université de BORDEAUX

Collège Sciences et technologies

Année	2022-2023	Type	Devoir Surveillé
Master	Informatique		
Code UE	4TIN705U	Épreuve	Systèmes d'Exploitation
Date	18/11/2022	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 Définissez brièvement ce que l'on appelle "ordonnanceur" dans un système d'exploitation. Quand et par qui le code d'ordonnancement est-il exécuté (listez des situations bien précises) ? Dans un système Unix, quels sont les principaux paramètres pris en compte pour le choix du prochain processus à exécuter ?

Question 2 Dans les noyaux Linux 2.4.x, l'ordonnancement des processus s'appuie sur un système de *crédits* qui sont débités à l'issue de chaque quantum de temps utilisé. Est-ce que ce système garantit une alternance d'exécution entre deux processus de même priorité qui utilisent tous deux le processeur de manière intensive (i.e. sans jamais effectuer d'appel système) ? Expliquez.

2 Synchronisation

On souhaite pouvoir lancer des traitements en « tâches de fond » dans un programme, c'est-à-dire déléguer à un ou plusieurs *threads* l'exécution de fonctions pendant que le programme principal continue son exécution. On appelle *tâche* un couple (fonction, argument) dont l'exécution est ainsi effectuée de manière asynchrone.

Voici un exemple de programme que l'on souhaiterait pouvoir exécuter :

```
1  void f (void *arg)
2  { ... }
3
4  void g (void *arg)
5  { ... }
6
7  int main ()
8  {
9    tasks_init ();
10
11   tasks_submit (f, "task 1");
12   tasks_submit (g, "task 2");
13   tasks_submit (f, "task 3");
14
15   // Do some work while tasks are running in background...
16 }
```

L'appel `tasks_init` (ligne 9) initialise une file de tâches d'une capacité finie, et crée un pool de threads prêts à extraire des tâches de la file dès qu'il y en aura. Chaque appel à `tasks_submit` (lignes 11–13) dépose une nouvelle tâche dans la file. L'opération est bloquante si la file est pleine.

Si le pool se limite à un seul thread, alors les tâches seront exécutées les unes après les autres. Dans le cas d'un pool de plusieurs threads, certaines tâches pourront être exécutées en parallèle (dans la limite du nombre de threads disponibles).

Une implémentation préliminaire vous est fournie ci-dessous. Pour l'instant, on suppose que le programme principal se termine systématiquement (miraculeusement même) après que toutes les tâches aient été exécutées.

```c
typedef struct {
  func_t f;
  void *arg;
} task_t;

// Implementation details of queue are omitted
#define MAX_QUEUE ...
int __push (const task_t *t) { ... }
int __pop  (task_t *t) { ... }
unsigned __size (void) { ... }

int push (const task_t *t)
{
  return __push (t);
}

int pop (task_t *t)
{
  return __pop (t);
}
```

```c
void pop_and_execute (void)
{
  task_t t;

  if (pop (&t) == 0)
    t.f (t.arg);
}

void thread_body (void)
{
  // each thread executes an infinite loop
  for (;;)
    pop_and_execute ();
}

void tasks_submit (func_t f, void *arg)
{
  task_t t = { f, arg };
  while (push (&t) == -1)
    /* nothing */ ;
}
```

Cette implémentation s'appuie sur une file de tâches accessible au travers des primitives `__push`, `__pop` et `__size`. L'implémentation importe peu [1], on sait juste que la file a une capacité maximale égale à `MAX_QUEUE`, et qu'aucune précaution particulière n'a été prise pour que ces primitives fonctionnent lorsqu'elles sont appelées de manière concurrente. Si la file est pleine, `__push` échoue et renvoie -1. De même, si la file est vide, `__pop` échoue et renvoie -1.

On ne s'intéressera pas au code qui crée les threads dans `task_init`. On sait juste que tous les threads créés exécutent la fonction `thread_body`.

Question 1 En l'absence de garantie sur l'implémentation des fonctions `__push`, `__pop` et `__size`, a-t-on tout de même l'assurance que le programme fonctionne correctement lorsqu'un seul thread est créé par `task_init` ? Expliquez. Par ailleurs, le code risque-t-il de solliciter maladroitement les processeurs de la machine ? Pourquoi ?

Question 2 En utilisant les outils associés aux moniteurs de Hoare (cf memento en fin de sujet), ajoutez la synchronisation nécessaire aux primitives `push` et `pop` de sorte que

— `task_submit` soit bloquante lorsque la file est pleine ;

— `pop_and_execute` soit bloquante lorsque la file est vide.

Indiquez quelles sont les variables globales ajoutées, et précisez à quel moment elles doivent être initialisées.

Question 3 On souhaite maintenant disposer d'une fonction **void** `tasks_wait_all` (**void**) qui sera appelée par le programme principal pour attendre que toutes les tâches soumises soient exécutées. Donnez le code de cette fonction, et indiquez quelles sont les modifications éventuelles à apporter aux autres fonctions. Attention à bien attendre qu'il n'y ait plus de tâche en cours (il ne s'agit pas seulement d'attendre qu'il n'y ait plus rien dans la file).

Question 4 On décide maintenant de distinguer deux types de tâches : les tâches *normales*, et les tâches *exclusives*. Ces dernières sont uniquement exclusives entre elles : il ne doit jamais y avoir deux tâches exclusives exécutées en même temps. L'exécution des tâches normales n'obéit à aucune contrainte de ce type.

Pour ce faire, on ajoute un paramètre supplémentaire à `task_submit` ainsi qu'un champ à la structure `task_t` :

```c
typedef enum { REGULAR, EXCLUSIVE } task_type_t;

typedef struct {
  func_t f;
  void *arg;
  task_type_t type;
} task_t;

void tasks_submit (func_t f, void *arg, task_type_t type);
```

1. On n'essaiera pas de modifier ces fonctions.

Voici un exemple de programme principal :

```
int main ()
{
  tasks_init (nbthreads);

  tasks_submit (f, "task 1", EXCLUSIVE);
  tasks_submit (f, "task 2", EXCLUSIVE);
  tasks_submit (f, "task 3", REGULAR);

  tasks_wait_all ();
}
```

En supposant une file de capacité au moins égale à 3, voici quelques scenarii d'exécution :

— Avec un pool d'un seul thread, les 3 tâches sont exécutées séquentiellement ;

— Avec un pool de deux threads, la tâche 1 est exécutée d'abord, et une fois terminée les tâches 2 et 3 peuvent être exécutées en parallèle ;

— Avec un pool de trois threads ou plus, la tâche 1 et la tâche 3 peuvent s'exécuter en parallèle, et une fois la tâche 1 terminée, la tâche 2 pourra démarrer.

Proposez une solution simple et indiquez ce qu'il faut modifier dans les fonctions du code pour assurer la synchronisation demandée. Attention, la synchronisation ajoutée ne doit pas empêcher l'accès à la file pendant qu'une tâche exclusive est en cours...

Question 5 On souhaite mettre en place une solution plus sophistiquée au problème posé à la question précédente. En effet, dans l'exemple présenté en question 4, et dans le cas où le pool ne serait constitué que de deux threads, on voudrait que les tâches 1 et 3 s'exécutent en parallèle dès le début.

Autrement dit, le thread qui récupère la tâche 2 devrait pouvoir s'apercevoir qu'il y a déjà une tâche exclusive en cours et qu'il est dommage d'attendre alors qu'il existe au moins une tâche normale plus loin dans la file. L'idée serait alors que le thread remette dans la file les tâches exclusives qu'il récupère jusqu'à ce qu'il récupère une tâche normale.

Donnez le code de cette dernière version.

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);
```