Licence Mention Informatique, 3ème année

Programmation des systèmes
Travaux dirigés – Système de fichiers et threads

L'équipe pédagogique

Introduction

Ce document [1] est le support de travaux dirigés pour le cours de Programmation de Système, au 3ème année de Licence Informatique à l'Université de Lille. Le document a été crée par Philippe Marquet : il a évolué au cours des années avec des contributions par toute les équipes pédagogiques successives [2].

1 TD 1 : Implantation d'une bibliothèque d'entrées/sorties tamporisées

On propose ici une implantation d'une version très rudimentaire d'une bibliothèque d'entrées/-sorties tamporisées. Seuls les équivalents des primitives `fopen()`, `fclose()`, `fgetc()`, et `fputc()` seront implantés. Pour se distinguer des versions standard, on utilisera le préfixe « k ».

La gestion d'un tampon utilisateur nécessite de mémoriser un certain nombre d'informations. On pourra penser notamment à la position courante dans le tampon, l'index du tampon dans le fichier, le

1. Énoncé placé sous licence Creative Commons by-nc-sa (`https://creativecommons.org/licenses/by-nc-sa/3.0/deed.fr`).
2. Philippe Marquet, Gilles Grimaud, Samuel Hym, Giuseppe Lipari, et beaucoup d'autres.

nombre de caractères valides dans le tampon, un indicateur signalant que le contenu du tampon a été modifié, et une référence au fichier sous la forme du descripteur associé aux appels système. On définira donc une structure de données, `KFILE`, pour enregistrer toutes les données associées à un fichier ouvert.

Question 1 *Écrire un caractère* Proposez une fonction `int kputc(int ch, KFILE* kfile)` prenant en argument un caractère et un pointeur vers une structure `KFILE` et écrivant ce caractère dans le flot correspondant. Cette fonction retournant le caractère écrit dans le flot si tout s'est bien passé et `EOF` sinon. L'idée essentielle de cette fonction est de ne provoquer un appel système `write` que si le tampon mémoire du `KFILE` est rempli.

Vous définirez en même temps la structure `KFILE` en fonction des besoins pour écrire la fonction `kputc()`. ■

Question 2 *Ouvrir et fermer un flot* Proposez des fonctions :

— `KFILE* kopen(const char *path)` qui ouvre un `KFILE`,

— `int kclose(KFILE* kfile)` qui ferme un `KFILE`.

Vous pourrez profiter de l'écriture de ces fonctions pour définir des fonctions auxiliaires pertinentes. ■

Question 3 *Lire un caractère* Proposez une fonction `int kgetc(KFILE* kfile)` qui lit un caractère dans le flot et le retourne, ou retourne `EOF` si la fin du fichier est atteinte. Vous pourrez étendre la structure `KFILE` et modifier les fonctions précédentes si nécessaire. ■

2 TD 2 : Retrouver une commande

La commande Unix `which` recherche les commandes dans les chemins de recherche de l'utilisateur.

La variable d'environnement `$PATH` contient une liste de répertoires séparés par des « : ». Lors de l'invocation d'une commande depuis le shell, un fichier exécutable du nom de la commande est recherchée dans l'ordre dans ces répertoires. C'est ce fichier qui est invoqué pour l'exécution de la commande. Si aucun fichier ne peut être trouvé, le shell le signale par un message.

La commande `which` affiche le chemin du fichier qui serait trouvé par le shell lors de l'invocation de la commande :

```
$ echo $PATH
/bin:/usr/local/bin:/usr/X11R6/bin:/home/alice/bin
$ which ls
/bin/ls
$ echo $?
0
$ which foo
foo: Command not found.
$ echo $?
1
```

Question 1 *Liste des répertoires de recherche* Dans un premier temps, proposez une fonction `filldirs()` qui remplit un tableau `dirs` contenant la liste des noms des répertoires de `$PATH`. Ce tableau sera terminé par un pointeur `NULL`. ■

Question 2 *Une fonction `which()`* Écrivez maintenant une fonction `which()` qui affiche le chemin absolu correspondant à la commande dont le nom est passé en paramètre ou un message d'erreur si la commande ne peut être trouvée dans les répertoires de recherche de `$PATH`. Cette fonction retourne une valeur booléenne indiquant un succès ou un échec de la recherche.

On pourra utiliser l'appel système

```
#include <unistd.h>
int access(const char *path, int mode);
```

qui vérifie les permissions fournies sous la forme d'un *ou* des valeurs `R_OK` (*read*, lecture), `W_OK` (*write*, écriture), `X_OK` (*execution*, exécution), `F_OK` (existence). ■

Question 3 *La commande `which`* Terminez par écrire votre propre version de la commande `which` qui se termine par un succès si et seulement si toutes les commandes données en paramètre ont été trouvées dans les répertoires de recherche désignés par `$PATH`. ■

3 TD 3 : Calcul parallèle

3.1 Nombres premiers

La fonction suivante retourne `1` si le paramètre `n` en est un nombre premier.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10
#define M 1000

int is_prime(int n)
{
    if (n <=0) return 0;
    for (int i=2; i<n/2; i++)
        if (n %i == 0) return 0;
    return 1;
}

int main(int argc, char *argv[])
{
    int array[N];
    int i;
    int count = 0;

    /* generate N random values between 0 and M */
    srand(time(NULL));
    for (i = 0; i<N; i++)
        array[i] = ((int)rand()) % M;

    /* Sequential search of prime numbers */
    for (i = 0; i<N; i++)
        if (is_prime(array[i])) {
            printf("%d is prime \n", array[i]);
            count++;
        }
    printf("I found %d prime numbers\n", count);

    return 0;

}
```

Nous voulons paralléliser la boucle `for` à la ligne 28 ; on lance des *threads* qui invoquent la fonction `is_prime()` en parallèle.

Question 1 *Adaptation* Écrire une fonction `is_prime_wrapper()` qui constitue le corps d'un thread, et qui appellera la fonction `is_prime()`. Comment passer le paramètre `n` ? Comment re-

tourner le résultat de la fonction ? ■

Question 2 *Utilisation d'une variable globale* Le retour du résultat peut être évité si les threads modifient directement une variable globale `count`. Quel sont les problèmes d'un tel approche ? ■

Question 3 *Parallélisation* Modifier le main pour :
- créer un *thread* pour chaque itération de la boucle ;
- attendre la terminaison de tous les *threads*, et calculer le résultat final.

■

La fonction `is_prime()` est très simple et son temps d'exécution très court. En revanche, la création d'un thread, et la synchronisation sur sa terminaison sont des opérations beaucoup plus lourdes. En plus, le programme crée beaucoup de threads : la stratégie de parallélisation envisagée n'est pas efficace.

Question 4 *Stratégies de parallélisation* Décrire une stratégie plus efficace pour paralléliser le programme ci-dessous, en particulier quand la taille du tableau devient très importante. ■

3.2 Recherche parallèle

On désire réaliser une implantation multithreadée de la recherche d'une valeur `v` dans un tableau de valeurs selon la méthode suivante :
- on délègue la recherche de la valeur dans la première moitié du tableau à un thread ;
- on recherche la valeur dans la seconde moitié du tableau (en utilisant la même méthode de recherche parallèle) ;
- on récupère le résultat de la recherche effectuée par le thread pour le combiner au résultat de notre recherche.

Soit la fonction

```
int search(int *tab, unsigned int size, int v);
```

de recherche de la valeur entière `v` dans le tableau `tab` de `size` éléments. Cette fonction retourne l'indice du tableau contenant `v` ou `-1` si `v` n'apparaît pas dans `tab`.

Question 5 *Adaptation de la fonction `search()`* Donnez l'implantation d'une fonction `search_wrapper()` ayant un prototype de fonction principale d'un thread en vous appuyant sur l'exemple `adaptation` de l'exercice précédent. Cette fonction `search_wrapper()` réalise la recherche par un simple appel à `search()`. ■

Question 6 *Recherche parallèle* Donnez une implantation de `search()`.
Attention : pour éviter les problèmes d'efficacité vus dans l'exercice 3, éviter l'appelle récursive quand la taille `size` est inférieure à un seuil prédéfini. ■

4 TD 4 : barrière de synchronisation

Soient les deux activités `p1` et `p2` suivantes. Elles se partagent deux sémaphores, `sem1` et `sem2` initialisés à 0.

```
sem_t sem1, sem2;

sem_init(&sem1, 1, 0);
sem_init(&sem2, 1, 0);
```

```
p1() {
    a1();
    sem_post(&sem2);
    sem_wait(&sem1);
    b1();
}
```

```
p2() {
    a2();
    sem_post(&sem1);
    sem_wait(&sem2);
    b2();
}
```

Question 1 *Synchronisation* Quelle synchronisation a-t-on imposée sur les exécutions des `a1()`, `a2()`,`b1()` et `b2()` ? ■

Question 2 *Synchronisation à N* Donner le code qui exige la même synchronisation pour N processus en utilisant N sémaphores. ■

On cherche maintenant à implémenter une solution générique pour une barrière à N threads en utilisant seulement 2 sémaphores et un compteur. L'idée est la suivante :

— La variable compteur, initialisé à zéro, compte combien de threads sont arrivée à la barrière ; chaque thread va d'abord incrémenter ce compteur ;
— Si le compteur est inférieur à N, il signifie que le thread n'est pas le dernier, il doit se bloquer dans l'attente du dernier thread ;
— Le dernier thread débloque tous les autres.

Question 3 À quoi sert les deux sémaphores ? ■

Pour mettre en oeuvre la stratégie décrite ci-dessus de manière générale :

— On déclare une structure de données `struct barrier` qui contient les données nécessaires à implémenter la barrière ;
— On implémente une fonction `barrier_init(struct barrier *b, int n)` pour initialiser la structure de données (`n` est le nombre de threads qui se synchronise sur la barrière) ;
— On implémente une fonction `barrier_synch(struct barrier *b);` qui sera appelée par les threads et qui réalise la synchronisation.

Question 4 *Barrière avec 2 sémaphores* Implémenter la barrière. ■

5 TD 5 : Producteur / consommateur

Dans cet exercice, on analyse un problème classique de communication entre threads.

On considère un programme parallèle qui découpe un calcul compliqué et récurrent en deux étapes. La structure de ce programme est montrée dans la figure 1.

```
producer() {
    while (true) {
        /* produit une donnée */
        mailbox_put(data);
    }
}
```

```
consumer() {
    while (true) {
        mailbox_get(&data);
        /* élaboration de la donnée */
    }
}
```

FIGURE 1 – Structure producteur consommateur

Un thread (dit *producteur*) fait la première partie du calcul et mémorise les résultats intermédiaires à fur et mesure qui sont générés dans une structure de données tampon (appelée *mailbox*) ; un deuxième thread (appelé *consommateur*) récupère ces donnés intermédiaires avant de faire la deuxième partie du calcul. Il s'agit d'une *pipeline* d'exécution : pendant que le thread *producteur* produit l'n-ème donnée, le consommateur peut travailler sur les données produites précédemment.

Le problème qu'on essaye de résoudre est la synchronisation des données entre le producteur et le consommateur. Nous ne connaissons pas la vitesse à laquelle les données sont produites par le producteur, ni la vitesse à laquelle elles sont consommées par le deuxième thread. Il peut arriver que le consommateur essaye de récupérer les données avant qu'elles soient produites. De plus, comme la taille du tampon est limitée, il peut arriver que le producteur produit ses données trop rapidement en remplissant complètement le tampon avant que le consommateur puisse les récupérer. Il faut alors bloquer le producteur quand il n'y a plus de place dans le tampon.

Pour régler ces problèmes, nous allons implémenter la structure de données *mailbox* et les fonctions pour mémoriser et récupérer les données en respectant le consignes suivantes :

— Le thread consommateur doit se bloquer s'il essaye de récupérer des données d'une mailbox vide (producteur trop lent) ;

— Le thread producteur doit se bloquer s'il essaye d'écrire des données dans une mailbox pleine (consommateur trop lent) ;
— Aucune donnée doit être perdue ; toutes les données produites par le producteur sont éventuellement récupérées par le consommateur.

D'abord, nous considérons que le producteur produit des entiers et que la mailbox contient une seule place disponible.

Question 1 *Structure de donnée* Proposer une structure de donnée `struct mailbox` pour représenter le tampon.

Ensuite, écrire les code de la fonction :

```
void mbox_init(struct mailbox *m);
```

qui initialise la structure pointée par `m`. ■

Question 2 *Put et get* Écrire le code des fonctions pour mettre et retirer une donnée de la mailbox :

```
void mbox_put(struct mailbox *m, int d);
int  mbox_get(struct mailbox *m);
```

■

Question 3 *Producteur/Consommateur avec N places* Pour augmenter le parallélisme, on considère maintenant une mailbox avec `N` places disponibles. Généraliser le code précèdent. ■

Question 4 *Plusieurs producteurs et consommateurs* Finalement, on considère le cas ou il y a plusieurs producteurs et plusieurs consommateurs qui accèdent à la même mailbox. Coder les fonctions pour garantir la synchronisation et la correction des données. ■