Chapitre 3: Threads

3.1 Un serveur Web multi-fil a été programmé. Si sur un système d'exploitation les seuls appels de lecture disponibles sont bloquants, est-ce que des fils d'exécution en mode usager seraient une bonne solution?

Une librairie qui fournit des fils d'exécution en mode usager ne peut se permettre de bloquer tout le processus à chaque appel pour une lecture. Afin de ne pas bloquer, elle doit pouvoir commander les données de manière asynchrone sans bloquer (appel de lecture asynchrone) et pouvoir vérifier avec un appel (e.g. select ou poll) si les données sont prêtes (octets disponibles sur un socket ou lecture asynchrone complétée). Si ainsi les données ne sont pas prêtes, la librairie passe le contrôle à un autre fil d'exécution du processus. Dans ce cas-ci, ce ne serait pas un bon choix de prendre des fils d'exécution en mode usager, s'il n'y a pas moyen d'éviter de bloquer afin de permettre la poursuite avec un autre fil.

3.2 L'information associée à un fil d'exécution a été décrite comme étant une pile, les registres, un état et une référence à l'information sur le processus qui le contient. Nous savons que plusieurs piles, une par fil, seront effectivement présentes dans l'espace adressable. Par contre, le processeur ne contient qu'un ensemble de registres, pourquoi donc en avoir une copie par fil?

Si les fils d'exécution s'exécutent en parallèle sur plusieurs processeurs, déjà là il existe plusieurs ensembles de registres, un par processeur. Il faut ainsi sauvegarder le contenu des registres afin de pouvoir poursuivre l'exécution du fil là où elle a été interrompue. Ceci est vrai autant pour les processus (mono-fil), que pour chaque fil d'exécution, autant en mode usager qu'en mode noyau. En effet, chaque fil d'exécution a sa propre pile et exécute à un emplacement différent, ce qui requiert des registres pointeurs de pile, compteur de programme...

3.3 Certains processeurs supportent la technologie hyperthread. Le processeur apparaît alors comme deux processeurs et peut rouler deux fils d'exécution mais un seul progresse à la fois. L'intérêt est que le processeur passe à l'autre fil dès que celui en exécution bloque sur une faute de cache. Que cela demande-t-il comme ressources additionnelles dans le processeur? Est-ce que cela pose problème pour un système temps réel?

Le processeur doit pouvoir maintenir l'état complet pour deux fils d'exécution. Ceci requiert donc deux jeux complets de registres, autant les registres visibles que ceux internes (e.g. résultats internes dans le pipeline...). La technologie hyperthread est intéressante car elle permet d'utiliser le temps normalement perdu en attente pour la cache, ce qui peut facilement représenter 10 à 20% du

temps. Par contre, le système d'exploitation n'a aucun contrôle sur l'ordonnancement entre ces deux processus; ils apparaissent comme deux processus s'exécutant sur deux processeurs indépendants.

3.4 Vous faites la conception d'un serveur de fichier et hésitez entre une architecture mono ou multi-fil d'exécution. Chaque requête demande 15ms de traitement du CPU lorsque le bloc désiré est en cache d'E/S, ce qui arrive deux fois sur trois. Autrement, il s'ajoute 75ms de temps d'accès au disque pendant lequel le fil d'exécution est bloqué. Quelle sera la performance en requêtes servies par seconde, avec ou sans plusieurs fils d'exécution?

Dans tous les cas, il faut 15ms de temps CPU. Une fois sur trois, s'ajoute 75ms de latence pour les E/S, pour un temps moyen de 15ms + 75ms / 3 = 40ms. Un service mono-fil sera donc limité à 1000ms/s / 40ms/requête = 25 requête/s. Si un autre fil peut progresser pendant l'attente d'E/S, en supposant que les E/S ne deviennent pas le goûlot d'étranglement, un service multi-fil pourrait traiter une requête par 15ms, saturant le CPU, ce qui donne 1000ms/s / 15ms/requête = 66 requêtes/s.

3.5 Dans un système avec des fils d'exécution en mode noyau, tel que Linux, il faut une pile par fil. Est-ce différent lorsque les fils d'exécution sont gérés en mode usager?

Non, il demeure que chaque fil a ses propres appels, variables locales...

3.6 Pendant le développement d'un nouveau processeur, un modèle est usuellement construit afin de le simuler. Le processeur est alors simulé une instruction à la fois, séquentiellement, même s'il s'agit en fait d'un nouveau multi-processeur. Les courses qui se produisent normalement sur un multi-processeur pourront-elles être simulées et détectées dans un tel contexte où la simulation procède séquentiellement?

Même si le simulateur s'exécute séquentiellement, il peut, à chaque coup d'horloge simulé, traiter ce qui se passe sur de nombreux processeurs et correctement refléter le comportement d'une course entre les fils d'exécution qui roulent sur deux processeurs simulés.

Les questions suivantes sont associées à un code source.

3.7 Quelle est la valeur finale de a? Combien de millions de fois la variable a est-elle incrémentée?

```c
// ex03.c
#define MAX 100000000 // 100 millions
static uint64_t a = 0;

void *count(void *arg) {
    volatile uint64_t *var = (uint64_t *) arg;
    volatile uint64_t i;
    for (i = 0; i < MAX; i++) {
        *var = *var + 1;
    }
    return NULL;
}

int main(int argc, char **argv) {
    int p;
    int i;
    pthread_t t1;
    pthread_t t2;

    pthread_create(&t1, NULL, count, &a);
    pthread_create(&t2, NULL, count, &a);
    count(&a);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("pid=%d a=%" PRId64 "\n", getpid(), a);
    return EXIT_SUCCESS;
}
```