Théorie et pratique de la concurrence

CM11 Structures de données partagés

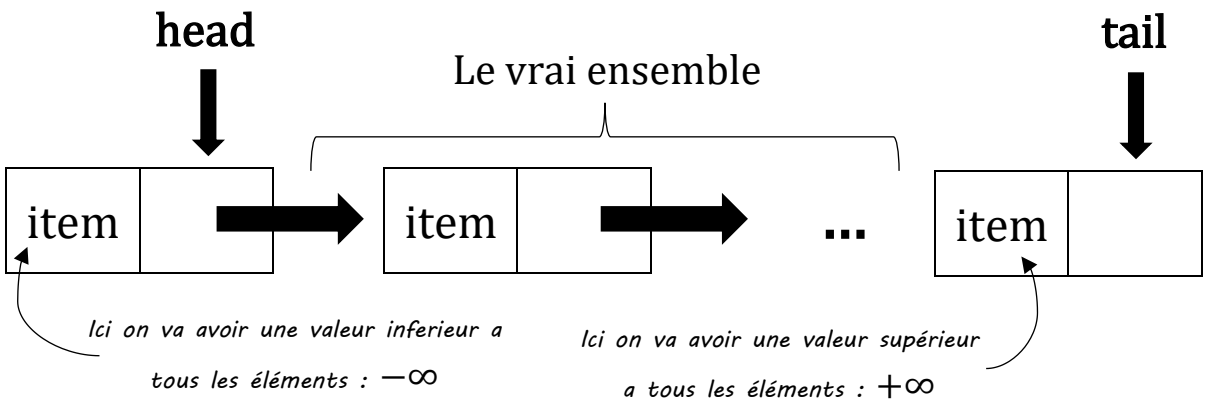
Lundi 06·12·2021

Comment programmer, en utilisant des lock, des structures de données partages ?

Ensembles d'entiers partagés

- **add(key)**
- **remove(key)** : Retirer l'élément **key**.
- **contains(key)** : un boolean, si l'élément existe dans l'ensemble ou pas.

- *Faut régler la synchronisation : l'accès à la structure de donnés·*
- *On ca choisir une représentation par liste chainées·*
- ***Une liste ordonnée** : ca simplifie la recherche d'un élément, notamment pour l'enlever de l'ensemble·*
- *Et on va avoir 2 pointers particuliers :* **head, tail**·



- *Supposons qu'on a une solution complètement séquentielle. C.-à-d. : chaque opération s'exécute seul, il y a pas aucun concurrence.*
- *Si je prend ce cas qu'on connait bien, on peut faire une version concurrente en ajouton un lock sur la structure. Un processus qui viens va se battre pour le lock, cela est appelé* **Coarse-gain locking** *(« le gain grossier »)*
- *On va supposer comme structure des nœuds (Node) :*

```
type Node {
        int key ;
        Node next ;
...
```

Coarse-gain locking (« le gain grossier »)

On va écrire une première fonction ajouter, on lui donne un certain **key**, *qui est un entier. Il y a toujours au moins 2 Node : début et fin. Chaque élément existe dans la liste au plus une fois (l'occurrence d'un élément est 0 ou 1).*

boolean remove(int key)

```
Node pred, curr ; // Prédécesseur, séquenceur
lock(l) ; // On prend le lock
try
    pred := head ;
    curr := pred.next ;
    while (curr.next < key) // key : le paramètre
        pred := curr ; // Le prédécesseur devient le courant
        curr := curr.next ; // Le courant devient le successeur
    if (key == curr.key) // On a trouvé l'élément
        pred.next := curr.next ;
        return true ;
    else
        return false ; // L'élément existe pas
finally
    unlock(l) ; // On lâche le lock
```

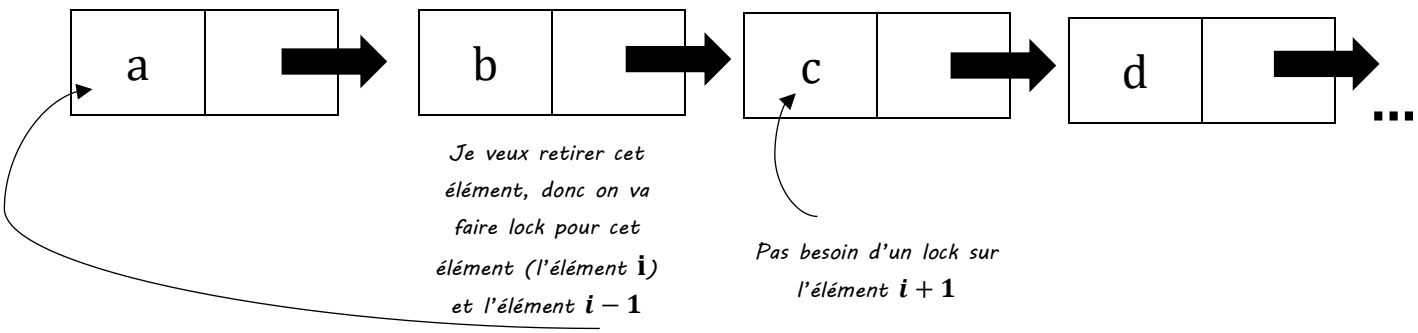
boolean add(int key)

```
Node pred, curr ; // Prédécesseur, séquenceur
lock(l) ; // On prend le lock
try
    pred := head ;
    curr := pred.next ;
    while (curr.next < key) // key : le paramètre
        pred := curr ; // Le prédécesseur devient le courant
        curr := curr.next ; // Le courant devient le successeur
    if (key == curr.key) // L'élément est déjà dans l'ensemble
        return false ; // L'élément est déjà dans l'ensemble, on veut pas avoir de duplications
    else
        Node node = new Node(key) ; // Création d'un nouveau nœud
        node.next := curr :
        pred.next := node :
        return true ; // Tous c'est bien passer
finally
    unlock(l) ; // On lâche le lock
```

Inconvénient

L'ajout d'un élément a un emplacement X ne doit pas empêcher l'ajout d'un élément dans une autre zone.

Fin-gain locking (plus locale)



- *Je lâche* **a** *que lorsque je prends* **c** *… à chaque fois j'ai donc lock sur 2 éléments.*
- *Quand je commence a traverser la liste c'est possible que il y a un processus plus loin que nous, on est tous les deux sur la liste, je l'attend pas…*
- *Si un processus arrive, et un autre processus fait une opération devant lui, il doit attendre, il peut pas le dépasser.*

Fin-gain locking

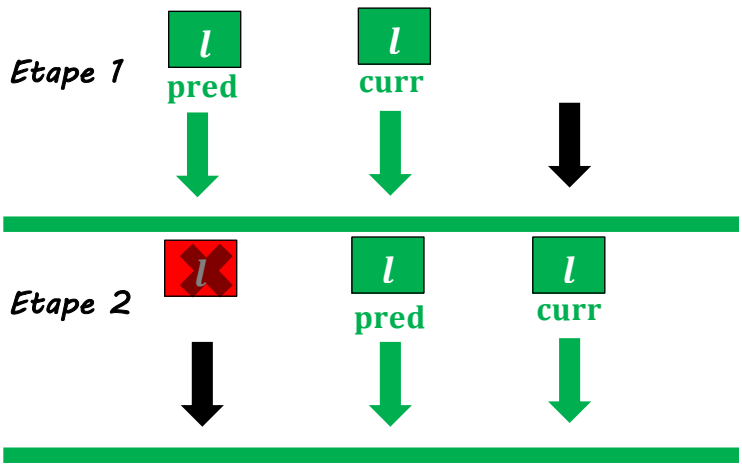
boolean add(int key)

```
lock(head) ; // Verrou sur la tête de la liste
Node pred = head ;
try
    curr := pred.next ;
    lock(curr) ; // A ce stade lock sur l'élément head et le suivant
    try
        while (curr.next < key) // On fait la recherche, pendant la recherche on va déplacer nos lock.
            unlock(pred) ;
            pred := curr ; // Le prédécesseur devient le courant
            curr := curr.next ; // Le courant devient le successeur
            lock(curr) ;
        if (key == curr.key) // Si l'élément est déjà dans l'ensemble
            return false ; // On veut pas avoir de duplications
        Node node := new Node(key) ; // Création d'un nouveau nœud
        node.next := curr :
        pred.next := node :
        return true ; // Tous c'est bien passer
    finally
        unlock(curr) ; // On lâche un lock sur les 2 qu'on a pris
finally
    unlock(pred) ; // On lâche le deuxième lock
```

boolean remove(int key)

```
lock(head) ; // Verrou sur la tête de la liste
try
  Node pred = head ;
  curr := pred.next ;
  lock(curr) ; // A ce stade lock sur l'élément head et le suivant
  try
    while (curr.next < key) // On fait la recherche, pendant la recherche on va déplacer nos lock.
      unlock(pred) ;
      pred := curr ; // Le prédécesseur devient le courant
      curr := curr.next ; // Le courant devient le successeur
      lock(curr) ;
    if (key == curr.key) // On a trouvé l'élément
      pred.next := curr.next ;
      return true ; // Tous c'est bien passer
    else
      return false ; // L'élément existe pas
  finally
    unlock(curr) ; // On lâche un lock sur les 2 qu'on a pris
finally
  unlock(pred) ; // On lâche le deuxième lock
```

Comment ça marche ?



Initialement

On va garder le lock sur **curr**, on va lâcher **pred**, on avance le pointeur et on lock le nouveau **curr**.

Cette chose, on le fait successivement

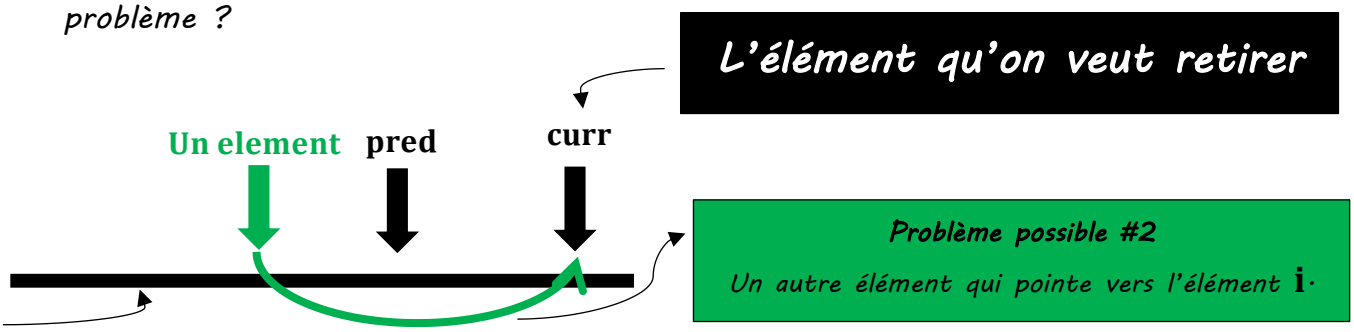
- *Maintenant, pour* **contains(key)**, *on fait la même chose (comme la partie de* **remove(key)** *ou on cherche l'élément)·*
- *Donc, dans ces codes la, il faut avoir les 2 lock pour assurer que le résultat est correct·*

On est toujours obligé de prendre les lock dans un certain ordre comme on fait ?

- *Par exemple, si* **add()** *prend les lock dans le sens qui prend, et* **remove()** *prend les lock dans l'autre sens ?*
- *Si* **add()** *veux insérer un élément, il va prendre* **head**, *et ensuite aller prendre le 2ème élément, mais si il commence par prendre d'abord le lock sur le 2ème élément, et l'autre processus commence par prendre le lock sur le* **head** ⇒ *blocage·*
- *Quand ils prennes les lock dans le même ordre : aucun problème !*

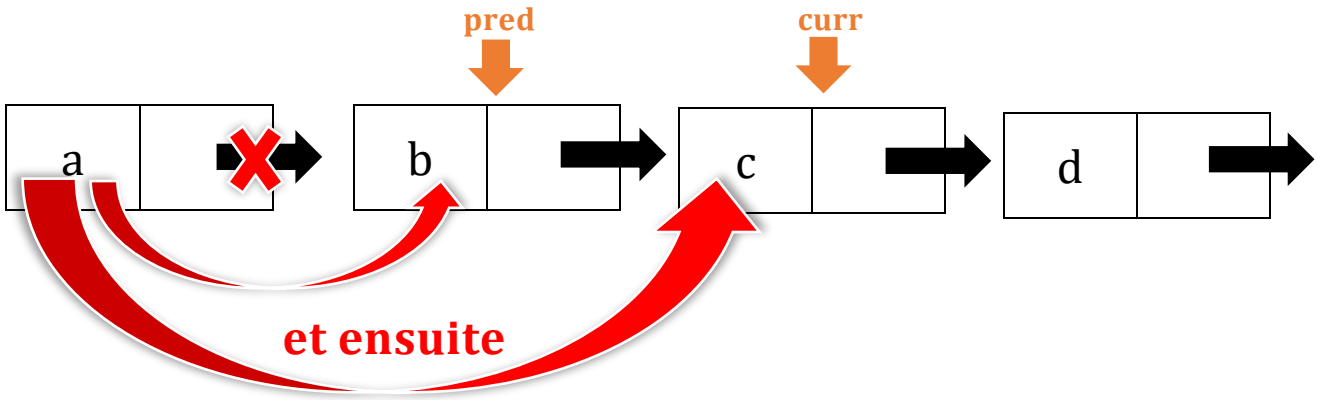
- ***On ne peut pas se dépasser*** *: Lorsque un processus fait une recherche, les processus qui se trouve derrière lui, ne peuvent pas le dépasser ; On a une sorte de file d'attente· Ça aussi, c'est un peu long...*

- ***Donc, si on essaie d'optimiser l'utilisation des lock, faire moins de synchronisation, Quesque on peut faire ?***
- *Si je « fonce » sur la structure sens prendre de lock, et je veux retirer l'élément* **i**, *et je prends les lock que lorsque j'arrive à* **i**· *Quelle est le problème ?*



De manière plus général, pendant que je me trouve à l'emplacement i, il y a d'autres processus qui peuvent venir et supprimer tous cette partie de la liste : par exemple, avant même d'avoir pris le lock, il y a un processus qui change le panorama, donc lorsque on va prendre le lock, tous le monde a déjà changé, donc c'est possible que je travaille sur une liste qui n'existe même plus (éléments plus accessible à partir du **head***).*

Dans l'exemple suivante *on veut retirer l'élément* **curr**, *mais un autre processus peut venir et faire lock sur le prédécesseur de* **pred**, *mais alors, lorsque ce processus lâche les locks, j'arrive a faire lock mais je ne suis pas au courant que la liste à changer. Je retire* **curr**, *et alors quand je fais «* **return true** *», la signification est que l'opération a réussi alors que c'est faux.*



Donc, l'idée de « foncer », faire lock et effectuer l'opération, c'est une bonne idée. Mais cela ne suffit pas car entre le moment de la recherche et le moment du lock, il se passe des choses qui peuvent modifier notre liste.

Alors c'est quoi que faut vérifier ?

- *Faut voir si l'environnement des pointers* **pred** *et* **curr** *a changé entre le moment ou j'ai vu l'élément et le moment où j'ai pris les lock.*

- *Comment voir si « ma » partie dans la liste n'est plus accessible ? Reparcourir la liste en repartant du* **head**, *si un processus a modifié la liste et a coupé « notre » partie de la liste du* **head**, *on va pouvoir identifier ça.*

boolean add(int key)

```
while true
    pred := head ;
    curr := pred.next ;
    while (curr.next < key)
        pred := curr ; // Le prédécesseur devient le courant
        curr := curr.next ; // Le courant devient le successeur
    lock(pred) ; lock(curr) ; // C'est que lorsque qu'on arrive qu'on prend les lock
    try
        if (validate(pred, curr)) // On regarde la situation
            if (key == curr.key) // Si l'élément est déjà dans l'ensemble
                return false ; // On veut pas avoir de duplications
            else
                Node node := new Node(key) ; // Création d'un nouveau nœud
                node.next := curr :
                pred.next := node :
                return true ; // Tous c'est bien passer
    finally
        unlock(pred) ; unlock(curr) ; // On lâche les lock
```

// Regarder si la structure n'a pas changer…

boolean validate(Node pred, Node curr)

```
node := head ;
while (node.key ≤ pred.key)
    if (node == pred)
        return (pred.next == curr) ;
    node := node.next ;
return false ;
```

- *Si on a dépassé* **pred**, **pred** *n'est plus accessible.*
- *Si je suis arriver a* **pred**, *je regarde que la situation n'a pas changé.*

boolean remove(int key)

```
while true
    pred := head ;
    curr := pred.next ;
    while (curr.next < key)
        pred := curr ; // Le prédécesseur devient le courant
        curr := curr.next ; // Le courant devient le successeur
    lock(pred) ; lock(curr) ; // C'est que lorsque qu'on arrive qu'on prend les lock
    try
        if (validate(pred, curr)) // On regarde la situation
            if (key == curr.key) // On a trouvé l'élément
                pred.next := curr.next ;
                return true ; // Tous c'est bien passer
            else
                return false ; // L'élément existe pas
    finally
        unlock(pred) ; unlock(curr) ; // On lâche les lock
```

boolean containes(int key)

```
while true
    pred := head ;
    curr := pred.next ;
    while (curr.next < key)
        pred := curr ; // Le prédécesseur devient le courant
        curr := curr.next ; // Le courant devient le successeur
    lock(pred) ; lock(curr) ; // C'est que lorsque qu'on arrive qu'on prend les lock
    try
        if (validate(pred, curr)) // On regarde la situation
            return (key == curr.key)
    finally
        unlock(pred) ; unlock(curr) ; // On lâche les lock
```

boolean validate(Node pred, Node curr)

On vérifie quoi ? Problèmes possibles ?

- *Ma zone devient inaccessible.*
- *Il se peut que j'arrive jusqu'à* **pred**, *sauf qu'avant,* **pred** *pointé sur* **curr** *mais maintenant un processus à changer ce pointer et donc à ce moment la* **pred** *n'est plus le predesseceur de l'élément courant.*

Comme je fais le **validate** *APRES avoir pris les lock, les choses risque pas de changer. Si la validation ne marche pas (la fonction retourne false) : on lâche les lock et on revient a la ligne « while true » pour essayer à nouveau.*

***Notre approche dans ce code est optimiste**, donc on commence sans prendre les lock.*