université de BORDEAUX

Collège Sciences et technologies

Année	2017	Type	Devoir Surveillé
Master	Informatique		
Code UE	4TIN705U	Épreuve	Systèmes d'Exploitation
Date	27/10/2017	Documents	Non autorisés
Début	10h15	Durée	1h30

1 Questions de cours (échauffement)

Question 1 Expliquez précisément ce qui se passe lorsqu'un processus effectue un appel système bloquant (par exemple une lecture sur son entrée standard). À l'aide d'un joli chronogramme, expliquez également la séquences des événements qui vont conduire à le débloquer, dès lors que la touche « entrée » a été tapée dans le bon terminal...

Question 2 Expliquez le principe de la segmentation mémoire. Détaillez (à l'aide un dessin) le support matériel offert par un processeur supportant quatre segments par processus. Donnez les principaux avantages et inconvénients de la segmentation mémoire, comparativement aux systèmes ne supportant que des espaces d'adressages monolitiques.

2 Barrières de synchronisation en deux temps

On s'intéresse aux barrières de synchronisation dites « en deux temps », où l'appel à `barriere` est remplacé par deux appels (respectivement à `signaler` et à `attendre`). Comme illustré sur la figure suivante, un processus appelle `signaler` lorsqu'il a terminé un bloc d'instructions A_i. Puis il exécute le bloc B_i immédiatemment. Avant de débuter l'exécution de C_i, il exécute `attendre` (qui est potentiellement bloquante) pour s'assurer que tous les autres processus j aient terminé d'exécuter les blocs A_j. Autrement dit, $\forall i, j$, on doit assurer qu'aucune exécution d'un bloc C_j ne puisse débuter tant que toutes les exécutions des blocs A_i ne sont pas terminées. Les blocs B_k permettent juste de faire des choses utiles en retardant le moment où le processus risque de se bloquer.

P_i :

```
A_i
signaler ();
B_i
.....attendre ();.....
C_i
```

Question 1 En utilisant des **sémaphores** pour assurer la synchronisation (ainsi que des variables globales si nécessaire), donnez le code des fonctions `signaler` et `attendre`. On suppose qu'il existe une constante `MAX` indiquant le nombre de processus participant à la barrière.

Question 2 Dans le cas où on voudrait utiliser cette barrière en deux temps plusieurs fois dans la vie d'un processus (i.e. dans une boucle par exemple), indiquez comment il faudrait modifier votre solution (on ne demande pas le code complet).

3 Synchronisation dans Nachos

Voici comment sont implantés les *sémaphores* dans Nachos :

```
void Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interrupts
    while (value == 0) {                                // semaphore not available
        queue->Append((void *)currentThread);   // so go to sleep
        currentThread->Sleep();
    }
    value--;                                // semaphore available: consume its value
    (void) interrupt->SetLevel(oldLevel);         // re-enable interrupts
}

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);
}
```

La fonction `Sleep` place le thread courant dans l'état bloqué et passe immédiatement la main à un autre. La fonction `ReadyToRun` place le thread dans l'état prêt.

Question 1 En vous inspirant de cette implémentation des sémaphores, donnez une implémentation des fonction `Acquire` et `Release` de la classe `Lock` (en utisant les variables de classe définies ci-dessous) permettant d'utiliser des moniteurs en Nachos. Les fonctions `Acquire` et `Release` sont respectivement les équivalents des fonctions `mutex_lock`, et `mutex_unlock`. On rappelle qu'un thread ne peut pas relâcher un verrou qu'il ne détient pas.

```
class Lock
{
  public:
    void Acquire ();
    void Release ();
 private:
    List *queue;   // threads waiting in Acquire() for the lock to be free
    Thread *owner; // Owner of the Lock, or NULL if lock is free
};
```

Question 2 Même question à propos des fonctions `Wait` et `Signal` de la classe `Condition`, qui implémente les conditions que l'on peut utiliser lorsque l'on détient un moniteur.

```
class Condition
{
  public:
    void Wait (Lock * lock);
    void Signal ();
  private:
    List *queue; // threads waiting on condition
};
```