INF463 — Systèmes d'exploitation
Correction du Devoir Surveillé du 20/11/2008

1 Ordonnancement de processus

Question 1 Dans un système d'exploitation interactif, un temporisateur matériel envoie des interruptions (souvent appelées «interruptions d'horloge») à intervalles régulier. Chacune de ces interruptions force le processeur à retourner en mode noyau, ce qui permet à ce dernier de ré-attribuer le processeur si nécessaire. Ainsi, puisqu'un processus exécutant une boucle infinie voit sa priorité dynamique décroître au fil du temps, il arrivera un moment où sa priorité deviendra inférieure à celle d'autres processus prêts. À ce moment, l'ordonnanceur effectuera un changement de contexte vers l'un d'entre eux.

Question 2 Dans le cas où les autres processus ont une priorité de base très basse, le processus exécutant une boucle infinie sera certes moins souvent «préempté» par les autres, mais il le sera quand même régulièrement.

Question 3 Dans le simulateur Nachos, l'option `-rs` permet de simuler le comportement d'un système interactif en forçant des changements de contexte à certains endroits du code. Plus précisément, c'est la fonction `interrupt->OneTick()` qui simule «le temps qui passe» en faisant progresser le temps virtuel et en déclenchant pseudo-alétoirement (cf l'argument `seed` de l'option `-rs`) un changement de contexte.

Les limites de ce mécanisme résident dans le fait qu'aucun changement de contexte ne peut survenir en dehors d'un appel à `interrupt->OneTick()` (typiquement appelée par `interrupt->SetLevel()`). Donc il n'est malheureusement pas possible d'observer de bugs liés à des accès concurrents à des données non protégées («*race conditions*») dans la plupart des cas.

2 Outils de synchronisation

Question 1 `test_and_set` est une instruction machine. Elle positionne à 1 la valeur de l'entier dont l'adresse est passée en paramètre, et retourne son ancienne valeur. Voici son équivalent en C (sauf que cette version n'est pas atomique) :

```
int test_and_set(int *ptr)
{
    int old = *ptr;
    *ptr = 1;
    return old;
}
```

Sa propriété fondamentale est qu'elle s'exécute de manière **atomique** vis-à-vis de la mémoire.

Question 2 Cette version alternative de `mutex_lock` s'appuie sur le principe suivant : si l'appel à `test_and_set` renvoie 1, c'est que le verrou est déjà pris, donc il est inutile de retenter un `test_and_set` tant que la valeur du verrou n'est pas redevenue 0...

L'intérêt de procéder de la sorte est de passer l'essentiel de l'attente à lire une valeur en mémoire (ce qui est très efficace depuis un cache) plutôt que de polluer le bus mémoire en exécutant sans cesse une instruction atomique complexe.

Question 3 Lorsqu'un processus se trouve en section critique (après un appel réussi à `mutex_lock`), il peut être interrompu et retiré du processeur par l'ordonnanceur au profit d'un autre processus. Si ce dernier essaie à son tour d'acquérir le verrou, il va gaspiller en vain la totalité de son «quantum de temps processeur» à attendre la libération du verrou, ce qui n'arrivera pas d'ici le prochain changement de contexte (puisque le détenteur ne s'exécute pas pendant ce temps) !

Question 4 Dans le simulateur Nachos comme dans d'autres systèmes, le masquage des interruptions masque en particulier les interruptions «d'horloge», ce qui évite tout changement de contexte intempestif. On garantit donc au processus courant de ne pas être interrompu pendant toute la section critique...

Sur une machine multiprocesseur, cette technique ne fonctionne évidemment pas car elle n'empêche pas un processus s'exécutant sur un processeur différent d'accéder à la section critique.

Question 5 À la fin d'une section critique, Nachos exécute `interrupt->SetLevel(oldLevel)` pour revenir à l'état de masquage qui précédait la section. Ceci est particulièrement important lors d'appels imbriqués à des fonctions utilisant des sections critiques : si une fonction imbriquée rétablissait brutalement les interruptions avec `interrupt->SetLevel(IntOn)`, les changements de contexte seraient à nouveau possibles dans les fonctions englobantes, potentiellement en pleine section critique...

Question 6 L'idée est d'utiliser d'abord le masquage des interruptions pour désactiver les changements de contexte sur le processeur courant, puis d'utiliser `test_and_set` pour arbitrer l'accès à la section critique entre plusieurs processeurs différents. Voici ce à quoi cela pourrait ressembler :

```
void mutex_lock(mutex_t *m)
{
    disable_interrupts();
    while(test_and_set(&m->value) == 1)
        /* rien */ ;
}

void mutex_unlock(mutex_t *m)
{
    m->value = 0;
    enable_interrupts();
}
```

3 Barrières de synchronisation en deux temps

```
unsigned nb = 0;
mutex_t mutex;
cond_t wait;

void signaler()
{
    mutex_lock(&mutex);
    nb++;
    if(nb == MAX)
        cond_bcast(&wait);
    mutex_unlock(&mutex);
}

void attendre()
{
    mutex_lock(&mutex);
    if(nb < MAX)
        cond_wait(&wait, &mutex);
    mutex_unlock(&mutex);
}
```