RAPPORT DE PROJET PCV

HUANG Heqing
LIU Yunkai

Ⅰ Sujet Choisit : CyclicBarrier

On choisit CyclicBarrier comme notre sujet de projet en construisant un modèle de 3-Producteurs. A l'aide de la synchronisation qui permet à un ensemble de threads à tous attendre les uns des autres pour atteindre un point d'arrête commun. CyclicBarrier est utilisé dans le programme impliquant un ensemble de threads avec une taille fixe qui devront attendre les uns des autres. La barrière est appelée cyclique, car il peut-être ré-utilisé après tous les threads en attente sont libérés.

Ⅱ Décrire la sémantique de l'outil

--Que se passe-t-il lors d'une suspension(file attente? géréé en FIFO?), que deviennent les verrous associés?

Chaque fois un thread arrive à la barrière(après appelle await()), le nombre de threads attendus va augmenter un, si ce nombre n'est pas égale à la valeur prévue de threads attendus, il doit attendre les autres jusqu'à nombre attendu est égale à la valeur prévue. Cette attente n'est pas file attente ni géréé en FIFO, car l'ordre de réveil est vraiment aléatoire. Quand un thread change la valeur de NumWaiting, il sera attribué un verrou sur cette attribut. Car chaque thread peut visiter et changer cet attribut, cette classe est comme une variable globale. Après ce changement, le verrou sera récupédé

--Que se passe-t-il lors du réveil(un thread ou plusieurs?), récupèrent-t-ils les verrous ou non?

Une fois le dernier thread arrive à la barrière bien, tous les threads étant en suspension seront réveils. La classes CyclisBarrier supporte une commande complémentaire ***Runnable***(est appelé par ce dernier thread) en option qui sera exécutée une fois par point de barrière. Après le dernier thread dans l'ensemble arrive, mais avant n'importe quel thread est libéré. Cete action de barrière est utile pour faire la mise a jour à l'état partagé avant les autres ensemble de threads continuent. Et quant à l'ordre de réveil, c'est aléatoire. Il n'y pas de file d'attente ni géréé en FIFO. Le verrou associé(verrou sur ***NumWaiting***) sera libéré.

--Les suspensions peuvent-elles être interrompues? comment? Quel est l'effet du réveil(récupération des verrous?)

Les suspensions peuvent-elles être interrompues. Si un thread arrive à la barrière mais pas le dernier, il va dormir jusqu'à tous les threads arrivent puis ils sont réveils em même temp, ou il peut-être être interrompue si l'une des situations suivantes apparaît:
Le délai spécifié écoulé(***timeout*** elapses).
Certains autre thread interrompt le thread courant.
Certains autre thread interrompt l'un des autres threads en attente.
Certains autre thread appelle ***reset()*** sur cette barrière.
Si un thread est interrompu en attendant par n'importe quelle situation, tous les autres threads en attente vont jeter ***BrokenBarrierException*** ou ***InterruptedException*** et la barrière est placée dans l'état cassé(isBroken() sera true).

--Peut-on avoir des attentes limitées dans le temps, comment se passe le réveil?

Une autre situation, si on met un ***timeout*** pendant l'attente, le thread va attendre jusqu'à ce que toutes les parties ont invoqué attendre sur cette barrière, ou le temps d'attente spécifié écoulé. Si l'indication du temps d'attente écoulé alors ***TimeoutException*** est levée. Si le temps est inférieur ou égal à zéro, le procédure ne sera pas attendre du tout.

Ⅲ Choisir les méthdes qui seront implantées

On a implanté les méthodes :

public int await(): Attendre jusqu'à tous les threads arrivent à cette barrière. Si le thread courant n'est pas le dernier thread qui arrive, alors il n'est pas nécessaire à faire quelque chose, il va dormir jusqu'à tous les threads arrivent ou les suspensions sont interrompues.
Par contre, si le thread est le dernier thread arrivant à cette barrière, et une action de barrière non null a été fourni dans le constructeur, alors le thread va appeler cette action de barrière avant faire les autres threads réveils.

public int getNumberWaiting(): Obtenir le nombre de threads actuellement en attente à cette barrière. Elle va retourner le nombre de threads qui sont bloqués en ***await()*** actuellement.

public int getParties(): Obtenir et retourner le nombre de threads nécessaires pour déclencher cette barrière.

public boolean isBroken(): Requête si cette barrière est dans un état cassé. Retourner **true** si un ou plusierus threads blesse la barrière à cause de l'interromption ou ***timeout***. Sinon retourne ***false***.

public void reset(): Restaurer la barrière à son état initial.

Et on n'a pas fait les méthodes :

public int await(long timeout, TimeUnit unit): Cette méthode étant comme ***await()***, juste ajouter un jugement pour surveiller si les threads attendent passer une durée ***timeout*** ou pas**.** Si oui, il va conduire à une interromption. Car en promela on ne trouve pas la méthode pour chronométrer, et puis cette méthode réalise prèsque la même fonctionalité avec ***await()***, donc on n'a pas besoin de l'implanter.

Ⅳ Faire une implantation de Promela

On a définit une structures pour décrire la clasee CyclicBarrier:

```
typedef cyclicbarrier
{
    int total;              //Nombre de Thread doit attendre au total

    int nready;             //Nombre de Thread qui a déjà attendu
    bool broken;            //vérifie si la barrière a été cassé ou pas,
                            // si true ça signifie un mauvais état
}
```

On a définit six inlines :

inline new(c,v): initialiser une nouvelle cyclicbarrier(c) et lui indiquer le

nombre de Thread doit attendre au total(v). Cette méthode va assigner la valeur de v à c.total, mettre c.nready égale à zéro et broken à false.

inline await(c): implanté de la méthode ***await()*** en java. Si un thread arrive la barrière, il faut c.nready++, afficher un message informant qu'il est bien arrivé. Et puis , il doit faire un jugement, si c.nready==c.total, ça signifie que tous les threads arrivent bien, donc on fait un ***skip*** pour s'arrêter là. Mais sinon, il va bloquer ici jusqu'à cette condition satisfait ou il y a une interruption se produit.

inline getNumberWaiting(c,res): implanté de la méthode ***getNumberWaiting()*** en java. retourne la valeur de c.nready. C'est le nombre de threads actuellement en attente à cette barrière. res c'est la variable pour stocker la valeur de retourner, La même suivante.

inline getParties(c, res): implanté de la méthode ***getParties()*** en java. retourne la valeur de c.total. le nombre de threads nécessaires pour déclencher cette barrière.

inline isBroken(c, res): implanté de la méthode ***isBroken()*** en java. retourne la valeur de c.broken. **true** si un ou plusierus threads blesse la barrière à cause de l'interromption ou ***timeout***. Sinon ***false***.

inline reset(c): implanté de la méthode ***reset()*** en java. On voit bien que, si la méthode ***reset()*** est appelé, la barrière soit restaurée à l'étati initial(non Exception), soit cassée par une interruption(throws une Exception). Dans le cas de c.nready==0 ou c.nready==c.total, ça vous dire soit aucun thread n'arrive ou tous les threads arrivent bien, la méthode va restaurer la barrière sans Exception et on met c.broken à false et c.nready à zéro. Sinon(soit c.nready>0 et c.nready<c.total), alors la barrière était cassée avec une Exception. On utilise assert() pour produire une Exception en promela.

Ⅴ Trouver un exemple en java

On condtruit un modèle Producteurs. En ce modèle, on a trois producteurs. Chaque producteur doit produire dix produits automatiquement et indépendantement. Si un producteur finit son tâche, il faut attendre les autres producteurs jusqu'à tout le monde finit

dix produits.

On note bien que dans ce modèle là, il y a une seule variable devra être protégée par le mécanisme synchronisation -- NumWaiting. Mais cet attribut éatait encapsulé par Classes Java. Normalement on ne la voit pas. Et il n'y pas d'autre ressource critique, Donc dans ce modèle on n'a pas de section critique qui est vraiment visible. Mais pour faire les résultat plus lisible, on ajoute une variable privée pour chaque producteur nomée ***rate***, montre la vitesse de production. Chaque producteur possède une ***rate*** différente, donc quand le producteur produit, on lui fait ***Thread.sleep(rate*500)*** comme une procédure inutile mais assez long pour obtenir un résultat plus apercevable, car les vitesses de production sont différentes donc les temps qui'ils arrivent à la barrière sont différents. ça vous dire une durée suffisante pour examiner bien les conflits apparaissent possibles pour l'accès aux sections critiques. Si un thread finit son tâche(déjà produit dix), il affiche un message pour indiquer lequel thread a déjà finit et combien de thread qui a déjà finit en ce moment là. Quand chaque producteur finit dix, il va afficher un message pour indiquer que tous les threads arrivent bien. Cette procédure est accompli par le dernier producteur(le dernier thread arrivant la barrière) à travers par une interface ***Runnable barrierAction()***.

Même s'il n'y pas de section critique visible, on peut traiter la méthode ***await()*** comme une procédure qui visite la section critique(car la valeur de NumWaiting était modifiée pendant cette méthode). Donc en chaque fois un thread veut entrer en sections critiques cela doit apparaître sur une trace(afficher une phrase ou un message sur la console indiquant les informations nécessaires par exemple Producteur 1 est attendu), idem lorsqu'il sort de section critique, idem lorsqu'il réveille les autres.

Ⅵ En faire un modèle en Promela

On fait le même modèle avec Java. Trois producteurs au totals à initialiser et chaque producteur produit ses 10 produits(le nombre produit par chaque producteur noté ***ctn***).
Dans la vérification exhaustives, il n'y pas de erreurs ni exceptions, ni delay.

Du coup, normalement quand un producteur finit 10 produits, il devra appeler la méthode ***await()*** pour attendre les autres threads. On a bien fait un test sur ce cas, puis on fait un autre test qui initialise deux producteurs normals mais un producteur_bug au début, les deux producteurs va appeler la méthode ***await()*** après finit leurs tâches comme d'habitude, mais pour le producteur_bug, il va appeler la méthode reset() apères finit son tâche. Ce test va indiquer deux problèmes:

Soit le producteur_bug était le premier thread qui y arrive, il n'y pas des Exceptions mais il va mettre zéro à la valeur de c.nready, après en ce moment là, il reste encore deux producteurs, si ils arrivent bien, la valeur de c.nready sera égale à deux, mais jamais trois, donc ça signifie tous les deux producteurs devront attendre infiniment. Il va conduire un interblocage.

Soit une Exception sera relevée car le poducteur_bug arrive pas le premier. Une BrokenBarrierException sera relevée.