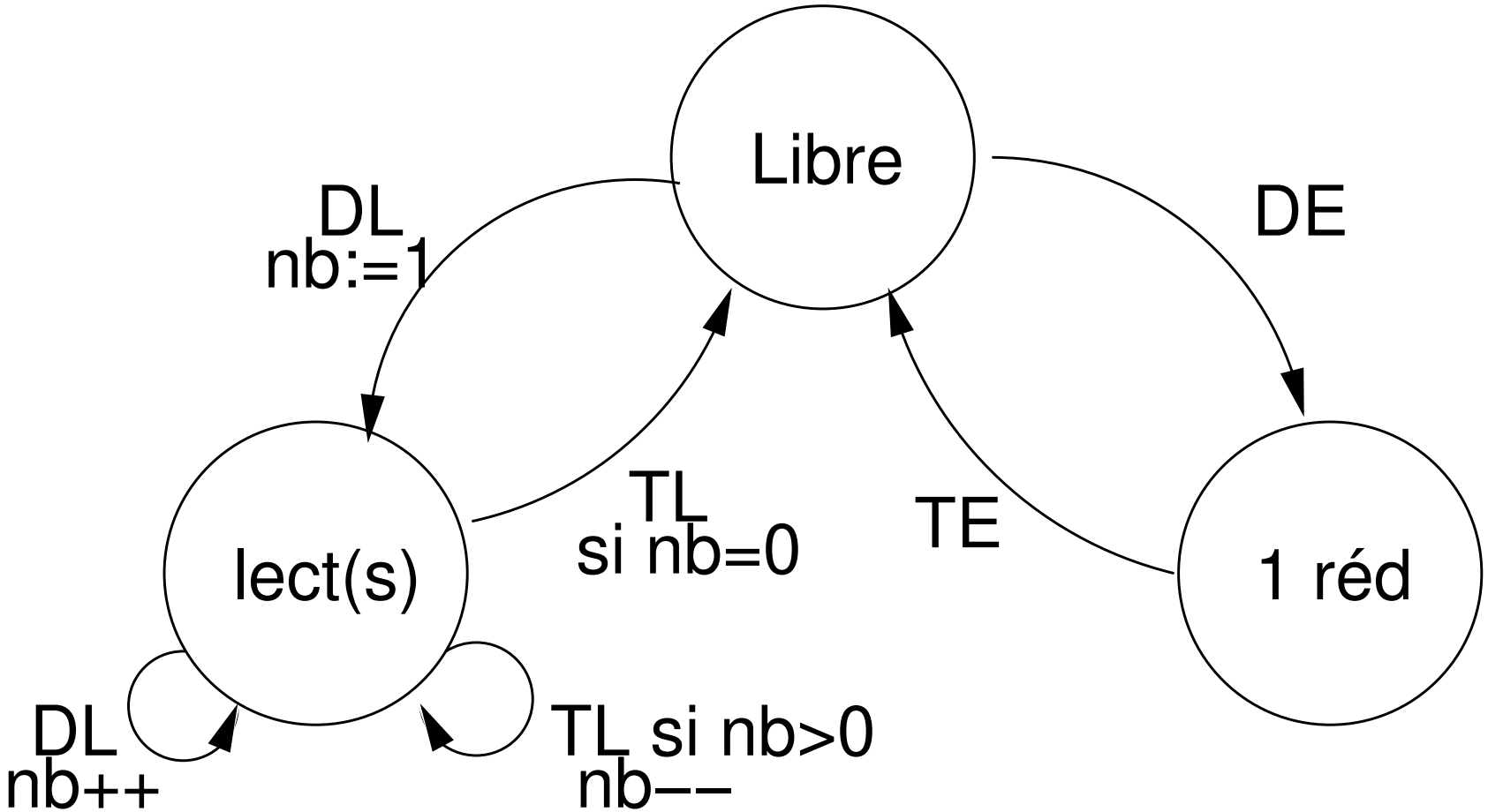
```
func when(b bool, c chan struct{}) chan struct{} {
    if b { return c } else { return nil }
}

func SynchroLR() {
  nblec := 0;
  ecr := false;
  for {
     select {
        case <- when(nblec == 0 && !ecr, DE):
              ecr := true;
        case <- when(!ecr, DL):
              nblec++;
        case <- TE:
              ecr := false;
        case <- TL:
              nblec--;
     }
}}
```

Synchronisation pure, approche par automate

Pour un problème de synchronisation pure (pas d'échange de données) :

1. Identifier l'interface = les requêtes recevables $\rightarrow$ un canal par requête
2. Construire un automate fini à états.
   Chaque état se distingue par les canaux acceptables en lecture.
3. L'activité serveur boucle, et selon l'état courant détermine quels messages peuvent être pris.

Exemple : lecteurs/rédacteurs (1/2)



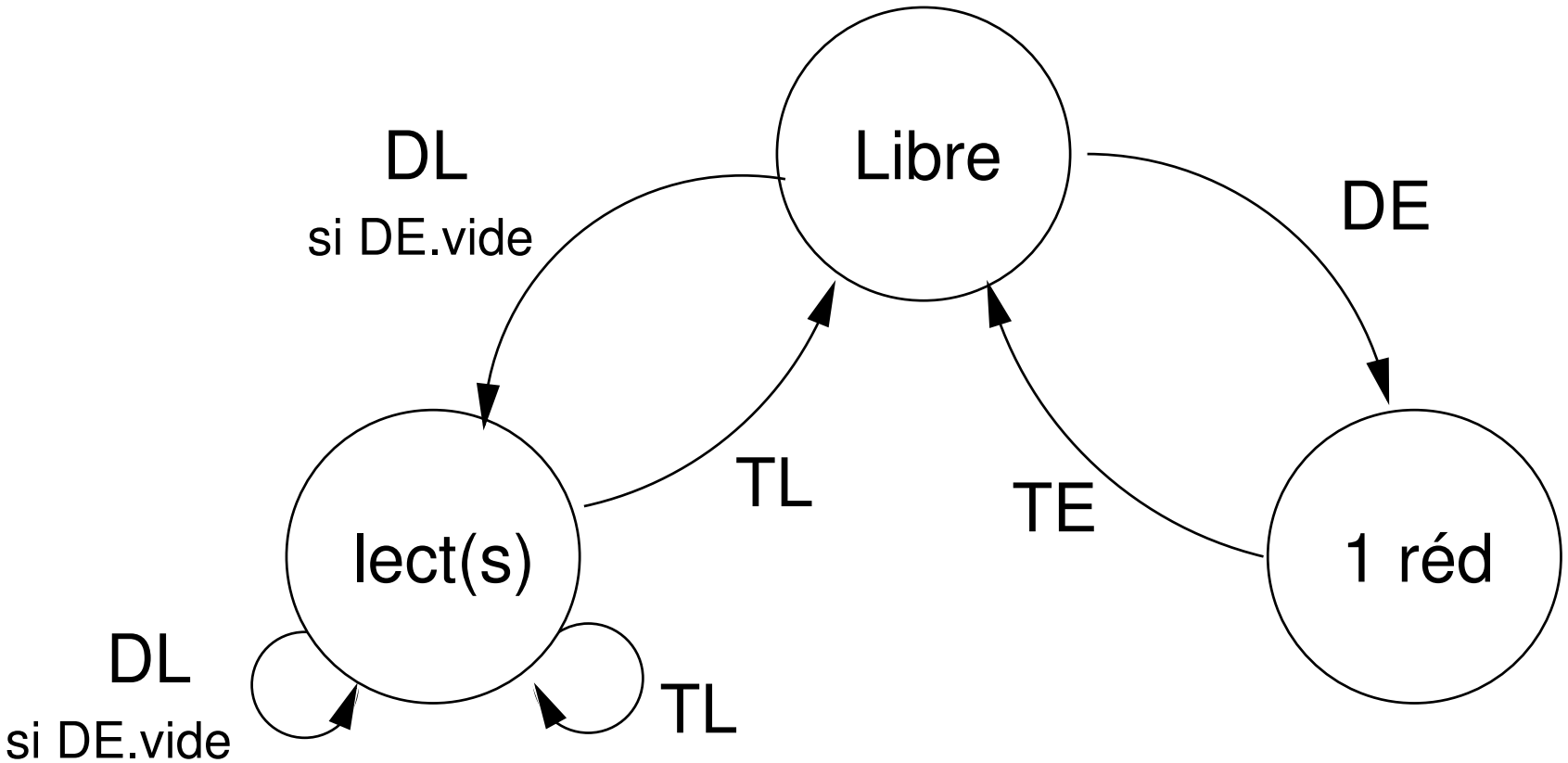
Automate étendu avec une variable d'état `nb` qui contrôle les transitions.

Exemple : Lecteurs/rédacteurs (2/2)

Goroutine de synchronisation

```
const ( lecteur = iota, redacteur , libre )
func SynchroLR() {
   nblec := 0;
   etat := libre ;
   for {
        if etat == libre {
             select {
                  case <- DE : etat = redacteur
                  case <- DL : etat = lecteur; nblec++ }
        } else if etat == lecteur {
             select {
                  case <- DL : etat = lecteur; nblec++
                  case <- TL : nblec--; if nblec == 0 { etat = libre } }
        } else { // etat == redacteur
             select {
                  case <- DE : etat = libre }
        }
    }
}
```

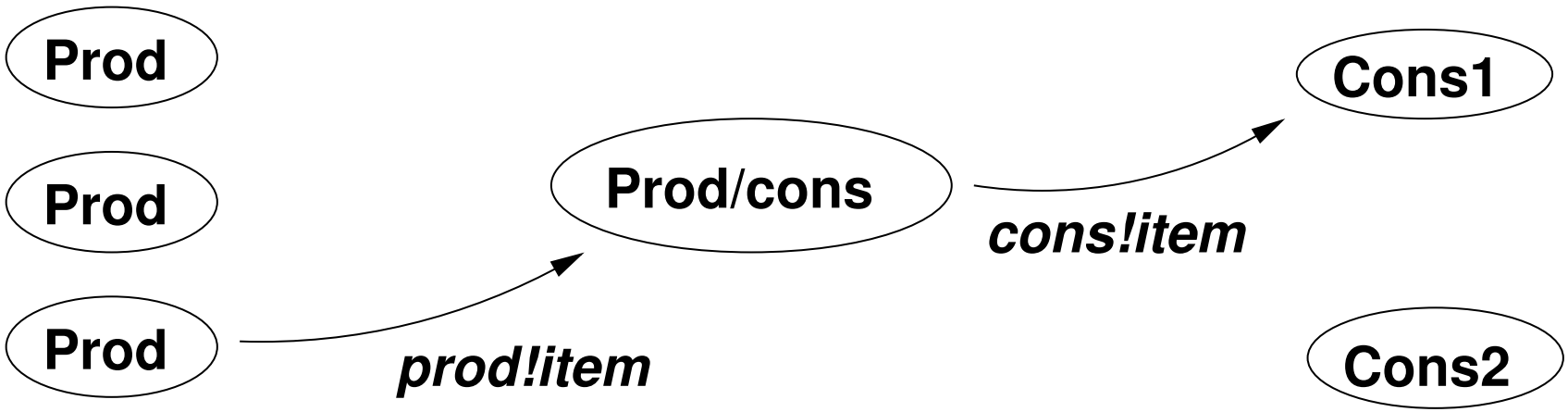
Lecteurs/rédacteurs, priorité rédacteur



`canal.Vide` est vrai s'il n'y a pas de message en attente. Toutes les variantes de CSP ne disposent pas de cette opération.

Transfert de données – architecture simpliste

Identifier l'interface en distinguant le sens de circulation de l'information : sens client → serveur ou serveur → client.



- un canal pour les demandes de dépôt (sens producteur → tampon)
- un canal pour les transmissions des valeurs déposées (sens tampon → consommateur)

(exercice futile : `make(chan T, N)` est déjà un tampon borné = un prod/cons de taille N)

Exemple : producteurs/consommateurs (1/2)

Producteur

```
func producteur(prod chan int) {
    for {
        ...
        item := ...
        prod <- item
    }
}
```

Consommateur

```
func consommateur(cons chan int) {
    for {
        ...
        item := <- cons
        // utiliser item
    }
}
```

Exemple : producteurs/consommateurs (2/2)

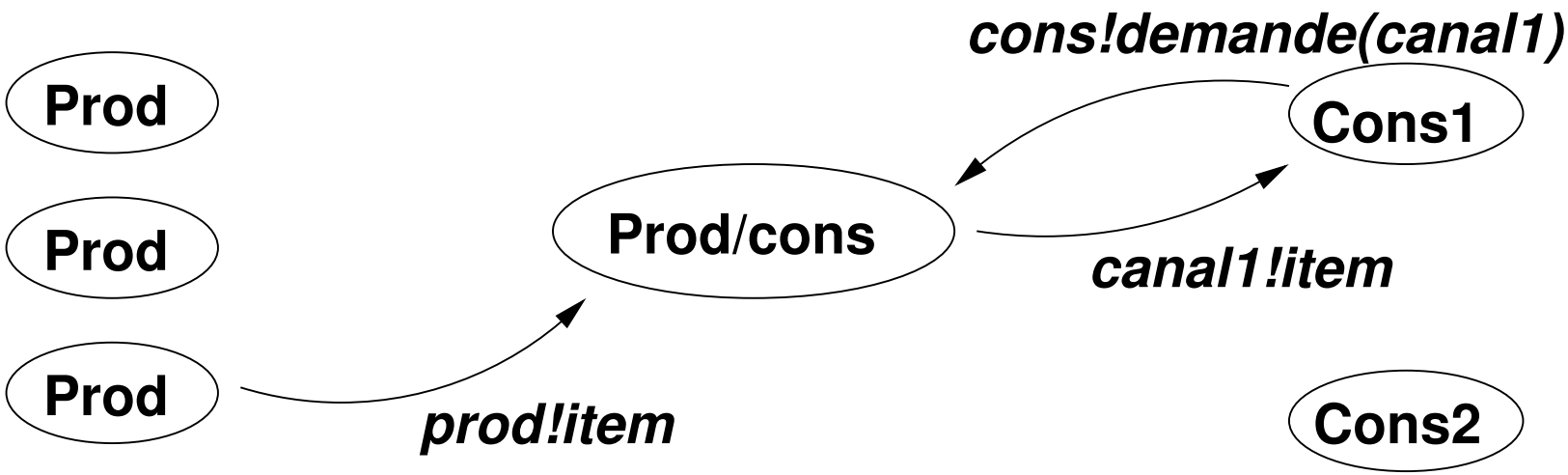
Activité de synchronisation

```
func prodcons(prod chan int, cons chan int) {
 nbocc := 0
 queue := make([]int, 0)};
 for {
     if nbocc == 0 {
         m := <- prod; nbocc++; queue = append(queue, m)
     } else if nbocc == N {
         cons <- queue[0]; nbocc--; queue = queue[1:]
     } else {
         select {
           case m := <- prod:
                 nbocc++; queue = append(queue, m)
           case cons <- queue[0]:
                 nbocc--; queue = queue[1:]
         }
     }
 }
}
```

Transfert de données – Architecture évoluée

1. Un canal par type de requête
2. La requête contient, outre les paramètres de celle-ci, un canal privé au client.
3. Le serveur répond sur le canal privé quand la requête peut être satisfaite.

Exemple : Producteurs/consommateurs (1/4)



- Un canal pour les demandes de dépôt, pas de réponse attendue
- Un canal pour les demandes de retrait
- Un canal par activité demandant le retrait (pour la réponse à celle-ci)

(exercice futile : `make(chan T, N)` est déjà un tampon borné = un prod/cons de taille N)

Producteurs/consommateurs (2/4)

Programme principal

```
func main() {
    prod := make(chan int)      // un canal portant des entiers
    cons := make(chan chan int) // un canal portant des canaux
    go prodcons(prod, cons)
    for i := 1; i < 10; i++ {
        go producteur(prod)
    }
    for i := 1; i < 5; i++ {
        go consommateur(cons)
    }
    time.Sleep (20*time.Second)
    fmt.Println ("DONE.")
}
```

Producteurs/consommateurs (3/4)

Producteur

```
func producteur(prod chan int) {
  for {
    ...
    item := ...
    prod <- item
  }}
```

Consommateur

```
func consommateur(cons chan chan int) {
  moi := make(chan int)
  for {
    ...
    cons <- moi
    item := <- moi
    // utiliser item
  }}
```

Variables « canal », passables en paramètre et en message.

Producteurs/consommateurs (4/4)

Goroutine de synchronisation

```
func prodcons(prod chan int, cons chan chan int) {
  nbocc := 0;
  queue := make([]int, 0)
  for {
    if nbocc == 0 {
      m := <- prod; nbocc++; queue = append(queue, m)
    } else if nbocc == N {
      c := <- cons; c <- queue[0]; nbocc--; queue = queue[1:]
    } else {
      select {
        case m := <- prod: nbocc++; queue = append(queue, m)
        case c := <- cons:
              c <- queue[0]; nbocc--; queue = queue[1:]
      }
    }
  }
}
```

Bilan processus communicants

+ Pas de partage implicite de la mémoire ($\rightarrow$ isolation)
+ Transfert explicite d'information ($\rightarrow$ traçage)
+ Réalisation centralisée et répartie
$\sim$ Contrôle fin des interactions
$\sim$ Méthodologie
– Performance (copies)
– Quelques schémas classiques, faire preuve d'invention ($\rightarrow$ attention aux doigts)

Les langages modernes proposent la communication et synchronisation à la fois par mémoire partagée et par canaux.

Huitième partie

Synchronisation non bloquante

Plan

4 Objectifs et principes

5 Exemples
- Splitter & renommage
- Pile chaînée
- Liste chaînée

6 Conclusion

Limitation des verrous

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

- Interblocage : ensemble d'activités se bloquant mutuellement
- Inversion de priorité : une activité de faible priorité bloque une activité 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'une activité
- Tuer un activité ?
- 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)

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

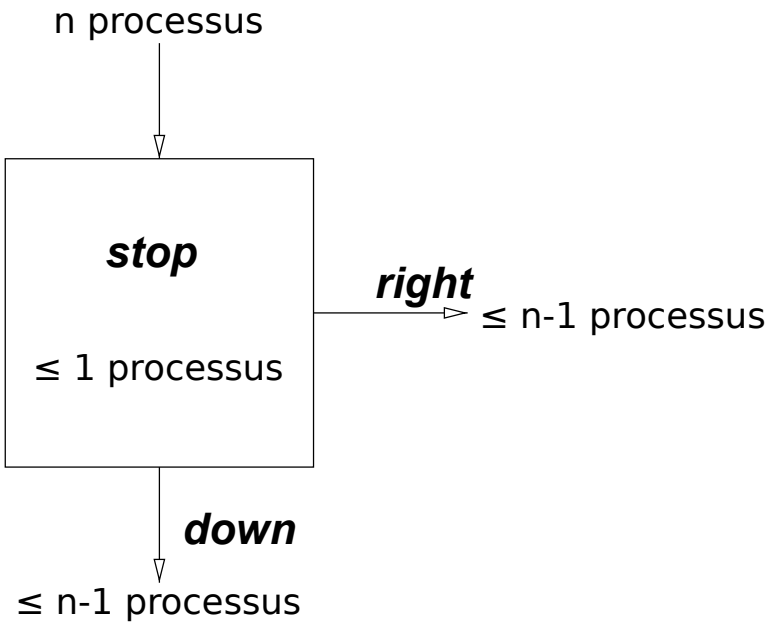
4 Objectifs et principes

5 Exemples
- Splitter & renommage
- Pile chaînée
- Liste chaînée

6 Conclusion

Splitter

Moir, Anderson 1995



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

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

au plus $n - 1$ 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

Problème

- 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

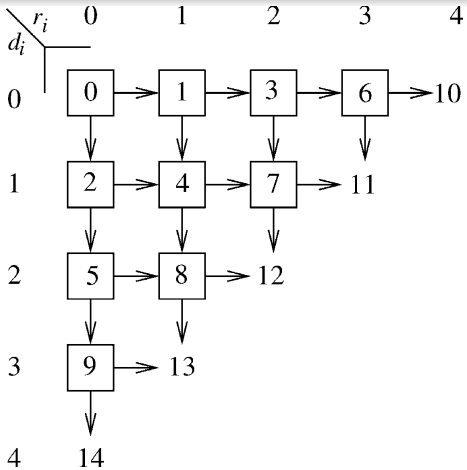
- Distributeur de numéro accédé en exclusion mutuelle
- $M = n$
- Complexité temporelle : O(1) pour un numéro, O(n) pour tous
- Une activité lente ralentit les autres

```
lock mutex; int numéro = 0;  // partagé
int obtenir_nom() {
    mutex.lock();
    int res = numéro; numéro = numéro + 1;
    mutex.unlock();
    return res;
}
```

Grille de splitters

Solution non bloquante

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



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 $n = 1$,
- 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

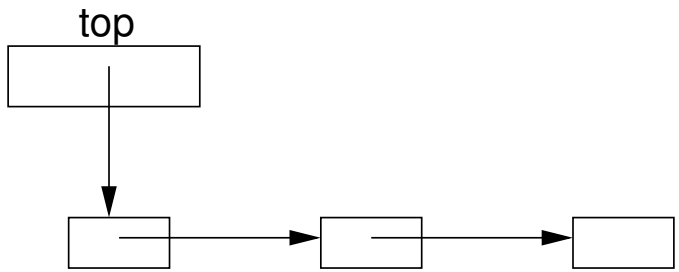
Objet avec opérations `push` et `pop`

```
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;
  }
}
```

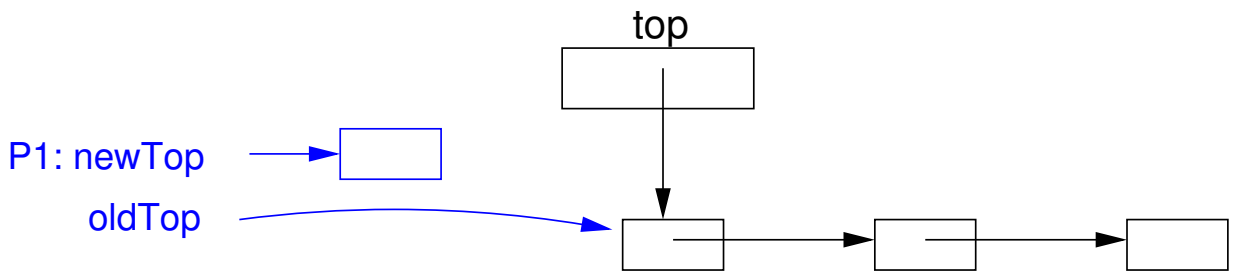
```
  public T pop() {
    Node<T> oldTop = top;
    if (oldTop == null)
      return null;
    top = oldTop.next;
    return oldTop.item;
  }
```

Non résistant à une utilisation concurrente par plusieurs activités

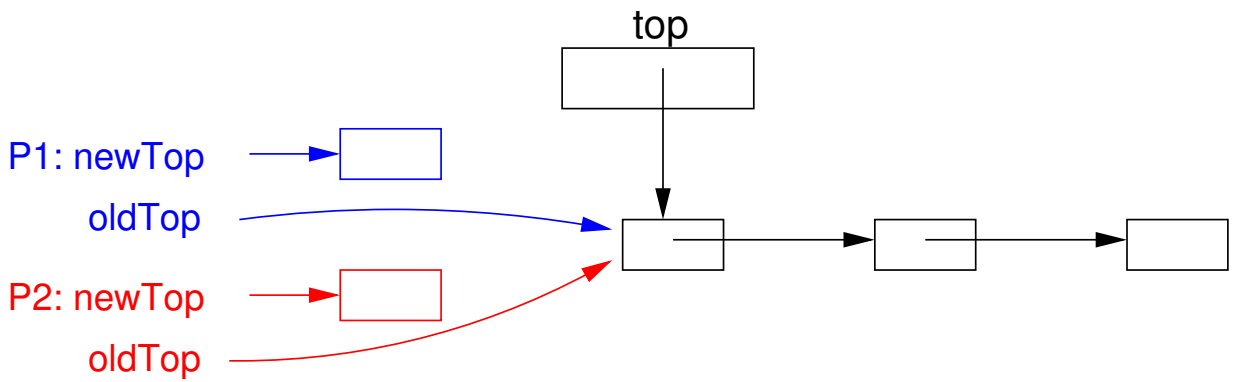
Pile chaînée basique : conflit push/push



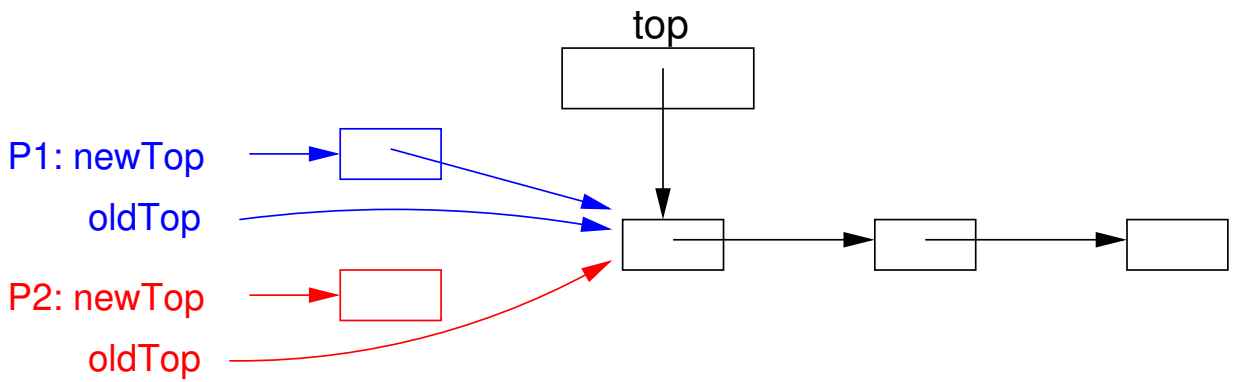
Pile chaînée basique : conflit push/push



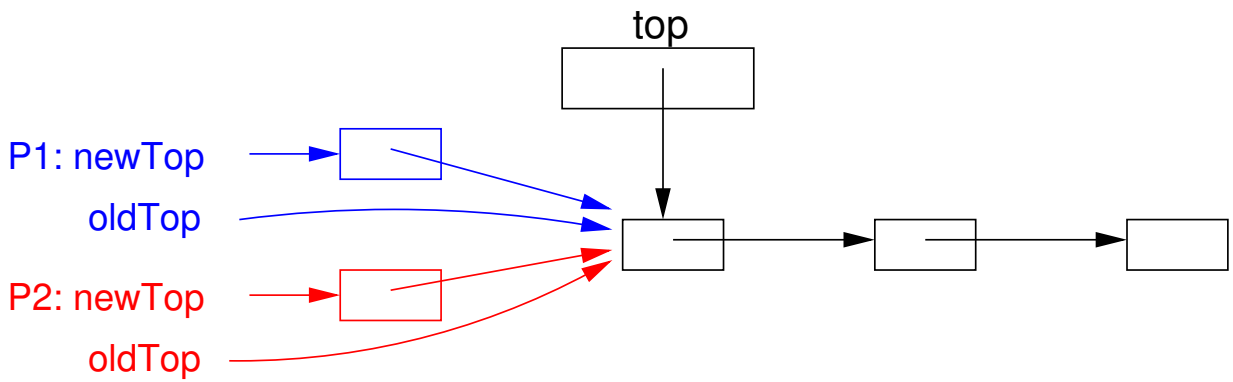
Pile chaînée basique : conflit push/push



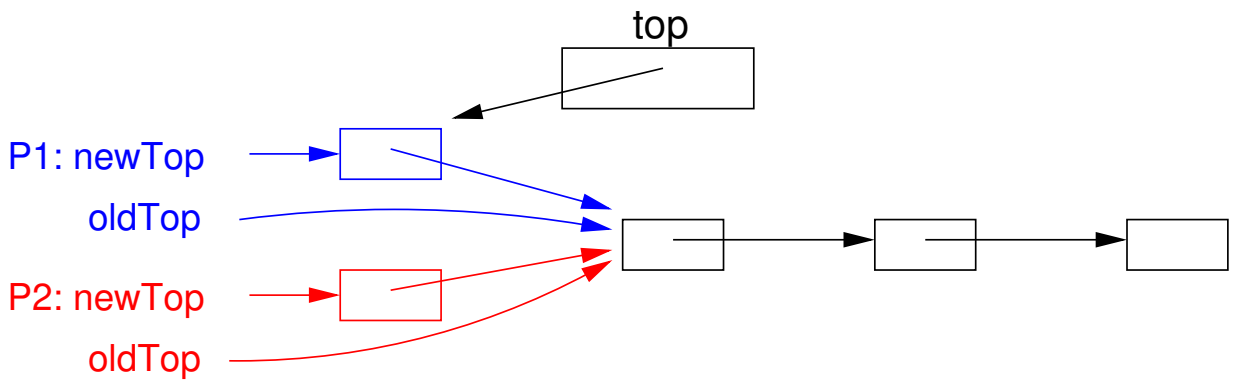
Pile chaînée basique : conflit push/push



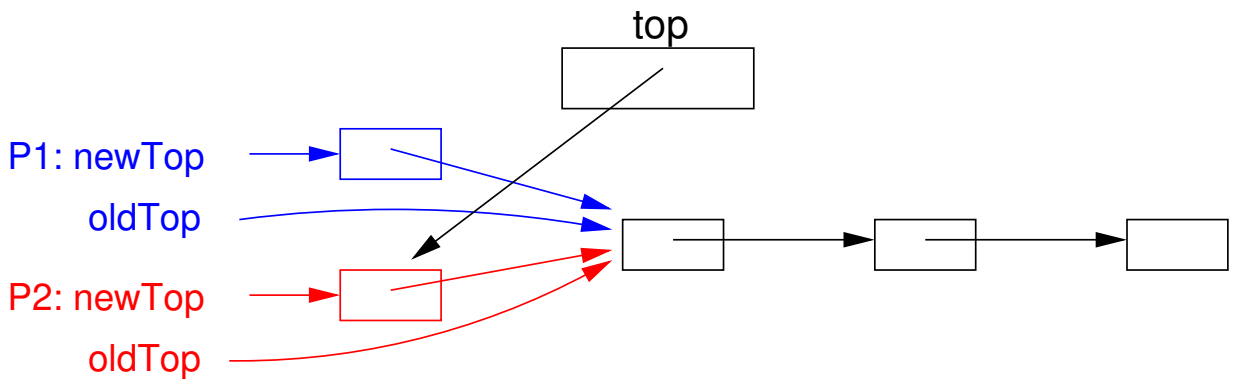
Pile chaînée basique : conflit push/push



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

Boucle uniquement si conflit, ce n'est pas de l'attente active.

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
$\Rightarrow$ 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 :

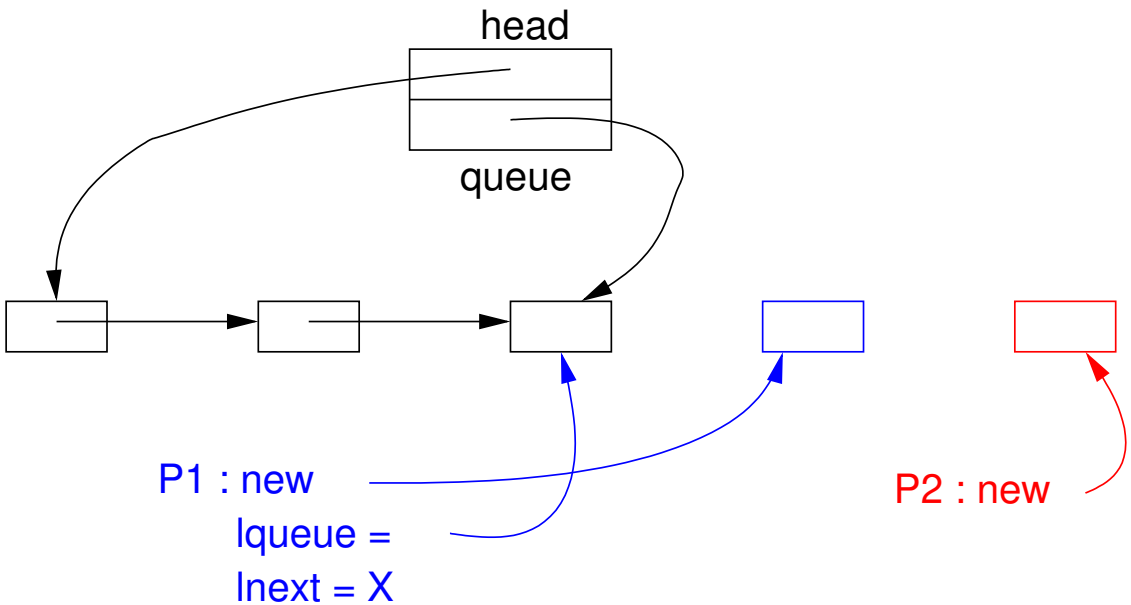
```
boolean CAS(*add, old, new) {
    atomically {
        if (*add == old ) { *add = new; return true; }
        else { return false ; }
    }
}
```

Enfiler non bloquant

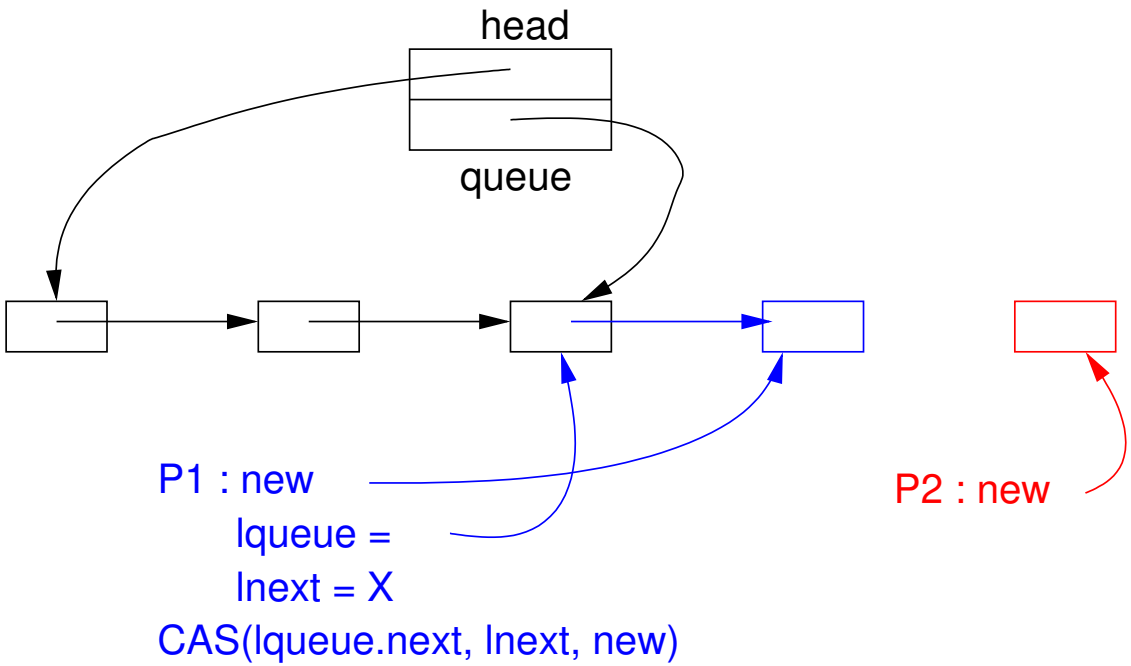
enqueue non bloquant

```
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
```

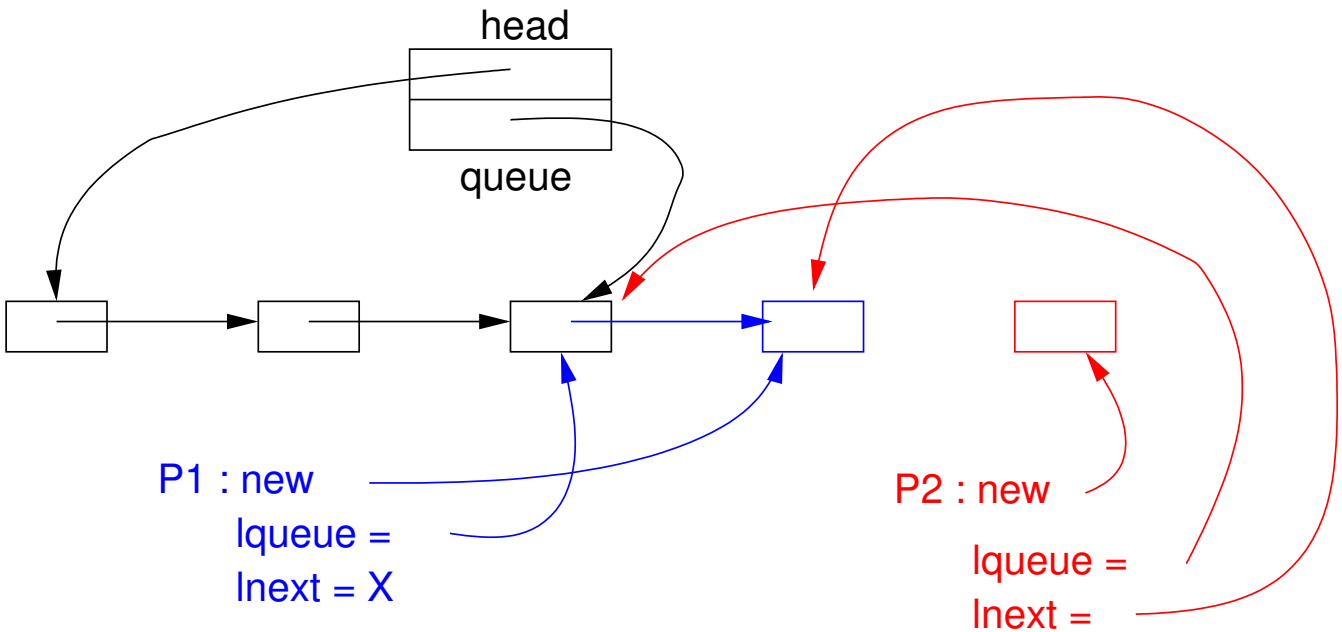
Exemple : deux enqueue concurrents



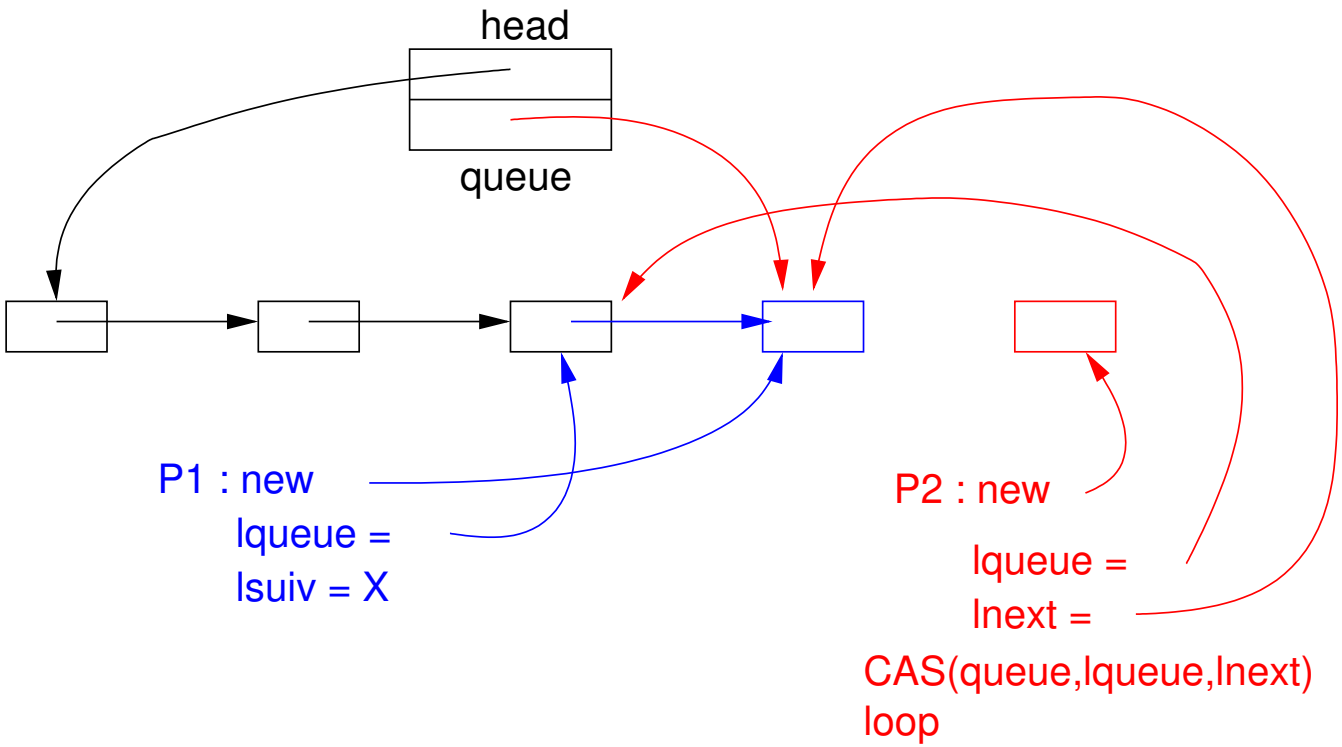
Exemple : deux enqueue concurrents



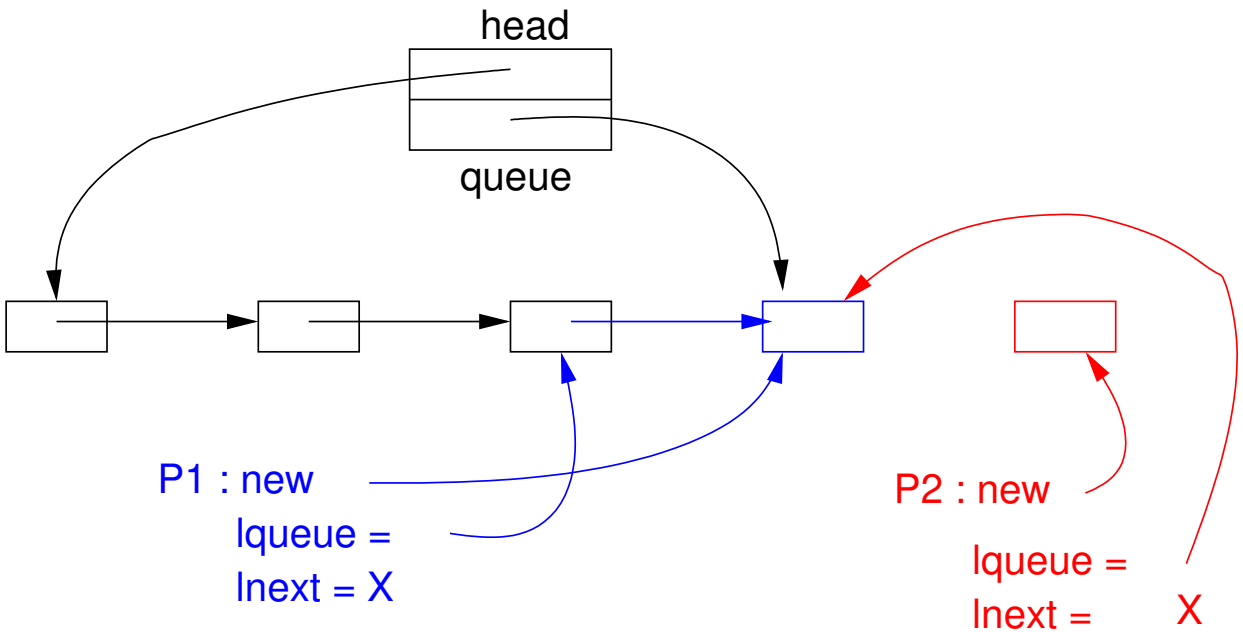
Exemple : deux enqueue concurrents



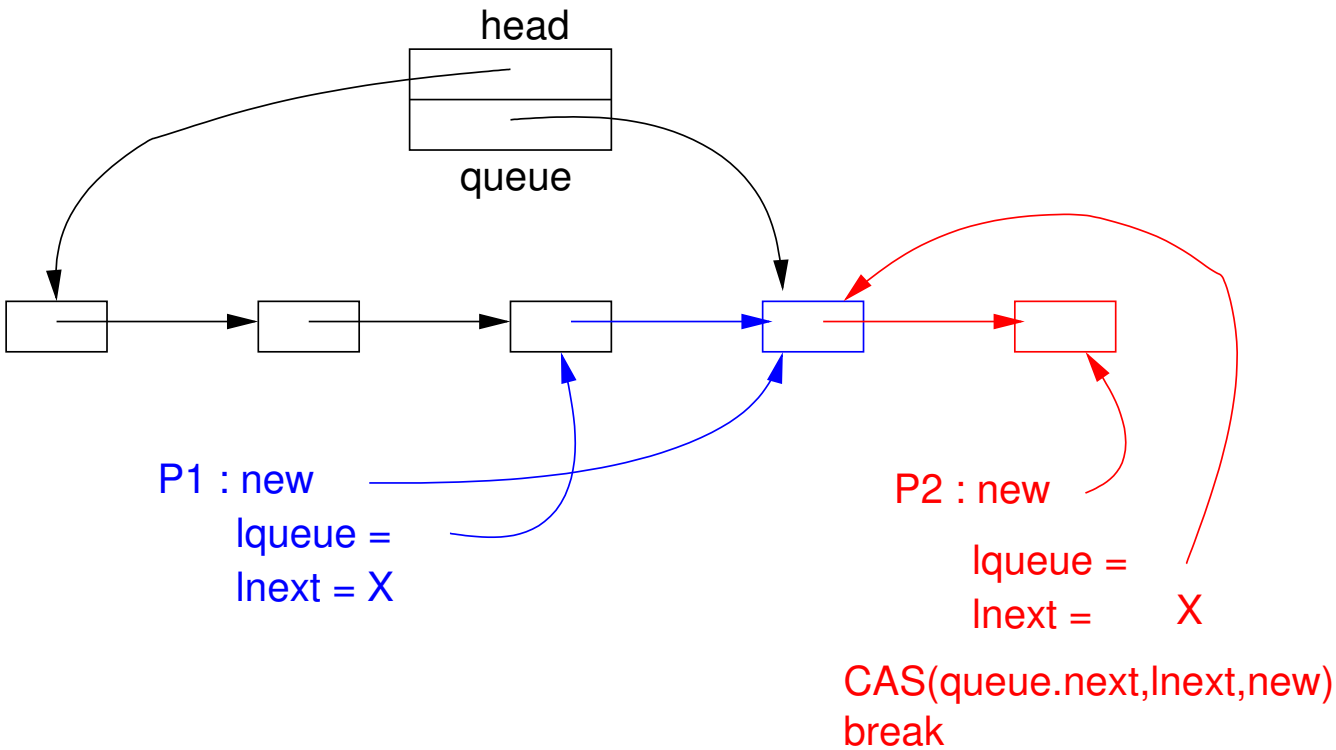
Exemple : deux enqueue concurrents



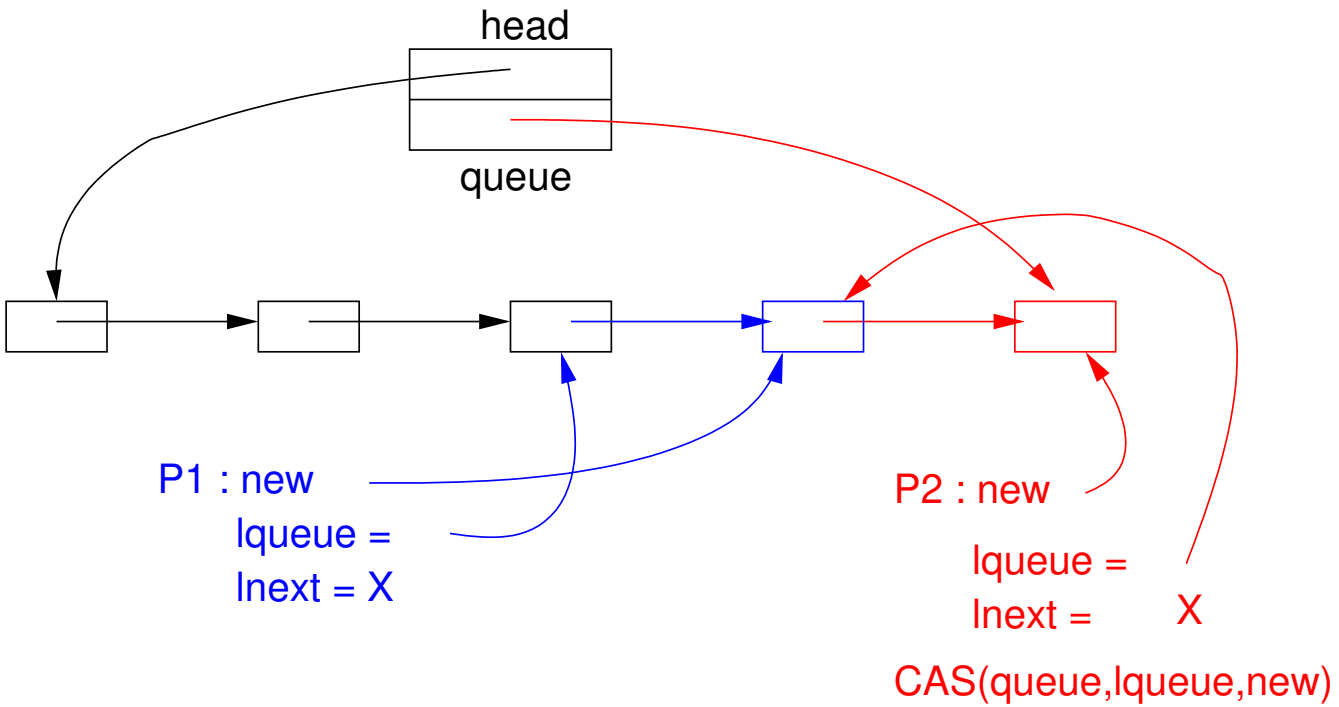
Exemple : deux enqueue concurrents



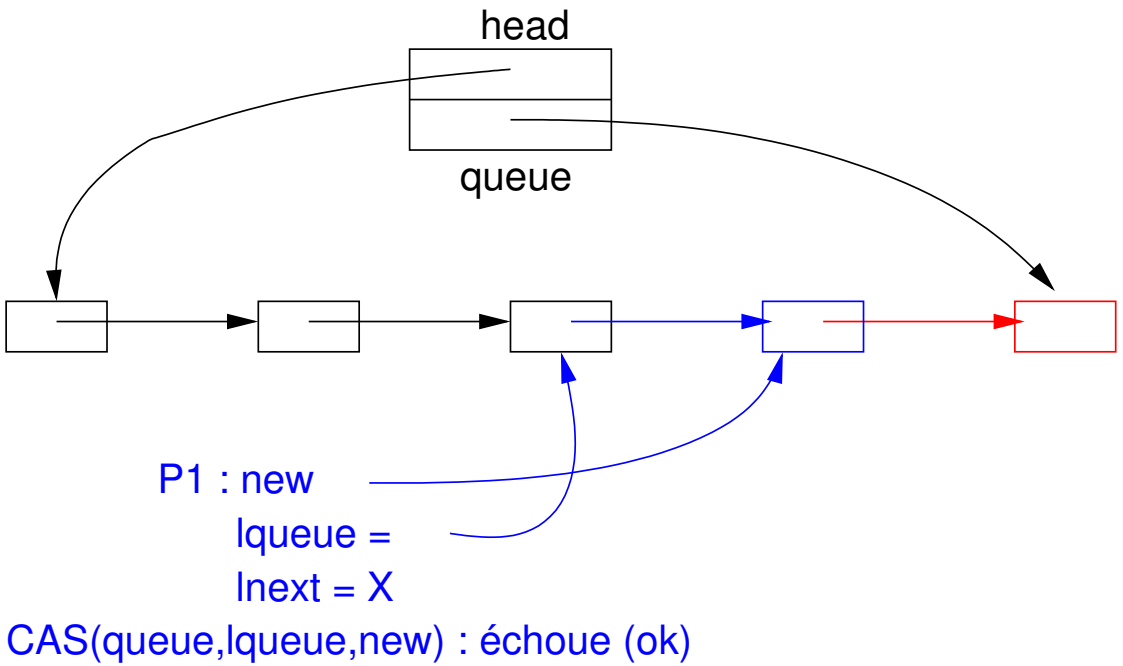
Exemple : deux enqueue concurrents



Exemple : deux enqueue concurrents



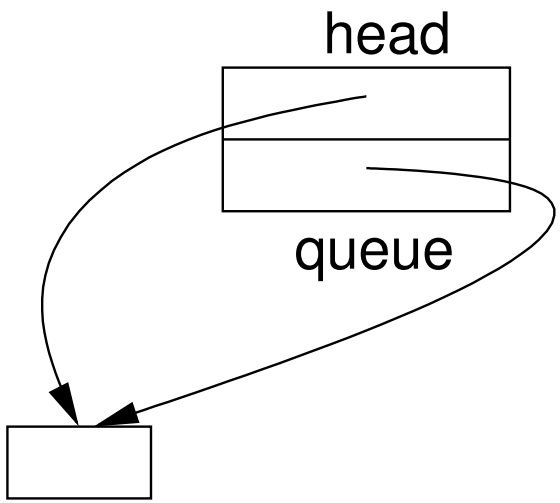
Exemple : deux enqueue concurrents



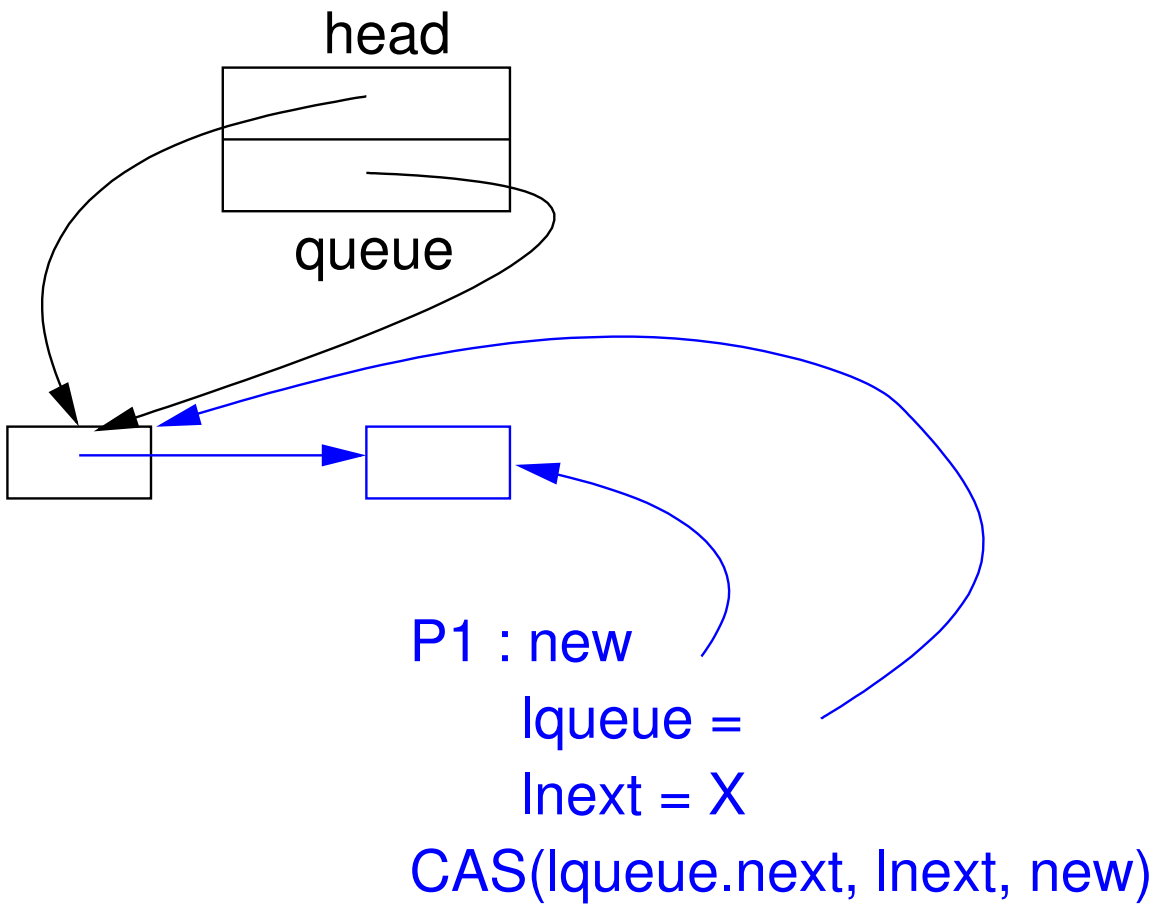
dequeue non bloquant

```
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 ;
```

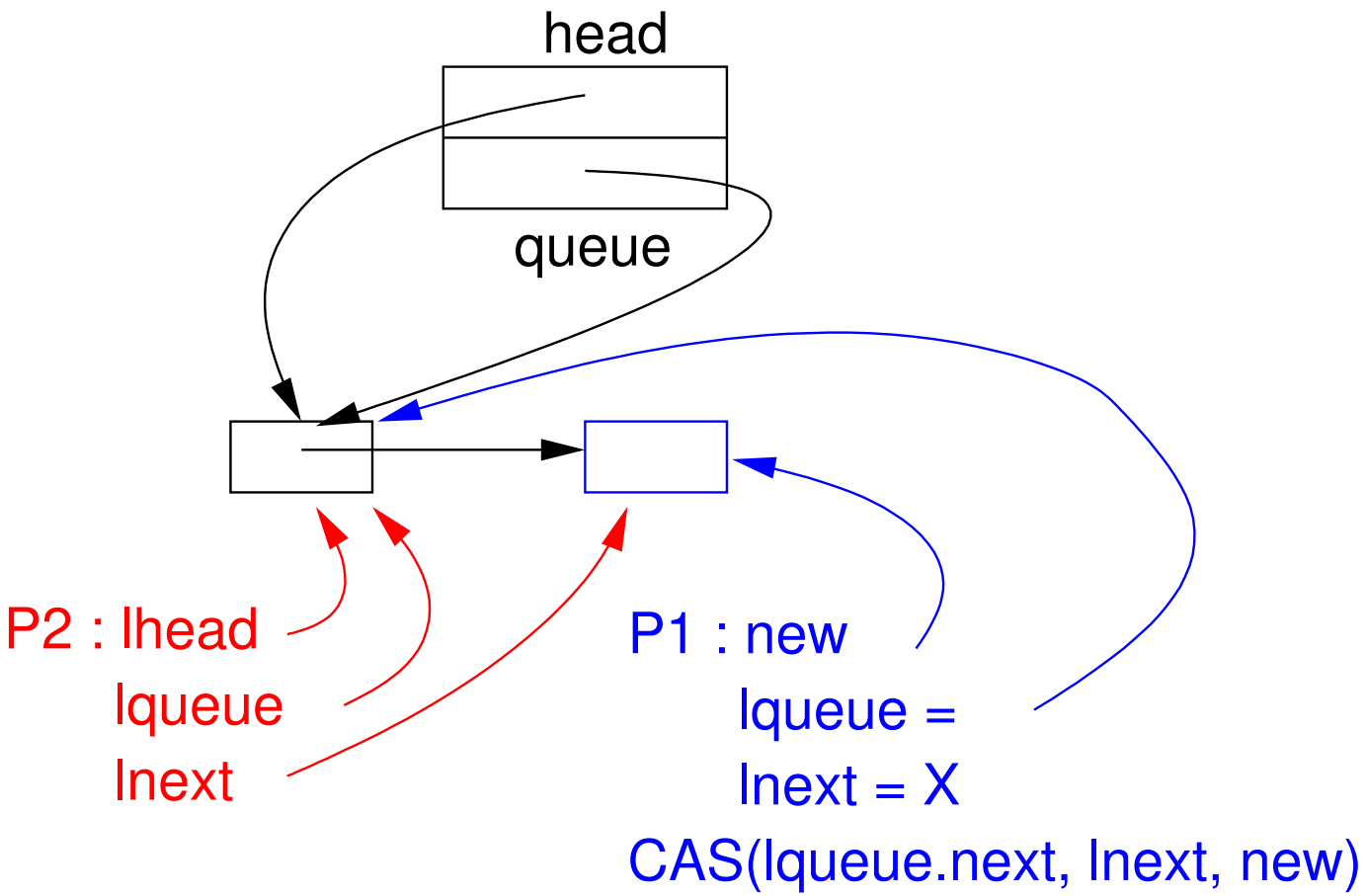
Exemple : dequeue et enqueue concurrents



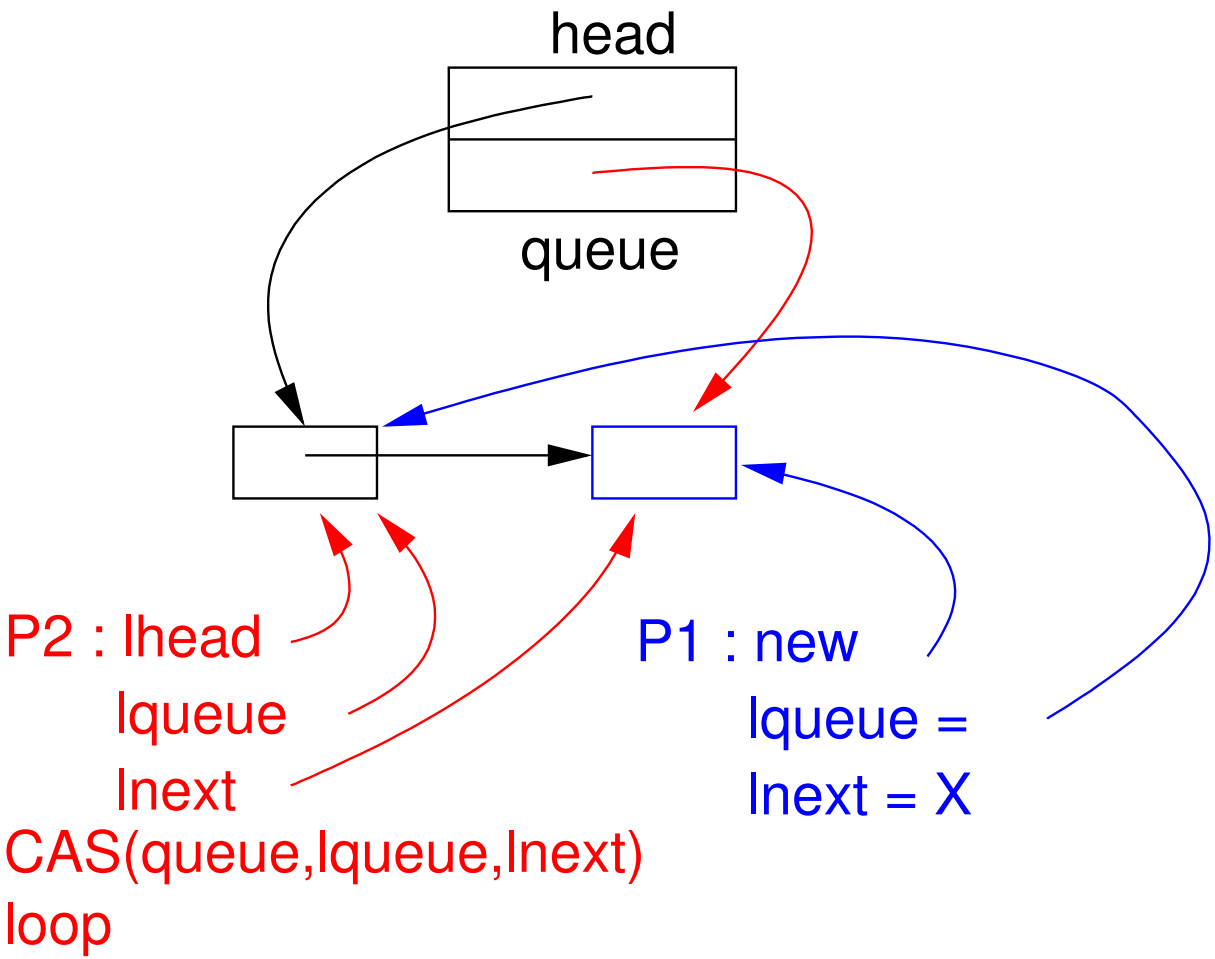
Exemple : dequeue et enqueue concurrents



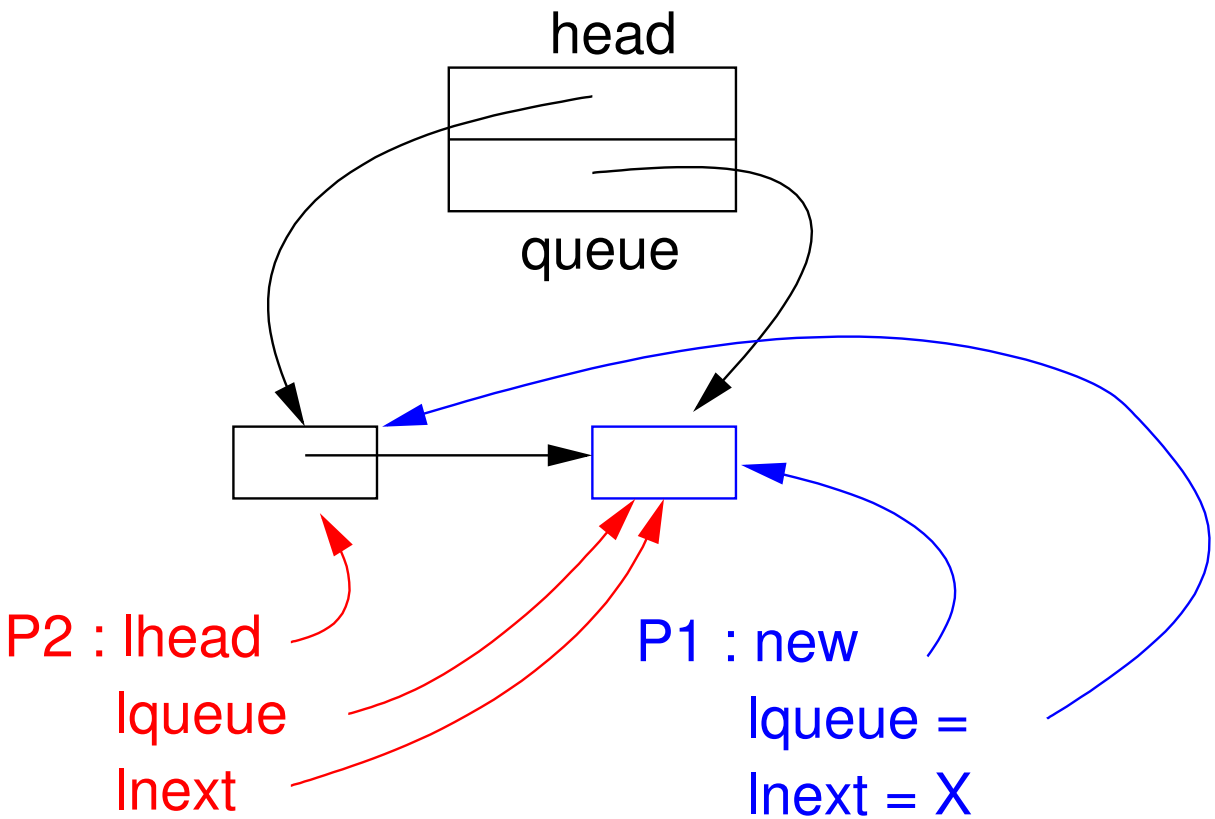
Exemple : dequeue et enqueue concurrents



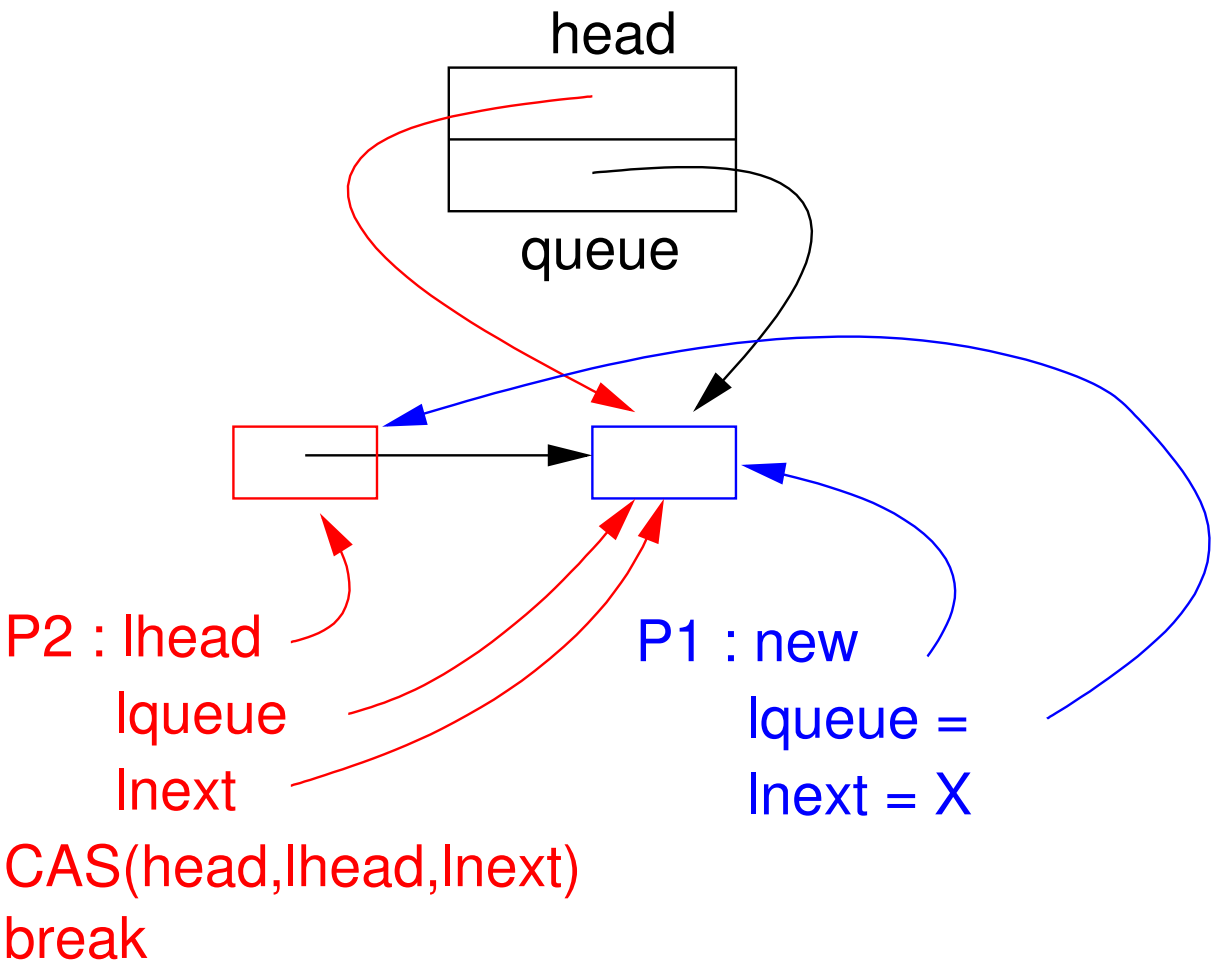
Exemple : dequeue et enqueue concurrents



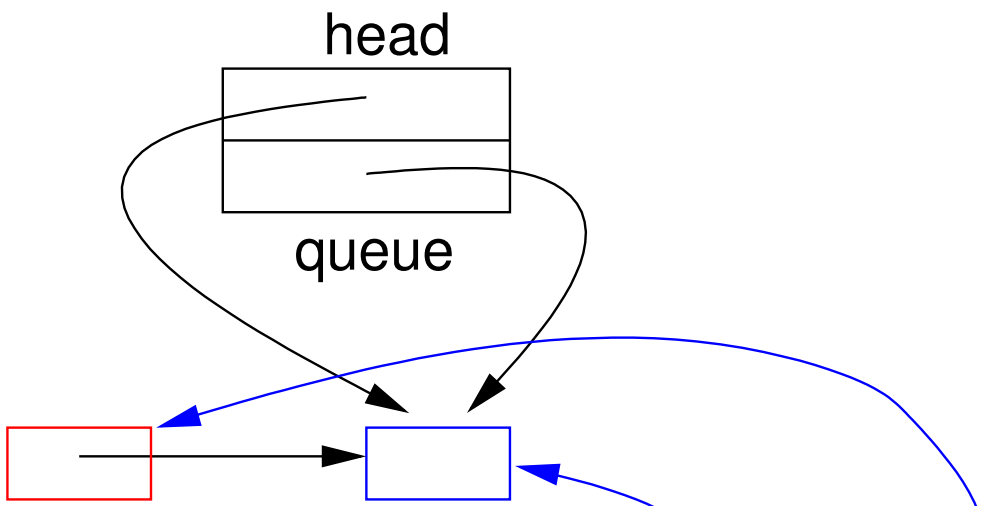
Exemple : dequeue et enqueue concurrents



Exemple : dequeue et enqueue concurrents



Exemple : dequeue et enqueue concurrents



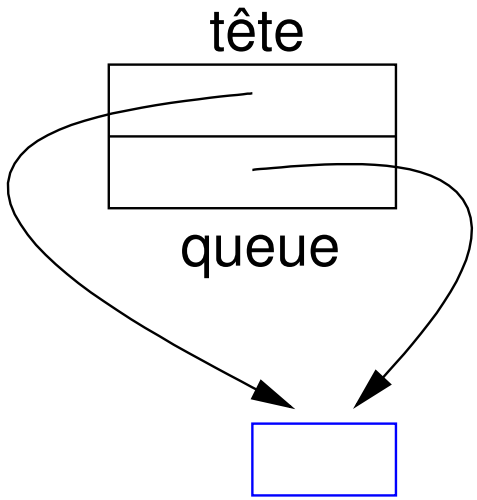
P1 : new

lqueue =

lnext = X

CAS(queue,lqueue,new) : échoue (ok)

Exemple : dequeue et enqueue concurrents



Plan

4 Objectifs et principes

5 Exemples
- Splitter & renommage
- Pile chaînée
- Liste chaînée

6 Conclusion

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`