Examen 2012-2013 Temps Réel

Le 20/6/2013, Tout document autorisé. Tout dispositif électronique interdit.
Durée : 1 heure et demie

Vous devez obligatoirement **justifier** chacune de vos réponses.
Questions indépendantes :

1. Avantages et inconvénients de n'utiliser chacun des verrous qu'en un endroit unique dans le code.

2. Avantages et inconvénients d'avoir beaucoup ou peu de verrous différents.

3. Comment choisissez-vous l'ordonnanceur de votre système temps réel ?

4. Citez un cas (général) où l'on a besoin de 2 verrous pour une structure de donnée unique que l'on veut protéger globalement.

5. Quels problèmes se posent quand on a des périphériques très très actifs sur une machine temps réel. Comment les résoudre (au niveau système et/ou *driver*)

6. Pour quelles raisons utilise-t-on des *threads* alors que c'est compliqué à mettre en oeuvre ?

7. Quels sont les avantages et inconvénients de l'implémentation en espace utilisateur des *mutex* ?

8. Comment faites-vous pour appliquer la méthode RCU si la donnée est un arbre binaire ?

9. La donnée manipulée est une liste doublement chaînée circulaire. Comme elle est très souvent modifiée, il y a un verrou par cellule de la liste. Comment gérez-vous les verrous, quand vous faites une insertion, ?

10. Soit la fonction suivante :

    `void swap(void**a, void**b) { void *tmp ; tmp = *a ; *a = *b ; *b = tmp ;}`

    En supposant cette fonction atomique, implémentez l'insertion d'une nouvelle cellule dans une liste **sans utiliser de verrou** :

    ```
    struct cell { int data ;
                  struct cell *next ;
                } ;

    void insertion(struct cell **ici, int valeur) ; // A vous de la définir

    struct cell* une_nouvelle_liste()
    {
        struct cell *ma_liste = NULL ;  // Liste vide
        insertion(&ma_liste, 6) ;       // 6
        insertion(&ma_liste->next, 5) ; // 6 5
        insertion(&ma_liste, 4) ;       // 4 6 5
        return ma_liste ;
    }
    ```

    Evidemment, cet exemple est stupide car la liste n'est utilisée que par une seule *thread*, il n'y a donc pas besoin de verrou. Votre algorithme devra fonctionner quand plusieurs *threads* feront simultanément des insertions dans la liste.

Une correction :

1. Avantages et inconvénients de n'utiliser chacun des verrous qu'en un endroit unique dans le code.

   Peu de verrous car il n'y a pas un verrou par instance d'objet.

   Les sources sont plus propres car chaque type objet n'est modifié qu'à un seul endroit dans le code.

   Une analyse statique de l'arbre d'appel des fonctions permet de savoir s'il peut y avoir des deadlocks.

   C'est forcément un verrouillage à gros grain qui se parallélise mal quand il y a beaucoup d'objets.

2. Avantages et inconvénients d'avoir beaucoup ou peu de verrous différents.

   Beaucoup de verrous : plus de parralélisme.

   Moins de verrous : moins de deadlock possible. moins de place mémoire utilisée pour stocker les verrous.

3. Comment choisissez-vous l'ordonnanceur de votre système temps réel ?

   Fiabilité absolue : hors-ligne

   Fiabilité des tâches critiques : HPF

   Utilisation maximale des ressources : LLF

   Dans tous les cas, on ne peut pas faire un système temps réel si l'on ne connaît pas la capacité des tâches.

4. Citez un cas (général) où l'on a besoin de 2 verrous pour une structure de donnée unique que l'on veut protéger globalement.

   lecteur/écrivain

5. Quels problèmes se posent quand on a des périphériques très très actifs sur une machine temps réel. Comment les résoudre (au niveau système et/ou *driver*)

   Beaucoup d'interruptions qui volent du temps CPU.

   Si le périphérique est très actif, il y a toujours des données à traiter. Il suffit donc de désactiver les interruptions pour ce périphérique et faire faire du polling dans une tache périodique à faible priorité.

6. Pour quelles raisons utilise-t-on des *threads* alors que c'est compliqué à mettre en oeuvre ?

   Lancement et arrêt peu coûteux en ressources.

   Permet de ne pas être bloqué par les entrées sorties.

   Parrallélisation avec partage de mémoire facile.

7. Quels sont les avantages et inconvénients de l'implémentation en espace utilisateur des *mutex* ?

   Pas d'appel système, donc plus rapide.

   Ne permet pas facilement de mettre en suspend la tâche. donc seulement dans le cas multiprocesseurs avec des sections critiques courtes.

8. Comment faites-vous pour appliquer la méthode RCU si la donnée est un arbre binaire ?

   Dans tous les cas l'écrivain verrouille l'arbre complet.

   Pour chaque noeud modifié, un nouveau noeud est créé.

   S'il y a plusieurs noeuds modifiés, on recrée tous les noeuds sur le chemin de leur ancêtre commun.

   On remplace l'ancêtre commun par sa nouvelle version et on déverrouille.

   La place et le temps sont en Log2 du nombre de feuilles si l'on est obligé de remonter à la racine.

   Il ne s'agit en aucun cas d'une recopie de l'arbre.

9. La donnée manipulée est une liste doublement chaînée circulaire. Comme elle est très souvent modifiée, il y a un verrou par cellule de la liste. Comment gérez-vous les verrous, quand vous faites une insertion, ?

   Il faut verrouiller les 2 cellules autour du point d'insertion.

   Mais pas la précédente puis la suivante : en effet il n'y a pas d'ordre total sur une liste circulaire. On peut arriver à un deadlock.

   Donc on verrouille en premier la cellule qui a la plus petite adresse en mémoire (ordre total).

10. Soit la fonction suivante :

    `void swap(void**a, void**b) { void *tmp ; tmp = *a ; *a = *b ; *b = tmp ;}`

    En supposant cette fonction atomique, implémentez l'insertion d'une nouvelle cellule dans une liste **sans utiliser de verrou** :

```
struct cell { int data ;
              struct cell *next ;
            } ;

void insertion(struct cell **ici, int valeur) ; // A vous de la définir

struct cell* une_nouvelle_liste()
{
    struct cell *ma_liste = NULL ;  // Liste vide
    insertion(&ma_liste, 6) ;       // 6
    insertion(&ma_liste->next, 5) ; // 6 5
    insertion(&ma_liste, 4) ;       // 4 6 5
    return ma_liste ;
}

void insertion(struct cell **ici, int valeur)
{
  struct cell *nouvelle = malloc(sizeof(*nouvelle)) ;
  nouvelle->data = valeur ;
  nouvelle->next = nouvelle ;
  swap(ici, &nouvelle->next) ;
}
```

    `nouvelle->next=*ici;swap()` *ne peut fontionner car la valeur pointée par* `ici` *peut changer avant l'appel à* `swap`*. Il n'y a pas atomicité.*