Série d'exercices 2 : Fils d'exécution

2.1 Un serveur Web multi-fil a été programmé. Si on considère utiliser seulement des appels systèmes bloquants, est-ce que des fils d'exécution en mode usager sont une bonne solution?

Une librairie qui fournit des fils d'exécution en mode usager ne peut se permettre de bloquer pour une lecture, car tout le processus est alors bloqué et rien d'autre ne peut s'exécuter. 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 les données ne sont pas prêtes, la librairie passe le contrôle à une autre fonction du processus. Dans ce cas-ci, s'il n'est pas possible d'éviter les blocages, ce ne serait pas un bon choix de prendre des fils d'exécution en mode usager, car on ne pourrait pas continuer l'exécution.

2.2 L'information associée à un fil d'exécution contient une pile, les registres et un ensemble d'information maintenues par le noyau (numéro, table de page, etc). Nous savons que chaque fil a sa propre pile dans l'espace adressable. Par contre, le processeur ne contient qu'un ensemble de registres, pourquoi donc en avoir une copie par fil?

Quand une interruption ou un changement de contexte (ordonnanceur) survient, il faut sauvegarder le contenu des registres pour les utiliser. Lorsque le fil est redémarré, les registres sont restaurés à partir de la sauvegarde pour poursuivre l'exécution du fil là où elle a été interrompue. On doit donc pouvoir sauvegarder les registres de chaque entité ordonnançable, dont les fils d'exécution.

2.3 La plupart des processeurs Intel supportent la technologie HyperThread. Un processeur réel apparaît alors comme deux processeurs logiques et peut exécuter simultanément deux fils d'exécution. Les instructions des deux programmes sont exécutées en alternance. L'intérêt est que le processeur passe d'un fil à l'autre, et si un défaut de cache se produit sur un fil d'exécution, il peut continuer de faire progresser l'autre fil. 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. Il est généralement préférable d'utiliser en priorité les processeurs physiques distincts, puis d'utiliser les cœurs logiques lorsque le nombre de

tâches augmente. Par contre, bien que le débit maximal soit amélioré par HyperThread, cela peut faire varier grandement le temps d'exécution d'un programme s'il partage un processeur physique avec un autre programme, ce qui n'est pas souhaitable dans un système temps réel. Dans un système temps réel, on veut un délai stable et borné, donc on pourrait à la limite réserver un cœur physique pour la tâche importante.

2.4 Vous faites la conception d'un serveur de fichier et hésitez entre une architecture mono ou multi-fil d'exécution. Le système possède un seul processeur. 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 un ou 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 goulot 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.

2.5 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, etc.

2.6 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 processeur multicœur. Les conditions critiques (courses) qui se produisent normalement sur un processeur multicœur 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.

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

```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;
}
```