université de BORDEAUX

Collège Sciences et technologies

Année	2020-2021	**Type**	**Entrainement**
Master	Informatique		
Code UE	4TIN705U	**Épreuve**	Systèmes d'exploitation
Date	13/11/2020		
Début	10h15	**Durée**	1h30

1 Questions de cours (échauffement)

Question 1 Dans un système d'exploitation interactif tel qu'Unix, rappelez précisément pourquoi un processus exécutant une boucle infinie ne monopolise pas pour autant le processeur s'il existe d'autres processus prêts dans le système.

Question 2 Dans le simulateur Nachos, l'option `-rs` permet de simuler le comportement d'un système interactif. Expliquez le mécanisme utilisé par Nachos dans ce cas. Quelles en sont les limites, par rapport à un véritable système d'exploitation ?

2 Le tram et les voitures

On souhaite simuler une version « moderne » d'un croisement entre une ligne de tramway et une route. La simulation met en jeu deux types de processus : `croisement` et `voiture`. Il n'y a qu'un seul processus `croisement`, qui automatise le fonctionnement des feux rouges clignotants et qui simule le passage du tram à certains moments. Il est lancé en tout début de simulation. Les autres processus (peu importe le nombre) sont des voitures et sont lancés à des instants aléatoires. Tous les processus ont accès à une zone commune de mémoire partagée, qui contient les variables suivantes :

```
boolean rouge_clignotant = FALSE;
int voitures_engagées = 0;  /* nombre de voitures en train de traverser */
```

Voici le code du processus `croisement` :

```
while(1) {
  sleep (PERIODE_SANS_TRAM);
  rouge_clignotant = TRUE;
  while (voitures_engagées > 0) /* rien */ ;
  sleep (TEMPS_PASSAGE_DU_TRAM);
  rouge_clignotant = FALSE;
}
```

Et voici le code exécuté par les processus `voiture` :

```
while(rouge_clignotant) /* rien */ ;
voitures_engagées++;
sleep(TEMPS_TRAVERSEE);
voitures_engagées--;
```

En résumé, une voiture ne s'engage que lorsque le feu ne clignote pas, et lorsque ce dernier se met à clignoter, le croisement attend patiemment que les dernières voitures aient franchi le croisement avant de laisser passer le tramway.

Question 1 Cette simulation va-t-elle solliciter intensivement le ou les processeur(s) de la machine sur laquelle elle va s'exécuter ? Expliquez pourquoi.

Question 2 Montrez que la programmation de la simulation telle que décrite ci-dessus risque d'aboutir à des accidents, c'est-à-dire à des situations où des voitures s'engagent en même temps que le tram... Donnez deux raisons distinctes pour lesquelles un accident peut se produire.

Question 3 Corrigez donc le protocole en introduisant des moniteurs de Hoare (et peut-être d'autres variables) partagés et en modifiant les programmes. Profitez de l'occasion pour éviter l'utilisation des boucles d'attente active. N'oubliez pas de préciser les valeurs initiales. Voici pour rappel les primitives que vous pouvez utiliser :

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

3 Histoire de groupes

Dans cet exercice, il s'agit d'utiliser des *sémaphores* pour synchroniser une fonction `thread_func` qui nécessite des précautions particulières lorsqu'elle est exécutée de manière simultanée par plusieurs processus légers. Le nombre de processus légers du programme n'est pas défini a priori. Toutefois, on sait que chaque processus léger appartient à un groupe (et un seul), les groupes étant numérotés de 0 à $N-1$ (N est une constante du problème). Lorsqu'un processus léger appelle la fonction `thread_func`, il lui passe le numéro du groupe auquel il appartient en paramètre (i.e. `grp`).

```
#define N ??

... // déclaration des sémaphores & variables globales

void thread_func(unsigned grp)
{
   ... // synchronisation à ajouter

   do_compute(grp);

   ... // synchronisation à ajouter
}
```

Les contraintes de synchronisation sont dictées par l'implémentation de la fonction `do_compute` **qui ne peut être exécutée en parallèle que par des threads appartenant au même groupe**. Autrement dit : Un thread du groupe i ne peut démarrer l'exécution de `do_compute(i)` que lorsqu'aucun thread d'un groupe j ($j \neq i$) n'est en train de l'exécuter. Le code de synchronisation doit par conséquent permettre l'exécution concurrente de `do_compute` par plusieurs threads du même groupe lorsque c'est possible. Il faut bien entendu bloquer les processus lorsque la situation ne leur permet pas d'exécuter `do_compute`.

Question 1 En remarquant que ce problème présente certaines (voire beaucoup de) similitudes avec le problème des lecteurs/rédacteurs vu en cours, proposez une solution **la plus simple possible** permettant de répondre aux contraintes posées.

Par exemple, un bon point de départ serait d'utiliser un tableau d'entiers `unsigned nb[N]` qui permettrait de compter, pour chaque groupe, le nombre de threads en train d'exécuter la fonction `do_compute`. De cette manière, il est aisé de faire faire un traitement spécial au premier thread arrivant/dernier thread sortant...

Donnez le code de synchronisation entourant l'appel à `do_compute`. N'oubliez pas de préciser les valeurs initiales des variables et sémaphores que vous allez déclarer.

Question 2 Il s'agit maintenant d'étendre cette solution pour la rendre *équitable*, c'est-à-dire qu'il va falloir prendre en compte l'ordre d'arrivée des threads dans la fonction `thread_func` pour régir l'accès à la fonction `do_compute`.

En supposant que les sémaphores gèrent les processus bloqués dans un ordre FIFO, il est possible d'utiliser un sémaphore additionnel qui jouerait le rôle de "compte-goutte", en empêchant les threads de se "doubler" à cause de la synchronisation mise en place à la question précédente.

Donnez une nouvelle version du code de `thread_func` qui préserve l'équité de progression entre les différents threads (indiquer aussi les nouvelles déclarations nécessaires).

Question 3 On souhaite maintenant borner le nombre de threads exécutant simultanément la fonction `do_compute` par une constante `MAX` (i.e. au plus `MAX` threads peuvent exécuter `do_compute` en parallèle).

Indiquez ce qu'il faut modifier dans le code précédent pour respecter cette contrainte supplémentaire.