

Threads POSIX (1) Création et gestion

Introduction

- ▶ Loi de Moore plus trop d'actualité
- ▶ La puissance de calcul augmente maintenant (majoritairement) avec la multiplication des coeurs, des processeurs et des machines
- ▶ Pour tirer parti des machines multicoeurs : utilisation d'algorithmes parallèles et programmes multithreadés

Introduction

Il est possible d'implémenter des algorithmes parallèles avec ce qu'on a vu jusque là, mais c'est lourd :

- ▶ fork : appel système lourd
- ▶ chaque processus a son espace mémoire (perte de mémoire)
- ▶ chaque processus a ses structures systèmes (table des pages, fichiers ouverts...)
- ▶ context switch lent
- ▶ communication inter-processus généralement lente

Solution : les processus légers ("threads")



Processus légers (*threads*)

- ▶ Plusieurs visions et implémentations possibles
- ▶ Introduction dans la norme POSIX en 1995 (POSIX 1.c)

Différence entre un thread et un processus normal :

- ▶ un thread est une "partie" d'un processus
- ▶ un processus est l'exécution d'un ensemble ($>=1$) de threads

Différence entre un ensemble de threads et un ensemble de processus :

- ▶ les threads partagent pratiquement tout (mémoire, pid, fichiers ouverts...)
- ▶ la synchronisation n'est plus gérée au niveau du système, mais est laissée à l'utilisateur



Threads : avantages et inconvénients

Avantages et inconvénients par rapport à des processus communiquants avec les "anciens" mécanismes

- ▶ partage de la mémoire : mécanisme rapide de communication inter-thread
- ▶ plus léger : moins de données système à recopier
- ▶ plus rapide : le context-switch est plus facile

Les inconvénients sont (uniquement) des "difficultés" de programmation :

- ▶ Les threads utilisent les mêmes copies des bibliothèques : les bibliothèques doivent être "MT-safe"
- ▶ Il faut gérer la synchronisation (mutex, sémaphores...)

En cas de mauvaise synchronisation :

- ▶ Comportement "aléatoire" : bugs, segfaults, exploitations...
- ▶ Interblocage



Ordonnancement des threads (et processus) :

▶ Coopératif :

Chaque thread doit explicitement rendre la main.

Problème : si un thread plante ou ne rend pas la main, les autres threads ne s'exécutent plus.

▶ Préemptif :

Le système peut arrêter n'importe quel thread, pour switcher à un autre thread (via un mécanisme de temporisation et d'interruption matérielle)

L'ordonnancement des systèmes d'exploitation "modernes" se fait préemptivement (Unix, windows...). Mais l'ordonnancement coopératif peut toujours exister au niveau utilisateur.



Modèle 1 :1 (threads système ou threads noyau)

- ▶ Chaque thread correspond à une entité ordonnancée par le noyau.
- ▶ L'ordonnancement est alors préemptive.
- ▶ Le comportement va être proche d'un ensemble de processus qui partagent leur mémoire et leurs données système.

Avantage : permet à un processus d'utiliser plusieurs coeurs

Implémentations de threads 1 :1 : Linux Thread, NPTL



Modèle N :1 (threads utilisateurs)

- ▶ Tous les threads du processus correspondent à une entité ordonnancée par le noyau.
- ▶ L'ordonnancement et le *switch* entre les threads se fait au niveau utilisateur

Avantage :

- ▶ le *switch* est très rapide (*pas d'appel système*)
- ▶ possibilité d'avoir énormément de threads
- ▶ possible même sur des systèmes légers, sans ordonateur (systèmes embarqués)

Implémentations de threads N :1 : GNU Pth, State Threads



Modèle N :1 (threads utilisateurs)

- ▶ Certains langages/paradigmes de programmation utilisent nativement le multi-threading
- ▶ C'est le cas notamment de ceux qui utilisent les *coroutines*.
- ▶ Une implémentation naïve donnerait trop de threads, et trop de "context switches", pour être efficacement traités par le système.
- ▶ Ces threads utilisateurs légers, exécutions de coroutines, sont appelés des *fibres*

processus / thread système / thread utilisateur / fibre



Modèle M :N ("hybride")

- ▶ Tire les avantages des modèles 1 :1 et N :1
- ▶ Idéalement, N = nombre de coeurs

Exemples : GHC (Glasgow Haskell compiler), threads NetBSD...



Threads POSIX

Une norme POSIX pour créer des threads, et de les synchroniser

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

Compiler avec l'option -pthread

Intègre :

- ▶ Fonctions pour créer, attendre et tuer un thread
- ▶ Des mécanismes de synchronisation : mutex et variables de condition



L'implémentation Linux de pthread

- ▶ L'implémentation actuelle des threads POSIX sous Linux est NPTL (Native POSIX Threads Library). Elle respecte la norme POSIX, et rajoute quelques fonctions non POSIX.
- ▶ En interne, il s'agit de threads systèmes (préemptifs et 1 : 1).
- ▶ NPTL utilise des appels système à clone() et futex(), et des signaux temps réels.



Threads POSIX : partage

Les pthreads d'un processus partagent :

- ▶ le PID, le PPID
- ▶ l'espace mémoire
- ▶ les descripteurs de fichiers ouverts
- ▶ les utilisateurs propriétaires
- ▶ les handlers des signaux
- ▶ le répertoire courant, le masque de fichiers...

Ne partagent pas :

- ▶ les identifiants des threads
- ▶ la pile
- ▶ le masque des signaux
- ▶ errno (exercice : comment cela est possible?)



Créer un thread

Au départ, le processus est constitué d'un unique thread : celui qui exécute la fonction main

On peut créer d'autres threads, avec la fonction suivante :

```
int pthread_create(  
    pthread_t *thread,  
    const pthread_attr_t *attr,  
    void *(*start_routine) (void *),  
    void *arg  
) ;
```

- ▶ thread : l'adresse mémoire où sera copié l'identifiant du nouveau thread
- ▶ attr : des attributs (NULL = défaut)
- ▶ start_routine : la fonction d'entrée du thread
- ▶ arg : l'argument de la fonction



Créer un thread

```
Exemple :  
void *fonction ( void *a )  
{  
    ...  
    return NULL;  
}  
  
...  
pthread_t id ;  
pthread_create(&id ,NULL, fonction ,NULL)  
...
```



Identifiant d'un pthread

- ▶ Un pthread n'a pas de PID propre (ils partagent tous le même PID, celui du processus contenant les threads)
- ▶ L'identifiant d'un pthread est un objet du type `pthread_t`.
- ▶ La norme ne dit pas ce qu'il y a dans `pthread_t` (objet opaque).
- ▶ `pthread_self()` renvoie le `pthread_t` du thread courant.
- ▶ `pthread_equal(pthread_t a, pthread_t b)` permet de comparer deux identifiants



Argument et retour d'un pthread

- ▶ Il est possible de passer un argument à la fonction qu'on appelle : un pointeur, qu'on peut faire pointer vers la structure de son choix
- ▶ Quand `start_routine` termine, le thread se termine.
- ▶ Un thread peut également terminer avec `pthread_exit()`
- ▶ Le thread `main` est spécial, sa terminaison termine le processus, même si d'autres threads sont encore en exécution.
- ▶ `exit()` dans n'importe quel thread termine le processus (i.e. tous les threads)



Fin et attente d'un thread

Similairement à un processus, un thread renvoie un code retour : un pointeur.

Similairement à un fils qui devient zombi, le code retour du thread est gardée en mémoire jusqu'à ce qu'un autre thread le "rejoigne"

Il n'y a pas de notion de père/fils :

- ▶ n'importe quel thread peut joindre n'importe quel thread
- ▶ si plusieurs attentes du même thread : comportement indéfini



Fin et attente d'un thread

```
int pthread_join(pthread_t thread, void **  
retval);
```

retval : un pointeur sur une variable (contenant un pointeur), où le code retour sera copié.

- ▶ NULL : ignoré
- ▶ thread "annulé" : PTHREAD_CANCELED

Il existe des versions non bloquantes dans NPTL (non POSIX!) :

```
pthread_tryjoin_np, pthread_timedjoin_np
```



Détacher un thread

Si on ne veut pas avoir à gérer la fin d'un thread, on peut le "détacher".

```
int pthread_detach(pthread_t thread);
```

- ▶ La structure sera détruite à la terminaison du thread
- ▶ Il ne sera pas possible de retrouver son pointeur retour avec pthread_join



Annuler un thread

```
int pthread_cancel(pthread_t thread);
```

Permet de d'"annuler" (de terminer) un thread

Attention : dans beaucoup de cas, on ne peut pas simplement terminer un thread sans risquer des fuites mémoires, ou des interblocages.

Il ne s'agit pas de "tuer" un thread : le thread doit préparer son annulation.

Il faut faire attention :

- ▶ aux fuites mémoires (mémoire dynamique allouée par le thread)
- ▶ aux sections critiques (par exemple, si le thread bloque un mutex)



Annuler un thread

Pour cela, on peut rajouter des "handlers" qui seront exécutés à l'annulation d'un thread

```
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- ▶ push : rajoute dans une pile une nouvelle fonction à exécuter en cas d'annulation
- ▶ pop : enlève le dernier élément de la pile (et l'exécute si execute n'est pas 0)



Annuler un thread

Le thread peut (doit) aussi dire quand il peut être annulé

```
int pthread_setcancelstate (int state , int *oldstate );
int pthread_setcanceltype (int type , int *oldtype );
void pthread_cancel (void );
```

- ▶ `pthread_setcancelstate` (des)active la possibilité d'annulation du thread courant
- ▶ `pthread_setcanceltype` spécifie si l'annulation se fait immédiatement ou à un point d'annulation
- ▶ `pthread_cancel` spécifie un point d'annulation



Attributs

`pthread_attr_t` : objet (opaque) spécifiant les attributs d'un thread.

- ▶ `pthread_attr_init` / `pthread_attr_destroy` : initialise (détruit) un attribut
- ▶ `pthread_attr_setstacksize` (`pthread_attr_setstackaddr`) permet de spécifier la taille (l'emplacement) de la pile
- ▶ `pthread_attr_setdetachstate` : spécifie l'état détaché
- ▶ Il est également possible de spécifier des paramètres d'ordonnancement



Autres choses de pthread

Des parties importantes de pthread seront vus par la suite :

- ▶ les *mutex*
- ▶ variables de condition

Ces mécanismes permettent de synchroniser les threads lors d'accès concurrents.

Comportement avec les signaux

- ▶ Les handlers des signaux sont partagés entre tous les threads.
- ▶ Mais chaque thread a son propre masque de signaux !
- ▶ On peut modifier le masque d'un thread via `pthread_sigmask` (mêmes arguments que `sigprocmask`).
- ▶ On peut envoyer un signal à un thread spécifique via `pthread_kill(pthread_t id, int sig)`
- ▶ On peut attendre un signal avec `sigwait`

Comportement avec exec et fork

- ▶ Comportement avec exec :
Si un thread fait un appel à exec (qui n'échoue pas), tous les autres threads sont tués.
- ▶ Comportement avec fork :
Seul le thread appelant fork est dupliqué.
Problème : comme les autres threads n'existent plus dans le fils, il se peut qu'un mutex ne soit jamais libéré.
Une solution est d'utiliser pthread_atfork qui rajoute des handlers en cas de fork.



Le début des problèmes...

- ▶ Les threads partagent leur mémoire. Y compris le segment des données des bibliothèques.
- ▶ Les (fonctions des) bibliothèques doivent donc être prévues pour un usage multi-thread
- ▶ Lors de la communication par mémoire partagée, il faut aussi faire attention à ce que deux threads ne travaillent pas sur la même zone mémoire en même temps
- ▶ Sinon : bug, plantage, exploitation, heisenbug...



Libraires "MT-safe"

- Quand on utilise une librairie (ou une fonction d'une librairie) dans un programme multi-threadé, il faut vérifier si elle a été prévue pour une utilisation multi-threadée (MT-safe)
- Certaines fonctions de la libc sont MT-safe, d'autres non
 - Il faut voir le manuel !

Ré-entrée :

- Une fonction est appelée "re-entrant" si elle peut être interrompue, et rappelée (de façon sûre).

Exemple : strtok

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

strtok coupe str aux caractères présents dans delim

Si str=NULL, elle continue de découper la chaîne précédente

```
char w[]="2:5:6:1:2:6:3";
char *p=strtok(w,":");
while (p) {
    printf ("%s",p);
    p=strtok(NULL,":");
}
```

Sortie : 2 5 6 1 2 6 3

Problème (avec les programmes multithreads) : strtok garde en interne un pointeur sur la position courante dans le chaîne.

Exemple : strtok

Version ré-entrant : strtok_r

```
char *strtok_r(char *str, const char *delim,
char **saveptr)
```

strtok_r ne garde pas un pointeur interne : il faut lui en fournir un

```
char *tmp;
char w[] = "2:5:6:1:2:6:3";
char *p=strtok_r(w,":",&tmp);
while (p) {
    printf("%s",p);
    p=strtok_r(NULL,":",&tmp);
}
```



Problème d'optimisation du compilateur

```
int i;

void *thread(void *a) {
    sleep(1);
    i=1;
}

pthread_create(id,NULL,thread,NULL);
i=0;
while(i==0) { usleep(1000); }
```

Avec trop d'optimisations, le compilateur peut considérer que `i` reste à 0, car jamais affectée à autre chose que 0 dans main.

Modificateur du C : volatile. Indique au compilateur qu'une variable peut changer entre ses différents accès.



Problèmes de synchronisation

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :

- ▶ y compris au milieu d'une ligne
- ▶ y compris au milieu d'une instruction basique en C (ex : i++)

Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistent.

- ▶ Comportement imprévisible !

Problèmes de synchronisation

```
int z=0;
```

```
void* th(void *r) {
    for(int a=0;a<1000000;a++)
        z++;
}
```

```
int main() {
    pthread_t id1,id2;
    pthread_create(&id1,NULL,th,NULL);
    pthread_create(&id2,NULL,th,NULL);
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    printf("%d\n",z);
}
```

sortie : 94829

Atomicité

Code assembleur de `z++` :

```
movq    z(%rip) , %rax
addq    $1 , %rax
movq    %rax , z(%rip)
```

On aimeraît que ces 3 instructions ne puissent être interrompues.

Instruction(s) atomique : instruction(s) ne pouvant être interrompues.

Mais : il n'est pas possible de bloquer les interruptions dans le mode utilisateur.

Pour rendre atomique (vis à vis des autres threads) une accès sur une zone mémoire : mécanisme d'exclusion mutuelle (mutex)
C'est le sujet du prochain cours !



Suite :

- ▶ Threads et synchronisation
- ▶ Réseau...

