Chapitre 4 : Systèmes Multi-Processeurs

Section 4.1 : Utilisation de plusieurs threads

La loi de Moore ous dit que la densité de transitstors dans un microprocesseur double tous les 12 mois. Au cours des dernières années nous avons assistés à de nombreuses améliorations permettant de maintenir vraie cette loi. la première fut l'accélération des fréquences d'horloges des processeurs (qq kHz à 5GHz), plus récemment, nous avons assisté à l'apparition des processeurs à plusieurs coeurs (1 coeur -> 32 coeurs dans les derniers xeons).

> En pratique, chaque processeur possè>de une copie des registres.

4.1.1 Les threads POSIX

Sous linux, ils doivent être explicitements compilés avec -lpthread sur GCC et inclus avec le :

```
    #include <pthread.h>
```

Gestion des threads

```
    pthread_t TNAME ;
    int err;
    err = pthread_create(&TNAME, NULL, &funct, NULL) ;
    err = pthread_join(TNAME,NULL);
```

On peut ajouter des arguments et valeurs de retours respectivement à create et join avec le dernier argument de la fonction.

Section : 4.2 Communication entre les

threads

Ce qui permets aux threads de communiquer entre eux est le partage du Heap, des données statiques et du text. par contre, chaque thread possède sa propre stack et son propre contexte thread (%esp, %eip, registres)

4.2.1 Arguments et threads

On peut utiliser les arguments du threads et les join pour réccupérer et partager de l'information avec les threads, par contre, celà limite les interractions pendant que le thread est en cours.

4.2.2 Les variables globales

Le thread peut modifier la variable globale. Mais permettre à plusieurs threads de modifier la même variable globale en même temps peut donner lieu à des colision lors de l'accès à celle-ci

4.2.3 Section critique

Le problème de la section critique ou exclusion mutuelle intervient lorsque deux threads se parasitent. cette section critique est la partie du code que seul un thread peut effectuer à la fois

Section 4.3 Coordination entre threads

Ces mécanismes ont pour objectif d'éviter de modifier la même zone mémoire.

4.3.1 Exclusion mutuelle

L'exclusion mutuelle doit satisfaire les conditions suivantes :

- aucune hypothèse sur la priorité relative d'un thread
- aucune hypothèse sur la vitesse relative d'un thread
- un thread doit pouvoir s'arrêter en dehors de sa section critique sans invalider la contrainte d'exclusion mutuelle
- aucun thread ne doit être être empéché indéfiniment d'accéder à sa section critique. (principe de *vivacité*)

Exclusion mutuelle sur monoprocesseur

Si il est clair qu'un processeur avec plusieurs coeurs peut effectuer plusieurs opérations en simultanées qu'en est-il pour les monoprocesseurs. Le **multitasking** est la capacité d'un processeur à exécuter plusieurs tâches sur un même processeur. (illusion car il n'est capable d'éxécuter qu'une séquence d'inistructions à la fois).

Pour effectuer cette illusion, le système génère des changements de contextes (registres) entre les threads. elle enregistre les registres dans la zone mémoire associée et charge les autres.

Sous UNIX, on peut générer ces changements de contextes de deux façons :

- **interruption** du *hardware* (ex : periphérique I/O)
- **appel système bloquant** d'un thread (interraction avec l'OS)

Le choix de quel thread doit être lancé est géré par l'ordonnanceur.

> round robin est un ordonnanceur simple qui donne acces à un thread 1/n temps (à cause de son implémentation en liste circulaire

Algorithme de Peterson

Deadlock : quand les threads sont bloqués mutuellement. **Livelock** : plusieurs thread continuent à foctionner mais aucun d'entre eux ne fait quelque chose d'utile.

```
    #define A 0
    #define B 1
    int flag[];
    flag[A]=false;
    flag[B]=false;

    //thread A
    flag [A] = true;
    turn = B;
    while ((flag[B]==true)&&(turn==B))
    { /*LOOP*/}
    section_critique();
    flag[A] = false;

    //thread B
    flag [B] = true;
    turn = A;
    while ((flag[A]==true)&&(turn==A))
    { /*LOOP*/}
    section_critique();
    flag[B] = false;
```

Cet algorithme n'est pas implémentable sur les machines actuelles car le compilateur modifie le code pour le rendre plus léger.

Opérations atomiques

une **opération atomique** : Opération qui lorsqu'elle est exécutée sur un processeur ne peut pas être interrompue par l'arrivée d'une interruption. (ex: xchg le pendant de movl)

Coordination par Mutex

Un **Mutex**: structure de données qui permet de contrôler l'accès à une ressource. Un mutex possède deux états :

- Locked : accès exclusif à la ressource
- Unlocked : libération de la ressource

```
pthread_mutex_t mutex;
err = pthread_mutex_init(&mutex,null);
err = pthread_mutex_lock(&mutexl);
err = pthread_mutex_unlock(&mutex);
err = pthread_mutex_destroy(&mutex);
if (err!=0){error lock}
```

4.3.2 Le problème des philosophes

Nécéssité de donner l'accès à plusieurs ressources à plusieurs threads. Ce problème porte le nom de **problème des philosophes**

Section 4.4 : Sémaphores

Un **Sématphore** est une structure de donneés qui est maintenue par le système d'exploitation et contient :

- Valeur positive ou nulle du sémaphore (int)
- Queue qui contient des pointeurs vers les threads qui sont bloquésen attente de se sémaphore

Un sémaphore initialisé à 1 est équivalent à un mutex

4.4.1 Sémaphores POSIX

```
#include <semaphore.h>
sem_t s;
int sem_init(s,0,VALINIT);
int sem_wait(&s);
int sem_post(&s);
int sem_destroy(&s);
```

4.4.2 Exclusion mutuelle

initialisée à 1, un sémaphore est équivalent à un mutex

4.4.3 Problème du rendez-vous

- 1 sémaphore (nb de threads dans la file)
- int count =0 (jusqu'à n)
- mutex pour le count

Permet d'attendre que tous aient passés la première phase avant d'entammer la deuxiemme.

4.4.4 Problème des Producteurs-consommateurs

- 2 sémaphores (full init à 0, empty init à N)
- 1 mutex pour protéger le buffer

Le modèle se base sur deux types de threads :

- **Producteurs** : threads qui produisent des données et placent leurs résultats dans une zone mémoire accessible par le consommateur.
- **Consommateur** : threads qui utilisent la valeur calculée par les producteurs.

4.4.5 Problème des Readers-Writers

- Les **Lecteurs** (Readers) : threads qui lisent une structure de données (ou database) sans la modifier. Rien ne s'oppose à ce que plusieurs readers readers s'exécutent simultanément.
- Les **écrivains** (Writers) : threads qui modifient la structure de donnée (ou database). Il ne peut y avoir aucun writer ou reader pendant que le writer écrit dans la database.

Composantes :

- 1 mutex (readcount)

- 1 sémaphore (accès à la db)
- int readcount (nb de readers)

Section 4.5 Compléments sur les threads POSIX

4.5.1 Variables Volatile

une **variable volatile** est une variable qui indique au compilateur qu'il doit recharger en mémoire la variable à chaque fois qu'elle est utilisée.

Attention

- volatile n'empèche pas l'obligation de mutex
- volatile provoque une reduction des performances

4.5.2 Variables spécifiques à un Thread

```
    --thread int VARGLOBALE;
```

Une **variable spécifique à un thread** est une variable qui est indiquée au compilateur et à la librairie POSIX comme obligant chaque thread à posséder une copie de cette variable globale pour chaque thread.

Il existe une deuxiemme option bien plus complexe que nous n'utiliserons pas

4.5.3 Fonctions Thread-safe

une **fonction thread-safe** est une fonction qui peut-être utilisée à l'interrieur de threads sans problèmes. On évitera donc les variables globales.

Section 4.6 Loi de Amdahl

Un programme P peut être découpé en deux parties :

- une partie purement séquentiell (génélamement initialisation et collecte des résultats).
- une partie parallélisable qui est généralement le coeur de l'algorithme

La loi d'amdahl est un maximum théorique et difficile à atteindre. elle suppose une parallélisation parfaire. Le gain lié à la parallélisation est aussi lié à la proportion de séquentiel par rapport au parallèle. Une partie séquentielle forte est caractéristique d'une performance faibe apporté par la paralélisation.

Qu'est-ce qui affecte les performances ?

- Les boucles
- Les grandes structures de données
- Mutex
- Sémaphores

Section 4.7 : Les processus

Sous unix, les programmes sont exécutées sous la forme de *processus*.

Un **processus** est un ensemble d'instructions pour le processeur, ainsi que des données en mémoire et un contexte (si le processeur utlise un seul thread, plusieurs contextes).

4.7.1 Les Librairies

Les librairies sont des ensembles de fonctions. elles doivent être compilées et incluses dans le programme. Il existe deux types de librairies en C :

- les **librairies statiques** : librairie de fonction intégrée directement avec le programme (lourd)
- Les **librairies partagées** : ensemble de fonctions qui peuvent être appelées mais stockées une seule fois sur le disque.

Pour créer une librairie, il faut d'abord compiler les fichiers objets de la librairie, ensuite créer l'archive de la librairie. ces librairies se terminent par .a

4.7.2 Appels Système

Le processeur possède deux modes :

- **mode utilisateur** : permets à l'utilisateur d'interragir mais effectue des vérifications supplémntaires
- **mode protégé** : est utilisé par le système et ne requiert pas de vérifications

4.7.3 Création d'un thread

1. Localisation de l'exécutable
2. Création d'un processus
3. Attend la fin du programme et récupère la valeur de retour

fork(2) : appel système qui permet de créer un processus (copie complète du processus appelant). Créant ainsi un processus père et un processus fils

- -1 si une erreur de création
- 0 si le fork s'effectue sans difficultées

Après le fork, les deux proccessus sont copiés à l'identique à l'exception de certaines données de l'os (ex: pid). Contrairement aux threads, le père et le fils ne partagent pas le segment, le heap et le stack. Ils ne peuvent donc pas être utilisés pour communiquer entre père et fils.

Le kernel gère le processus et attribue un identifiant à chaque processus.

Attention Le STDIN possède un buffer et peut-être impacté par les écritures concurentes.

4.7.4 Fin d'un processus

Un processus peut se terminer de deux façons principales :

- Par un return dans la main
- Par un appel explicit à exit(3)
    - permet au fils de retourner son statut à son père (qui le récupère via waitpid(2) (appel syst. bloquant))

Cas particuliers de processus

- Processus orphelin : le père se termine avant son fils (le fils est donc relié au processus 1 (init) qui devient son “père de substitution”)
- Processus Zombie : processus fils qui à terminé mais dont son père n'a pas encore réccupéré la valeur de retour. le Kernel va le transformer en une petite structure de donnée.

4.7.5 Exécution d'un programme

```
    execve(path, argv[],envp[])
```

- *Path* : nom complet du fichier exécutable qui doit être lancé
- *argv* : pointeur vers la chaine de caractères des arguments
- *envp* : pointeur vers l'envorionnement du programme
- fork
- execve
- waitpid

lors de l'exécution d'un programme, execve vérifie le bit de permission de l'exécutable

4.7.6 Table de processus

- **ps** : table de processus en cours d'exec
- **top** : table de processus en cours d'exec, temps cpu, mémoire (interractive)
- **pstree** : affiche les processus avec les relations père-fils