UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Master M1 Informatique, CSE
Année 2020-2021

Résumé

Ce TD/TP vise à améliorer votre compréhension des problèmes de synchronisation et de leurs solutions. Vous y aurez l'occasion d'utiliser une partie des primitives fournies par la bibliothèque de *threads* POSIX, en particulier :

- `pthread_mutex_init()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`
- `pthread_cond_init()`
- `pthread_cond_destroy()`
- `pthread_cond_wait()`
- `pthread_cond_broadcast()`
- `pthread_cond_signal()`

Vous aurez probablement besoin de consulter les pages de manuel associées (par exemple `man 3 pthread_mutex_lock`) afin de prendre connaissance de leur sémantique précise (Remarque : il y a beaucoup d'autres fonctions, et leur page de manuel associée, dans la bibliothèque `pthread`. Pour en obtenir une liste complète, exécutez : `man -k pthread`).

Lecteurs-Rédacteurs

Il s'agit d'accès concurrents à une ressource partagée par deux types d'entités : les lecteurs et les rédacteurs. Les lecteurs accèdent à la ressource sans la modifier. Les rédacteurs, eux, modifient la ressource. Pour garantir un état cohérent de la ressource, plusieurs lecteurs peuvent y accéder en même temps mais l'accès pour les rédacteurs est en accès exclusif. En d'autres termes, si un rédacteur travaille avec la ressource, aucune autre entité (lecteur ou rédacteur) ne doit pouvoir accéder à celle-ci. Le problème des lecteurs-rédacteurs est un problème classique de synchronisation lors de l'utilisation d'une ressource partagée. Ce schéma est typiquement utilisé pour la manipulation de fichiers ou de zones mémoire.

Dans la suite, nous allons considérer les fonctions suivantes :

- `initialiser_lecteur_redacteur(lecteur_redacteur *)` est la fonction d'initialisation des structures de synchronisation ;
- `detruire_lecteur_redacteur(lecteur_redacteur *)` est la fonction de destruction des structures de synchronisation ;
- `debut_lecture(lecteur_redacteur *)` et `fin_lecture(lecteur_redacteur *)` sont les primitives exécutées par chaque lecteur avant et après une lecture ;
- `debut_redaction(lecteur_redacteur *)` et `fin_redaction(lecteur_redacteur *)` sont les primitives exécutées par chaque rédacteur avant et après une écriture.

Le type `lecteur_redacteur` est à définir selon vos besoins. Il s'agira probablement d'une structure contenant les variables et les structures nécessaires pour assurer une synchronisation correcte.

Questions :

1. Etudiez le programme `test_lecteurs_redacteurs.c` qui simule à l'aide de thread un ensemble de lecteurs et de rédacteurs accédant à une ressources commune. Ce programme utilise les fonctions de synchronisation définies pécédemment. D'après vous, ce programme permet-il d'identifier à coup sûr toutes les situations de synchronisation incorrecte ? Justifiez votre réponse.
2. Le but est maintenant de proposer une implémentation pour les fonctions de synchronisation des lecteurs et des rédacteurs. Commenter la solution suivante, en quoi n'est-elle pas satisfaisante ?

```
Mutex LR;

initialiser_lecteur_redacteur(Mutex M) { initialiser(M): }

debut_lecture(Mutex M) {lock(M);}
fin_lecture(Mutex M) {unlock(M);}

debut_ecriture(Mutex M) {lock(M);}
fin_ecriture(Mutex M) {unlock(M);}
```

3. Proposer une solution qui corrige la solution précédente. Que pouvez vous dire sur l'équité de cette solution ?
4. Proposez des solutions pour chacune des différentes politiques de gestion des priorités entre lecteurs et rédacteurs :
   — lecteurs prioritaires
   — redacteurs prioritaires
   — priorité selon l'ordre partiel induit par l'ordre d'arrivée : les threads passent selon leur ordre d'arrivée en permettant tout de même l'accès en parallèle à plusieurs lecteurs. Par exemple, l'ordre d'arrivée suivant : L1, L2, L3, R1, R2, L4, L5, R3 donnera l'ordre de passage suivant :

```
L1,L2,L3 en parallèle
R1 seul
R2 seul
L4,L5 en parallèle
R3 seul
```

ANNEXE A : test_lecteurs_redacteurs.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "lecteur_redacteur.h"

typedef struct {
    lecteur_redacteur_t lecteur_redacteur;
    int iterations;
    int donnee;
} donnees_thread_t;

void dodo(int scale) {
    usleep((random()%1000000)*scale);
}

void *lecteur(void *args) {
    donnees_thread_t *d = args;
    int i, valeur;
    srandom((int) pthread_self());

    for (i=0; i < d->iterations; i++) {
        dodo(2);
debut_lecture(&d->lecteur_redacteur);
         printf("Thread %x : debut lecture\n", (int) pthread_self());
         valeur = d->donnee;
         dodo(1);
         printf("Thread %x : ", (int) pthread_self());
         if (valeur != d->donnee)
             printf("LECTURE INCOHERENTE !!!\n");
         else
             printf("lecture coherente\n");
        fin_lecture(&d->lecteur_redacteur);
    }
    pthread_exit(0);
}

void *redacteur(void *args) {
    donnees_thread_t *d = args;
    int i, valeur;
    srandom((int) pthread_self());

    for (i=0; i < d->iterations; i++) {
        dodo(2);
        debut_redaction(&d->lecteur_redacteur);
         printf("Thread %x : debut redaction......\n", (int) pthread_self());
         valeur = random();
         d->donnee = valeur;
```

```
            dodo(1);
            printf("Thread %x : ", (int) pthread_self());
            if (valeur != d->donnee)
                printf("REDACTION INCOHERENTE !!!\n");
            else
                printf("redaction coherente......\n");
            fin_redaction(&d->lecteur_redacteur);
        }
        pthread_exit(0);
    }



    int main(int argc, char *argv[]) {
        pthread_t *threads, *thread_courant;
        donnees_thread_t donnees_thread;
        int i, nb_lecteurs, nb_redacteurs;
        void *resultat;

        if (argc < 4) {
            fprintf(stderr, "Utilisation: %s nb_lecteurs nb_redacteurs "
                            "nb_iterations\n", argv[0]);
            exit(1);
        }

        nb_lecteurs = atoi(argv[1]);
        nb_redacteurs = atoi(argv[2]);
        donnees_thread.iterations = atoi(argv[3]);

        threads = malloc((nb_lecteurs+nb_redacteurs)*sizeof(pthread_t));
        thread_courant = threads;
        initialiser_lecteur_redacteur(&donnees_thread.lecteur_redacteur);

        for (i=0; i<nb_lecteurs; i++)
            pthread_create(thread_courant++, NULL, lecteur, &donnees_thread);
        for (i=0; i<nb_redacteurs; i++)
            pthread_create(thread_courant++, NULL, redacteur, &donnees_thread);

        for (i=0; i<nb_lecteurs+nb_redacteurs; i++)
            pthread_join(threads[i], &resultat);
        detruire_lecteur_redacteur(&donnees_thread.lecteur_redacteur);
        free(threads);
        return 0;
    }
```

ANNEXE B : Exemple d'utilisation : liste chaînée thread-safe

```
struct linked_list {
 int nb;
 struct linked_list *next;
};
struct linked_list_head {
 struct lectred sync;
 struct linked_list *head;
};

void list_init(struct linked_list_head *list) {
 list->head=NULL;
 init(&list->sync);
}

int exists(struct linked_list_head *list, int val) {
 struct linked_list *p;
 begin_read(&list->sync);
 p=list->head;
 while(p) {
   if (p->nb == val) {
     end_read(&list->sync);
     return 1;
   }
   p=p->next;
 }
 end_read(&list->sync);
 return 0;
}

struct linked_list* remove(struct linked_list_head *list, int val) {
 struct linked_list **p, *ret=NULL;
 begin_write(&list->sync);
 p=&list->head;
 while(*p) {
   if ((*p)->nb == val) {
     ret=*p;
     *p=(*p)->next;
     break;
   }
   p=&(*p)->next;
 }
 end_write(&list->sync);
 return ret;
}
```