R. Namyst
LaBRI, Bureau 322

le 1/12/2004

Licence/Master d'informatique — Systèmes d'exploitation
Devoir Surveillé

Durée : 1h30 — Notes de cours autorisées

1 Ordonnancement de processus (question d'échauffement)

Question 1 Définissez brièvement ce que l'on appelle "ordonnanceur" dans un système d'exploitation. Quand et par qui le code d'ordonnancement est-il exécuté (listez des situations bien précises) ?

Question 2 Dans un système Unix, quels sont les principaux paramètres pris en compte pour le choix du prochain processus à exécuter ?

2 Nachos et la vraie vie

Le simulateur Nachos, bien que très réaliste sur de nombreux aspects, introduit tout de même plusieurs simplifications par rapport aux systèmes d'exploitations réels. En particulier, la protection des régions de code nécessitant un accès en exclusion mutuelle s'effectue dans Nachos en masquant les interruptions pendant toute la durée du traitement (au moyen de l'instruction `interrupt->SetLevel(IntOff)`). En voici une illustration avec le code de la primitive `Semaphore::V()` :

```
void Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL)    // make thread ready, consuming the V immediately
        scheduler->ReadyToRun(thread);
    value++;

    (void) interrupt->SetLevel(oldLevel);
}
```

Question 1 Expliquez pourquoi, sur une machine multiprocesseurs, une telle stratégie ne fonctionnerait pas.

Question 2 Quel mécanisme de bas niveau peut-on utiliser dans les noyaux pour protéger l'accès à des portions de code très courtes ? Décrivez précisément une situation où cette utilisation peut conduire à de très mauvaise performances (pensez à des processus s'exécutant sur le même processeur).

Question 3 Expliquez comment, en utilisant conjointement le mécanisme précédent et le masquage des interruptions, on pourrait résoudre ce problème (on ne demande pas d'écrire du code précis).

3 Synchronisation

Dans cet exercice, il s'agit d'utiliser des *sémaphores* pour synchroniser une fonction `thread_func` qui nécessite des précautions particulières lorsqu'elle est exécutée de manière simultanée par plusieurs processus légers. Le nombre de processus légers du programme n'est pas défini a priori. Toutefois, on sait que chaque processus léger appartient à un groupe (et un seul), les groupes étant numérotés de 0 à $N-1$ (N est une constante du problème). Lorsqu'un processus léger appelle la fonction `thread_func`, il lui passe le numéro du groupe auquel il appartient en paramètre (i.e. `grp`).

```
#define N ??

... // déclaration des sémaphores & variables globales

void thread_func(unsigned grp)
{
   ... // synchronisation à ajouter

   do_compute(grp);

   ... // synchronisation à ajouter
}
```

Les contraintes de synchronisation sont dictées par l'implémentation de la fonction `do_compute` **qui ne peut être exécutée en parallèle que par des threads appartenant au même groupe**. Autrement dit : Un thread du groupe i ne peut démarrer l'exécution de `do_compute(i)` que lorsqu'aucun thread d'un groupe j ($j \neq i$) n'est en train de l'exécuter. Le code de synchronisation doit par conséquent permettre l'exécution concurrente de `do_compute` par plusieurs threads du même groupe lorsque c'est possible. Il faut bien entendu bloquer les processus lorsque la situation ne leur permet pas d'exécuter `do_compute`.

Question 1 En remarquant que ce problème présente certaines (voire beaucoup de) similitudes avec le problème des lecteurs/rédacteurs vu en cours, proposez une solution **la plus simple possible** permettant de répondre aux contraintes posées.

Par exemple, un bon point de départ serait d'utiliser un tableau d'entiers `unsigned nb[N]` qui permettrait de compter, pour chaque groupe, le nombre de threads en train d'exécuter la fonction `do_compute`. De cette manière, il est aisé de faire faire un traitement spécial au premier thread arrivant/dernier thread sortant...

Donnez le code de synchronisation entourant l'appel à `do_compute`. N'oubliez pas de préciser les valeurs initiales des variables et sémaphores que vous allez déclarer.

Question 2 Il s'agit maintenant d'étendre cette solution pour la rendre *équitable*, c'est-à-dire qu'il va falloir prendre en compte l'ordre d'arrivée des threads dans la fonction `thread_func` pour régir l'accès à la fonction `do_compute`.

En supposant que les sémaphores gèrent les processus bloqués dans un ordre FIFO, il est possible d'utiliser un sémaphore additionnel qui jouerait le rôle de "compte-goutte", en empêchant les threads de se "doubler" à cause de la synchronisation mise en place à la question précédente.

Donnez une nouvelle version du code de `thread_func` qui préserve l'équité de progression entre les différents threads (indiquer aussi les nouvelles déclarations nécessaires).

Question 3 On souhaite maintenant borner le nombre de threads exécutant simultanément la fonction `do_compute` par une constante `MAX` (i.e. au plus `MAX` threads peuvent exécuter `do_compute` en parallèle).

Indiquez ce qu'il faut modifier dans le code précédent pour respecter cette contrainte supplémentaire.

Question 4 (bonus) On se replace maintenant dans les conditions définies avant la question 3. On souhaite accorder une priorité forte aux threads du groupe 0, c'est-à-dire que dès qu'un thread de ce groupe a débuté l'exécution de `thread_func`, il doit être assuré qu'il pourra exécuter `do_compute` dès que possible, même si d'autres threads (d'autres groupes) attendent depuis longtemps que les conditions soient favorables pour exécuter `do_compute`...

Indiquez ce qu'il faut modifier dans la solution proposée à la question 2 pour assurer cette propriété.