université de BORDEAUX

Collège Sciences et technologies

Année	2021-2022	**Type**	Devoir Surveillé
Master	Informatique		
Code UE	4TIN705U	**Épreuve**	Systèmes d'Exploitation
Date	13/11/2021	**Documents**	Non autorisés
Début	10h30	**Durée**	1h30

La plupart des questions peuvent être traitées même si vous n'avez pas répondu aux précédentes. *À la fin du sujet, vous trouverez un memento vous rappelant la syntaxe de quelques fonctions utiles.*

1 Questions de cours (échauffement)

Question 1 Décrivez brièvement le principe de la segmentation mémoire. De quel support matériel a-t-on besoin pour l'implémenter (faites un petit dessin pour un processeur autorisant 4 segments) ? Donnez quelques avantages et inconvénients de cette technique.

Question 2 Rappelez la différence fondamentale entre processus et *threads*. Comment le noyau prend-t-il en compte ces deux catégories d'acteurs dans sa stratégie d'ordonnancement ?

Question 3 L'algorithme d'ordonnancement implanté dans les noyaux Linux $2.4.x$ utilise un système de crédits que les processus sont autorisés à utiliser pendant la durée d'une « *époque* ». Tracez un petit chronogramme sur une durée de deux époques illustrant à quels moments surviendront les changements de contexte pour la configuration suivante de deux processus (`bash` et `gcc`) :

— la priorité statique de `bash` et `gcc` leur donne droit à 2 crédits chacun initialement ;
— on suppose que c'est `gcc` qui démarre le premier ;
— on suppose que `bash` se bloque au milieu de sa première tranche de temps, et qu'il redevient prêt durant la troisième tranche de temps consommée par `gcc` sur le CPU.

Précisez bien le nombre de crédits que possède chaque processus à tout moment.

2 Nachos

Voici comment sont implantés les *sémaphores* dans Nachos :

```
void Semaphore::P ()
{
  IntStatus oldLevel = interrupt->SetLevel (IntOff);

  while (value == 0) {
    // semaphore not available, go to sleep
    queue->Append ((void *) currentThread);
    currentThread->Sleep ();
  }
  value--; // semaphore available, consume its value

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

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

  Thread *thread = (Thread *) queue->Remove ();

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

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

La fonction `Thread::Sleep` bloque le thread appelant et effectue un changement de contexte vers un autre thread prêt. La fonction `Scheduler::ReadyToRun` place le thread dans l'état prêt.

Question 1 On peut remarquer la présence d'une boucle **`while`** dans la méthode `Semaphore::P()`. Pour quelle raison n'utilise-t-on pas un simple test **`if`** ? Expliquez précisément.

Question 2 Lors de l'appel `currentThread->Sleep()`, on remarque que les interruptions sont désactivées au moment de l'appel. À quel moment les interruptions vont-elles être ré-activées (on ne demande pas de mentionner un endroit précis dans le code de Nachos) ? Par qui ?

Question 3 En vous inspirant de cette implémentation des sémaphores, donnez une implémentation des moniteurs de Hoare, c'est-à-dire de la classe `Mutex` esquissée ci-dessous. Il s'agit de donner le corps des fonctions `Lock` et `Unlock`. On rappelle qu'un thread ne peut pas relâcher un verrou qu'il ne détient pas.

```
class Mutex
{
  public:
    void Lock ();
    void Unlock ();
 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 4 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 (Mutex *m);
    void Signal ();
  private:
    List *queue;        // threads waiting on condition
};
```

3 Barrières de synchronisation en deux temps

On s'intéresse aux barrières de synchronisation « 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 les processus risquent 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 Même question en utilisant les **moniteurs de Hoare** (voir memento ci-dessous).

Memento (pour exercice 3)

```
typedef ... mutex_t;
void mutex_lock(mutex_t *m);
void mutex_unlock(mutex_t *m);
```

```
typedef ... cond_t;
void cond_wait(cond_t *c, mutex_t *m);
void cond_signal(cond_t *c);
void cond_bcast(cond_t *c);
```