R. Namyst
LaBRI, Bureau 322

Licence/Master d'informatique — Systèmes d'exploitation
Correction du Devoir Surveillé du 1/12/2004

Remarque préliminaire : *Certaines réponses proposées aux questions de cours sont volontairement "verbeuses", et ce dans un but pédagogique. Bien sûr, il n'était pas nécessaire de fournir autant de détails pour répondre correctement aux questions (et donc obtenir le maximum de points) ...*

1 Ordonnancement de processus (question d'échauffement)

Question 1 L'ordonnanceur d'un système est le code qui est invoqué lorsqu'il s'agit de décider s'il faut provoquer un changement de contexte et, si oui, au profit de quel processus il faut l'effectuer. La plupart du temps, ce code est exécuté « involontairement » par les processus lors de leur passage en mode noyau, par exemple dans les situations suivantes :

- lors du blocage d'un processus (ou de sa terminaison), auquel cas il faut obligatoirement placer un autre processus sur le processeur ;
- lors du réveil d'un processus, pour voir si ce dernier mérite de s'exécuter immédiatement ;
- lorsqu'une interruption déclenchée par le temporisateur (« l'horloge ») signale la fin d'un quantum de temps ;
- lors d'un appel système tel que `Yield()` ;
- etc.

Question 2 Les principaux critères que l'ordonnanceur examine lorsqu'il inspecte la liste des processus prêts pour déterminer le prochain qu'il exécutera sont :

- la priorité statique des processus ;
- la priorité dynamique calculée par l'ordonnanceur (principalement en fonction du temps processeur déjà consommé par le processus) ;
- le processeur sur lequel ils se sont exécuté la dernière fois (affinité par rapport aux caches) ;
- l'appartenance (ou non) au même espace d'adressage que le processus qui vient de s'exécuter sur le processeur ;
- d'éventuels attributs empêchant certains processus de s'exécuter sur certains processeurs (directives du programmeur) ;
- etc.

2 Nachos et la vraie vie

Question 1 L'instruction qui désactive (ou plus précisément « masque ») les interruptions est employée pour s'assurer qu'il n'y aura pas de changement de contexte intempestif pendant l'exécution du code à protéger. Dans un contexte multiprocesseurs, cela ne suffit pas à assurer qu'aucun autre thread pourra exécuter le code puisque l'on empêche pas l'autre processeur de travailler... Par ailleurs, l'instruction de masquage des interruptions n'a d'effet que pour le processeur sur lequel elle est exécutée. Mais quand bien même elle aurait un effet sur l'ensemble des processeurs de la machine, elle ne suffirait pas à assurer l'exclusion mutuelle de toute façon...

Question 2 Pour protéger l'accès aux portions de code très courtes, on peut utiliser une instruction atomique telle que `test_and_set` pour implanter un verrou élémentaire. Toutefois, si on remplace systématiquement les instructions `interrupt_on/interrupt_off` par la paire `lock/unlock` implémentée avec `test_and_set`, on risque de voiir surgir des situations où un processus détenant le verrou est préempté au profit d'un processus tentant d'acquérir le même verrou, ce qui provoquera une période d'attente active importante puisqu'elle durera jusque la fin du quantum de temps...

Question 3 Pour éviter les situations telles que celle décrite précédemment, on pourrait ajouter un `yield()` pour provoquer un changement de contexte lorsque `test_and_set` échoue, mais ce n'est pas

une bonne solution car elle aura un impact néfaste (du point de vue des performances) sur les situations où les procesus compétiteurs se trouvent sur des processeurs différents.

Une solution serait donc de désactiver les interruptions, pour empêcher les changements de contexte sur le processeur, puis d'utiliser `test_and_set` pour acquérir le verrou servant d'arbitre entre les processeurs...

3 Synchronisation

Question 1 Il suffisait de reprendre le code des processus *lecteurs* (cf cours) et de changer `mutex` en `mutex[grp]` et `nb` en `nb[grp]`. Le principe de fonctionnement est le suivant : le premier thread d'un groupe `grp` tente d'acquérir le `jeton` qui permettra à son groupe d'obtenir le droit d'exécuter la fonction `do_compute`. Les threads suivants (du même groupe) passent sans se bloquer dès qu'il y parvient... Les premiers threads des autres groupes restent bloqués sur `P(jeton)`.

```
semaphore mutex[N](1);
semaphore jeton(1);
unsigned nb[N] = {0, ..., 0};

void thread_func(unsigned grp)
{
   P(mutex[grp]);
   if(++nb[grp] == 1)
      P(jeton);
   V(mutex[grp]);

   do_compute(grp);

   P(mutex[grp]);
   if(--nb[grp] == 0)
      V(jeton);
   V(mutex[grp]);
}
```

Cette solution n'est pas équitable car elle avantage terriblement les threads du même groupe que ceux en train d'exécuter `do_compute`.

Question 2 En supposant que les sémaphores gèrent les processus bloqués dans un ordre FIFO, on propose d'utiliser un sémaphore `sas_d_entree` qui va nous permettre de réguler l'arrivée des threads à l'entrée de la fonction `thread_func` :

```
semaphore sas_d_entree(1);

void thread_func(unsigned grp)
{
   P(sas_d_entree);

   P(mutex[grp]);
   if(++nb[grp] == 1)
      P(jeton);
   V(mutex[grp]);

   V(sas_d_entree);

   do_compute(grp);
   ...
}
```

Voici un exemple de scénario illustrant comment cela fonctionne :

1. un thread du groupe 0 arrive : il passe le `sas_d_entree` sans problème, obtient le `jeton`, et peut donc exécuter la fonction `do_compute` ;
2. un second thread du groupe 0 arrive : il passe également le `sas`, n'essaie pas d'obtenir le `jeton` et exécute la fonction `do_compute` ;
3. il en va de même pour tous les threads du groupe 0 qui arrivent consécutivement ;
4. lorsqu'un thread du groupe 1 arrive, c'est le premier du groupe donc il tente d'acquérir le `jeton`, ce qui le bloque sur `P(jeton)` ;
5. notons que ce thread n'a pas relâché le `sas_d_entree`, ce qui provoque le blocage sur ce sémaphore de tous les threads qui arriveront dorénavant (et on sait qu'ils seront relâchés dans leur ordre d'arrivée par hypothèse) ;
6. le point important est que même les threads du groupe 0 sont maintenant bloqués au niveau du `sas_d_entree`, donc ils ne doublent plus les autres threads ;
7. lorsque tous les threads du groupe 0 qui exécutaient `do_compute` ont terminé, le `jeton` est relâché et c'est donc au tour du premier thread du groupe 1 de s'en emparer ;
8. ce thread relâche le `sas_d_entree` et de deux choses l'une : soit le thread suivant est aussi du groupe 1, alors il pourra exécuter `do_compute` directement, soit il est d'un groupe différent (par exemple 0) et il se bloquera en tentant d'acquérir le `jeton`...

Question 3 Il suffit de déclarer un sémaphore `limite` initialité à `MAX` et d'encadrer l'appel à `do_compute` avec `P(limite)` et `V(limite)`.

Question 4 (bonus) Voici une solution possible :

```
semaphore mutex[N](1);
semaphore jeton(1);
semaphore jeton_bis(1);
semaphore sas_d_entree(1);

unsigned nb[N] = {0, ..., 0};

void thread_func(unsigned grp)
{
   if(grp) P(sas_d_entree);

   P(mutex[grp]);
   if(++nb[grp] == 1) {
      if(grp) P(jeton);
      P(jeton_bis);
   }
   V(mutex[grp]);

   if(grp) V(sas_d_entree);

   do_compute(grp);

   P(mutex[grp]);
   if(--nb[grp] == 0) {
      if(grp) V(jeton);
      V(jeton_bis);
   }
   V(mutex[grp]);
}
```