Ecole Polytechnique

Cours : "Parallélisme" Travaux dirigés

E. Goubault & M. Martel

TD 3

22 janvier 2002

1 Ordonnanceur à tourniquet

On cherche à implémenter un ordonnanceur à tourniquet pour les threads JAVA en JAVA.

L'idée est que l'on doit définir un thread ordonnanceur qui gère une file d'attente de threads "utilisateurs" JAVA et les ordonnancer un à un pour un quantum de temps élémentaire donné (une constante de votre programme). Cet ordonnanceur ne gèrera pas de notion de priorité.

Chaque thread utilisateur devra s'enregistrer auprès de l'ordonnanceur en appelant la méthode `addThread()` (à écrire).

Avant de commencer, on répondra aux questions suivantes: quelle doit être la priorité du thread ordonnanceur? du thread utilisateur actif? des threads non encore actifs?

2 Lecteurs/rédacteurs

On considère maintenant une base de données partagée accédée par des threads JAVA de deux types. L'un est le type *lecteur*, l'autre le type *rédacteur*.

Les lecteurs sont des threads qui par définition ne peuvent que lire (interroger) la base de données. Les rédacteurs par contre peuvent lire et écrire. On suppose que la lecture parallèle de la même location dans la base de données est autorisée, par contre l'écriture en parallèle avec soit une lecture soit une autre écriture sur la même location doit être effectuée en section critique (exclusion mutuelle).

On simulera (il ne sera pas nécessaire d'avoir une vraie donnée, une simple simulation des actions à l'écran suffit) le début de l'accès en lecture par la méthode (à écrire) `startRead()`, la fin par `end Read()`, le début de l'accès en écriture par `startWrite()` et la fin par `endWrite()`.

On veut implémenter un tel système avec au moins la propriété suivante: aucun lecteur ne doit rester en attente sauf si un rédacteur a obtenu la permission d'utiliser la base de données. Peut-il y avoir des cas de famine?

On pourra également (optionnel) se poser la contrainte supplémentaire suivante: dès qu'un rédacteur est prêt, il doit effectuer son écriture "aussi vite que possible".

Remarque: on pourra soit utiliser la classe `Semaphore` du cours (combien en faut-il et de quel type alors?), soit directement les mots-clés JAVA `wait()` et `notify()`.

3 Sémantique

Le but de ce problème est d'étendre la sémantique vue en cours afin d'y incorporer des versions simplifiées des primitives `synchronized`, `wait` et `notify`. Pour cela nous allons procéder en plusieurs étapes.

(i) Définir le comportement de la commande `synchronized(o){ prg }` à l'aide des primitives présentes dans le mini-langage du cours.

(ii) De la même manière, définir le comportement d'une version simplifiée des primitives `wait` et `notify`. On fera les hypothèses simplificatrices suivantes :

- à la différence de Java, la méthode `wait()` suspend l'exécution du thread courant t_c jusqu'à ce que celui-ci soit réveillé par un autre thread par la primitive $\texttt{notify}(t_c)$.
- on suppose que tous les threads ont accès à un tableau de sémaphores binaires, appelé `isActiveThread`, présent en mémoire partagée et alloué lors du lancement de la JVM. On pourra indiquer comment l'élément i du tableau est initialisé lors de l'invocation de la méthode `start()` du i-ième thread.

(iii) Vérifier formellement le bon fonctionnement des primitives définies à la question précédente à l'aide de la sémantique par système de transitions du cours. Plus précisément, nous nous intéressons aux états initiaux de la forme :

$$I = (b_1, \ldots, b_m; \texttt{wait()}; b'_m, \ldots, b_n; \texttt{notify(m)}; b'_n, \ldots, \kappa_0)$$

où κ_0 est la fonction qui donne uniformément 1. Montrer que, dans toute trace d'exécution, c.à.d. dans tout chemin du graphe de transition partant de l'état I, les transitions partant de l'état b'_m sont toujours précédées (pas forcément immédiatement) de la transition étiquetée par l'instruction implémentant le `notify(m)`.

(iv) Nous souhaitons maintenant ajouter directement les primitives `wait` et `notify` sans au langage sans les définir à partir des autres opérations. Définir les nouveaux espaces des états, des étiquettes, ainsi que les nouvelles transitions devant être présentes dans la sémantique par système de transition. On enrichira les états par une fonction τ associant à chaque thread sont état d'activité.

(v) Appelons S la sémantique vue en cours et S' la sémantique définie à la question précédente. Pour conclure, nous allons prouver la correction de S' par rapport à la sémantique S dans laquelle les primitives `synchronized`, `wait` et `notify` sont définies à l'aide des sémaphores. Montrer que pour tout chemin c' de longueur n de S' partant de l'état initial, il existe un chemin c de longueur n dans S partant de l'état initial tel que

$$\forall i,\ 1 \leq i \leq n,\ \tau_i(t) = \texttt{isActiveThread}_i\texttt{[t]}$$

où τ_i représente la fonction τ présente dans le i-ième état du chemin dans S' et $\texttt{isActiveThread}_i$ représente la valeur du tableau `isActiveThread` dans le i-ième état dans S.

(vi) Enrichir S' afin d'y incorporer directement la primitive `synchronized(o){ prg }`. Prouver la correction de la nouvelle sémantique en vous inspirant de la question 5.

(vii) Implémentation : créer une classe Java appelée `myThread` étendant la classe `Thread` et contenant des primitives `myWait` et `myNotify` fonctionnant de la manière décrite à la question 2.

4 Points morts

On considère un système constitué de 4 ressources partagées du même type (disons des entiers...), protégées par un mutex (sémaphore binaire - cela veut dire que l'on supposera leurs lectures et écritures automatiquement mis dans un bloc synchronized), auxquelles accèdent 3 processus en parallèle, chacun d'entre eux ayant besoin d'au plus 2 de ces ressources. Montrer qu'il ne peut pas y avoir d'interblocage.