université de BORDEAUX

Collège Sciences et technologies

Année	2020-2021	**Type**	Devoir Surveillé
Master	Informatique		
Code UE	4TIN705U	**Épreuve**	Systèmes d'Exploitation
Date	20/11/2020	**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 ? Donnez quelques avantages et inconvénients de cette technique.

Question 2 Expliquez en quoi consiste une opération de « changement de contexte » entre threads au sein d'un système d'exploitation. Y a-t-il une différence de traitement suivant que les threads appartiennent (ou non) au même processus ?

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 (`emacs` et `firefox`) :

— la priorité statique d'`emacs` lui donne droit à 2 crédits initiaux, alors que `firefox` a droit à 3 crédits au départ ;
— on suppose qu'`emacs` se bloque au milieu de sa première tranche de temps, et qu'il redevient prêt durant la quatrième tranche de temps de `firefox` 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);
}
```

Question 1 À quoi servent les appels `interrupt->SetLevel(IntOff)` ? Que garantissent-il au thread appelant ?

Question 2 On peut remarquer la présence curieuse d'une boucle dans la méthode `Semaphore::P()`. Pour quelle raison n'utilise-t-on pas un simple test **`if`** au lieu d'une boucle **`while`** ? Expliquez précisément.

Question 3 La fonction `Thread::Sleep()` bloque le thread appelant et effectue un changement de contexte vers un autre thread prêt. On remarque que les interruptions sont désactivées au moment de l'appel. À quel moment les interruptions vont-elles pouvoir être ré-activées (on ne demande pas de mentionner un endroit précis dans le code de Nachos) ? Par qui ?

3 Synchronisation

En temps normal, les étudiants des parcours Info/CSI/CMI/IDI auraient probablement projeté d'aller au cinéma ce soir pour oublier cette journée d'examen éprouvante. La simulation informatique de cette sortie met en jeu un processus par étudiant qui exécute une séquence que nous détaillerons progressivement :

```
void etudiant ()
{
  rendez_vous ();              // Rendez-vous devant le cinéma

  int ma_place = acheter_ticket (); // Faire la queue

  ... // Profiter du film, ou rentrer chez soi si ma_place == -1 :(
}
```

Dans cet exercice, nous utiliserons les **moniteurs de Hoare** pour synchroniser les processus (les types et fonctions associées sont rappelés en fin de sujet).

Question 1 Les étudiants se donnent rendez-vous devant le cinéma. Mais comme ils ne savent pas exactement combien ils seront, ils décident d'attendre jusqu'à ce qu'ils soient 50.

Donnez le code de la fonction `rendez_vous`, qui doit bloquer les 49 premiers processus l'appelant jusqu'à l'arrivée du 50^e, et ne plus bloquer les suivants (ils arrivent en retard au rendez-vous mais il est encore temps d'acheter un ticket). Indiquez clairement les variables globales dont avez besoin.

Question 2 La seconde étape consiste à acheter un ticket en appelant la fonction `acheter_ticket ()`. Notons que cette fonction peut également être appelée par d'autres processus[1] que les étudiants, dont on ne connaît pas le nombre. Voici le code initial de cette fonction :

```
#define CAPACITE_MAX ...
int place = 0;

int acheter_ticket ()
{
  if (place == CAPACITE_MAX)
    return -1;

  place ++;

  sleep (DELAI_PAIEMENT);

  return place;
}
```

Donnez le nouveau code de `acheter_ticket`, de manière à ce que les processus obtiennent leur ticket tour à tour. Notez qu'une file d'attente implicite va se créer, en raison du temps de paiement de chaque personne. On souhaite que si le nombre de personne dans la file (i.e. bloqués à l'intérieur de la fonction `acheter_ticket`) atteint une constante `MAX_FILE`, cela devienne dissuasif pour les personnes qui arrivent à ce moment-là : elles font demi-tour immédiatement (i.e. `acheter_ticket` renvoie -1 *sans délai*).

Question 3 Le cinéma possède en fait plusieurs guichets auprès desquels les clients achètent leur ticket, une fois la file d'attente passée. La constante `NB_GUICHETS` indique le nombre de guichets ouverts. Donnez une nouvelle version de la fonction `acheter_ticket` dans laquelle il peut y avoir jusqu'à « `NB_GUICHETS` » personnes qui achètent simultanément leur place (i.e. les processus correspondants exécutent `sleep(DELAI_PAIEMENT` en parallèle).

Memento

```
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);
```

1. des gens normaux quoi