Cours 2: Processus et Threads

Christophe Morvan

Université Paris-Est, Marne-la-Vallée

15 septembre 2015

Plan

Progression

Qu'est un processus

- Un processus est un programme qui s'exécute
- Les OS modernes permettent l'exécution en parallèle de plusieurs processus

Exemple

Sur le même ordinateur

- `gcc prog1.c` `# compilation 1`
- `gcc prog2.c` `# compilation 2`
- `firefox` `# navigateur web`
- `eclipse` `# IDE java`

Non processus multiples

- Plusieurs fenêtres de firefox
- Plusieurs fenêtres de LibreOffice

Pourquoi avoir des processus

Globalement

La programmation d'un processus ne tient pas compte de l'existence simultanée des autres processus
Y compris de l'existence de plusieurs instances du même processus

Gain de vitesse

Le processeur est l'élément le plus rapide d'un ordinateur
La plupart des programmes interagissent :

- Mémoire
- Disques
- Réseau
- Utilisateurs

Lorsqu'un programme attends des données, un autre peut s'exécuter

Les processus sont antérieurs aux ordinateurs

Les principes de processus et de parallélisation sont très anciens

Exemple

- Supposons qu'un artisan réalise un meuble en 10 mois
- Un entreprise peut en employer 100 pour faire 100 meubles (Donc, en moyenne, 10 meubles par mois)
- La latence pour obtenir le premier est supérieure à 1/10 mois
- La production mensuelle est éventuellement inférieure à 10 unités selon les possibilités de parallélisation
- Mais elle peut être plus grande que 10 si on élimine les attentes inutiles

Vision du processus

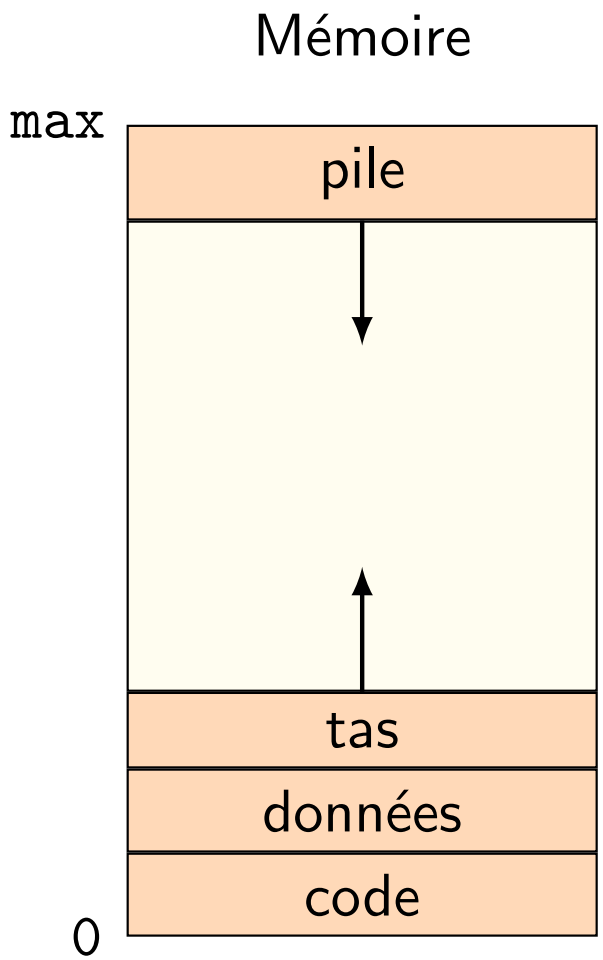
Pour chaque processus

- Espace d'adressage propre
- Fichiers ouverts propres
- Son propre CPU

La valeur : `*(char *)0x0087` est différente pour deux processus distincts

Interactions entre processus

Le plus simple : au travers des fichiers (emacs édite, gcc compile)
Plus complexe : un shell et une commande ou une application graphique et un environnement de bureau



Création du processus - 1

Quand

- Initialisation du système
- Appel système de création par un processus
- Demande d'un utilisateur
- Lancement d'un travail de traitement par lot (mainframes)

Sous UNIX *un seul appel système* pour créer de nouveau processus

Au démarrage un seul processus créé : `init` qui a pour descendants tous les processus qui s'exécuteront sur le système

Plus généralement il se crée une hiérarchie entre les processus avec des connexions entre chaque processus et ses descendants

Création du processus - 2

Appels systèmes

```
int fork ();                                    #include <unistd.h>
```

- Crée une copie identique du processus
- Le parent reçoit comme résultat le PID du descendant
- Le descendant reçoit 0 comme résultat

```
int waitpid (int pid, int *status, int option);
                                        #include <sys/wait.h>
```

- `pid` numéro à attendre (-1 n'importe quel descendant)
- `status` valeur de sortie ou signal
- `option` en général 0 (voir `man`)
- Fournit le PID du processus qui a changé, -1 en cas d'erreur

Destruction de processus - 1

Quand ?

- Arrêt volontaire
- Erreur
- Erreur fatale
- Arrêt par un autre processus

Destruction de processus - 2

Fonction bibliotèque

```
void exit(int status);
```

```
#include <stdlib.h>
#include <sys/types.h>
```

- Destruction du processus actuel
- `status` est placé dans la valeur de `waitpid`
- Convention : 0 en cas de succès, autre valeur en cas d'échec

Appel système

```
int kill(pid_t pid, int sig);
```

```
#include <signal.h>
```

- Adresse le signal `sig` au processus `pid`
- `SIGTERM` Tue le processus. Mais peut être intercepté pour "nettoyage"
- `SIGKILL` Tue le processus à coup sûr

Exécution de programmes

Appels systèmes

```
int execve(const char *file, char *const argv[], char
*const envp[]);                          #include <unistd.h>
```

- `file` : chemin de l'exécutable
- `argv` : tableau de paramètres transmis à `main`
- `envp` : tableau de variables d'environnement

Fonctions de la `libc` standard `#include <unistd.h>`

```
int execvp(const char *file, char *const argv[]);
```

Recherche le fichier dans le `PATH`, utilise l'environnement actuel

```
int execlp(const char *file, const char *arg, ...);
```

Idem, liste les arguments de `main` un par un.

Minish.c

Programme exemple

```
pid_t pid; char **av;
void doexec () {
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
/* Dans le main */
while (1) {
  parse_next_line_of_input (&av, stdin);
  switch (pid = fork ()) {
  case -1:
    perror ("fork"); break;
  case 0:
    doexec ();
  default:
    waitpid (pid, NULL, 0); break;
  } }
```

Pourquoi `fork` ?

Observation

Dans la plupart des cas, `fork` est suivi de l'appel de `execve`
Certains systèmes utilisent une combinaison des deux appels

Néanmoins

- Il est parfois utile de faire un simple `fork` :
  - Utilitaire UNIX `dump` pour archiver sur bandes
  - Lorsque la bande est terminée il faut repartir d'un certain point
  - Le fork est utilisé pour reprendre à ce point
- Le gain le plus manifeste est celui de la simplicité :
  - globalement on souhaiterai manipuler pleins d'aspects du processus fils : descripteurs de fichier, variable d'environnement, droits, ...
  - `fork` n'utilise aucun paramètre

Se passer de `fork` ?

En l'absence de `fork` la création de processus nécessite une multitude de paramètres

Se passer de `fork` ?

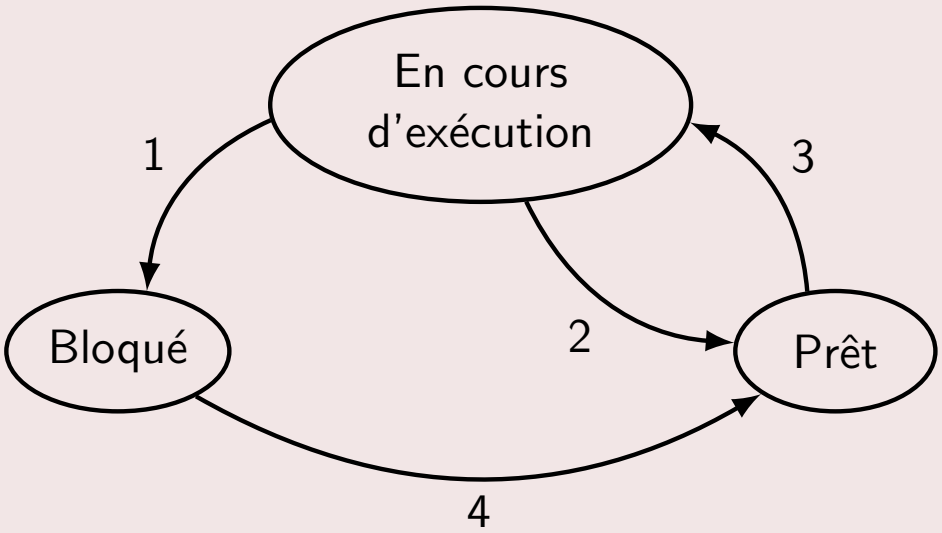
En l'absence de `fork` la création de processus nécessite une multitude de paramètres

Programme exemple

```
BOOL WINAPI CreateProcess(
  __in_opt     LPCTSTR lpApplicationName,
  __inout_opt  LPTSTR lpCommandLine,
  __in_opt     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in_opt     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in         BOOL bInheritHandles,
  __in         DWORD dwCreationFlags,
  __in_opt     LPVOID lpEnvironment,
  __in_opt     LPCTSTR lpCurrentDirectory,
  __in         LPSTARTUPINFO lpStartupInfo,
  __out        LPPROCESS_INFORMATION lpProcessInformation
);
```

États des processus

Principe



1. Le proc. est bloqué, attente de données
2. L'ordonnanceur choisi un autre processus
3. L'ordonnanceur choisi ce processus
4. La donnée est disponible

Quel processus exécuter ?

- Si aucun est exécutable : `idle`
- Si exactement 1 est exécutable, le noyau l'exécute
- Si plus de 1 processus est exécutable, le noyau prend une décision d'ordonnancement

Implémentation

PCB

Le *Process Control Block* est une structure de donnée qui contient toutes les informations nécessaires à la bonne exécution du processus.
Les PCB sont stockés dans la table des processus

Interruptions

Vecteur d'interruption : zone fixe en mémoire.
Contient les informations associés à un type de matériel
Lors de la survenue d'une interruption action sur la table des processus

État du processus
ID du processus
ID utilisateur, ...
Compteur ordinal
Registres
Mémoire
Pointeur → code
Pointeur → données
Pointeur → pile
Fichiers
Fichiers ouverts
Dossier racine
Signaux (interruptions)
...

Traitement d'un interruption

Ex : Le processus 3 est en cours d'exécution

- Un interruption disque survient (Matériel)
- Le mot d'état, ainsi que l'état des registres est placé sur la pile active par contrôleur du disque (Matériel)
- Le processeur passe alors à l'adresse spécifié par le *vecteur d'interruption* (Matériel)

Traitement logiciel

- Sauvegarde des données processus dans la *table des processus*
- La pile active est vidée
- Utilisation d'une pile temporaire "gestion de processus"
- Exécution du *service d'interruption* en C
- L'ordonnanceur détermine le nouveau processus à exécuter
- Démarrage du nouveau processus

Ordonnancement

Comment choisir le processus à exécuter

Il y a des tas de façon de choisir le processus suivant à exécuter
Il y a des contraintes :

- assurer que tout processus fini par obtenir le processeur
- ne pas donner la main à un processus bloqué
- assurer la réactivité du système
- limiter le coût du changement de contexte (cache)

Ordonnancement

Comment choisir le processus à exécuter

Il y a des tas de façon de choisir le processus suivant à exécuter
Il y a des contraintes :

- assurer que tout processus fini par obtenir le processeur
- ne pas donner la main à un processus bloqué
- assurer la réactivité du système
- limiter le coût du changement de contexte (cache)

Usuellement

Il est trop coûteux de parcourir la table des processus à chaque ordonnancement
Un approche raisonnable consiste à maintenir une file FIFO
Il est souvent raisonnable d'inclure une gestion des priorités

Efficacité de l'utilisation de processus

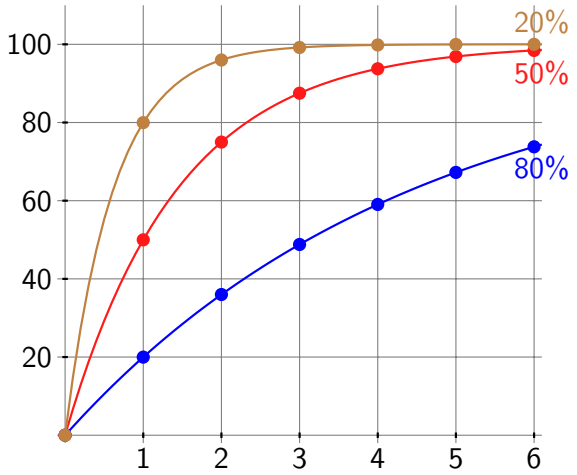
Modélisation (p, n)

Machine à 1 cœur – n processus

p : fraction de temps passé par un processus à attendre des données

La proba pour le processeur d'être inoccupé est p^n

Le taux d'occupation est donc $1 - p^n$



Efficacité de l'utilisation de processus

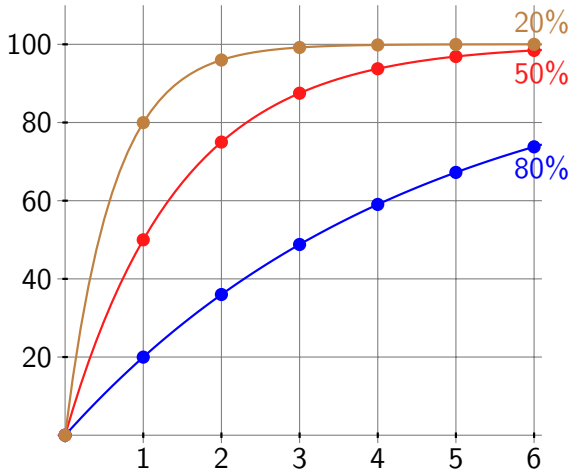
Modélisation (p, n)

Machine à 1 cœur – n processus

p : fraction de temps passé par un processus à attendre des données

La proba pour le processeur d'être inoccupé est p^n

Le taux d'occupation est donc $1 - p^n$



Ce modèle est imparfait car lorsque 2 processus sont prêt simultanément, il ne s'exécutent pas simultanément.

Néanmoins, la mémoire disponible permet une bonne approximation du degré effectif de multiprogrammation

Ainsi si on compte 256Mo pour l'OS et pour chaque programme : 1Go $\rightarrow$ 3 programmes $\rightarrow$ 49% avec 80%

Progression

Des processus legers ?

Les processus induisent une certaine lourdeur

- Changement de contexte
- Pas (peu) de partage de mémoire
- Dépendant de l'ordonnanceur
- Un processus pourrait tirer parti d'une architecture multi processeurs/multi cœur

Des processus legers ?

Les processus induisent une certaine lourdeur

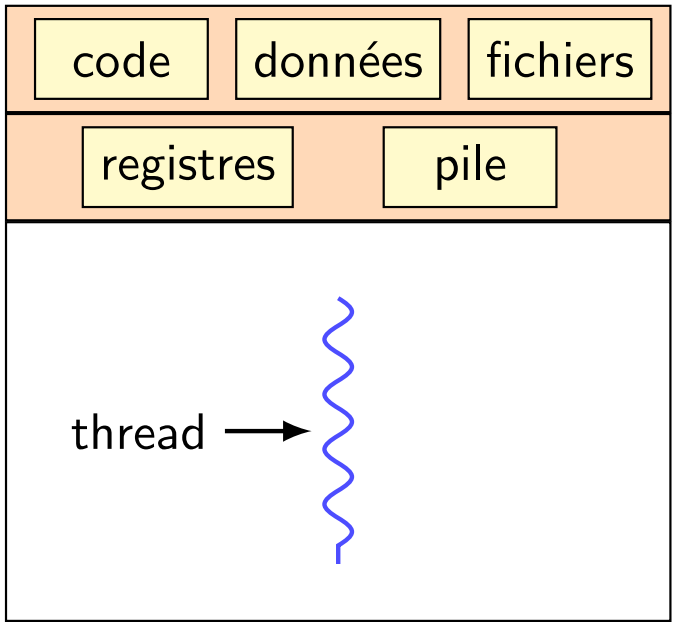
- Changement de contexte
- Pas (peu) de partage de mémoire
- Dépendant de l'ordonnanceur
- Un processus pourrait tirer parti d'une architecture multi processeurs/multi cœur

Les threads

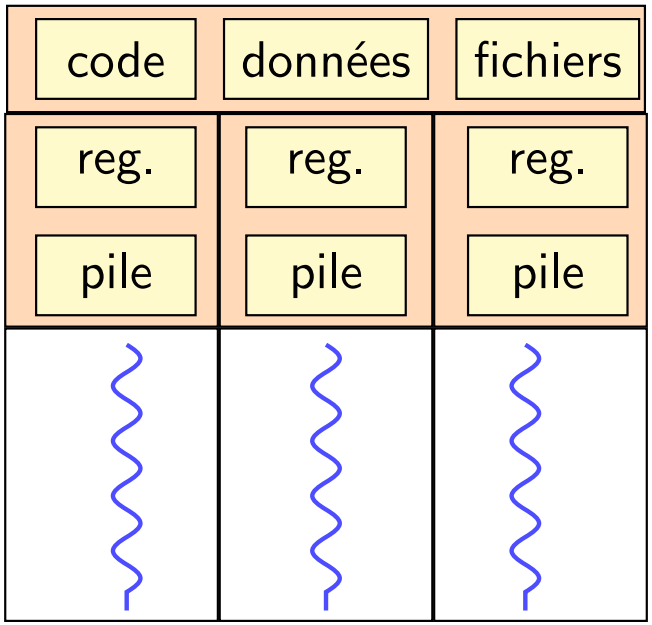
La programmation par thread permet d'intégrer dans la programmation des applications les bénéfices de la programmation parallèle
Tout en conservant une partie de la légèreté de la programmation classique (en particulier vis-à-vis de la mémoire)

Les threads

Application thread unique



Application threads multiples



Les threads

Un thread est un contexte d'exécution ordonnable

Comprend un compteur ordinal, des registres et une pile

Les programmes simples comportent un seul thread → pas de surcoût

Les programmes plus complexes peuvent tirer parti des threads

Problèmes des threads

Que se passe-t-il lorsqu'un processus possédant plusieurs threads fait l'appel système `fork` ?

Problème

Les threads sont ils recréés ?

- non : blocages
- oui : problème de lourdeur. Et que faire des attentes éventuelles ? (clavier, réseau,...)

Problèmes des threads

Que se passe-t-il lorsqu'un processus possédant plusieurs threads fait l'appel système `fork` ?

Problème

Les threads sont ils recréés ?

- non : blocages
- oui : problème de lourdeur. Et que faire des attentes éventuelles ? (clavier, réseau,...)

Au vu des données partagés, les risques d'erreurs sont nombreux :

- double allocation de mémoire
- fermeture accidentelle de fichier

Problèmes des threads

Que se passe-t-il lorsqu'un processus possédant plusieurs threads fait l'appel système `fork` ?

Problème

Les threads sont ils recréés ?

- non : blocages
- oui : problème de lourdeur. Et que faire des attentes éventuelles ? (clavier, réseau,...)

Au vu des données partagés, les risques d'erreurs sont nombreux :

- double allocation de mémoire
- fermeture accidentelle de fichier

Ces problèmes peuvent être résolus, mais demande une conception soignée, et des arbitrages.

Threads POSIX

La norme POSIX définit l'API `Pthreads`

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

Quelques appels

Création

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

Destruction

```
void pthread_exit(void *retval);
```

Attente d'un autre thread

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

`pthread_attr_init` Permet de créer et initialiser la structure de type `pthread_attr_t`

L'appel `pthread_yield` permet à un thread de *rendre la main*

Threads noyau

Un première façon de gérer les threads consiste à s'appuyer sur l'architecture des processus

Problèmes

- Toutes les opérations sur les thread sont résolues par des appels système

Threads noyau

Un première façon de gérer les threads consiste à s'appuyer sur l'architecture des processus

Problèmes

- Toutes les opérations sur les thread sont résolues par des appels système
  10 à 30 fois plus lent

Threads noyau

Un première façon de gérer les threads consiste à s'appuyer sur l'architecture des processus

Problèmes

- Toutes les opérations sur les thread sont résolues par des appels système
  10 à 30 fois plus lent
- Doit être universel $\rightarrow$ le coût des options moins usuelles est payé par tous les utilisateurs
- Plus gourmand en mémoire

Threads noyau

Un première façon de gérer les threads consiste à s'appuyer sur l'architecture des processus

Problèmes

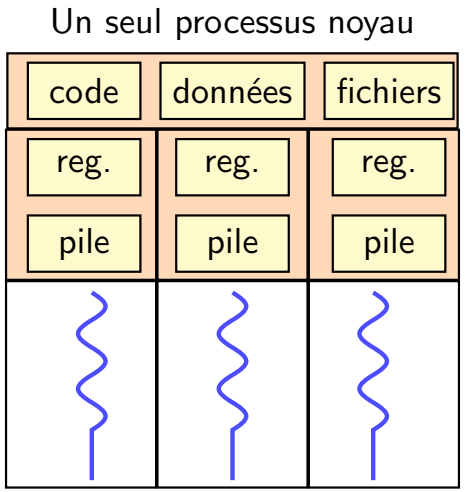
- Toutes les opérations sur les thread sont résolues par des appels système
  10 à 30 fois plus lent
- Doit être universel $\rightarrow$ le coût des options moins usuelles est payé par tous les utilisateurs
- Plus gourmand en mémoire

Avantages

Les appels systèmes bloquants ne bloquent que le thread concerné

Threads en espace utilisateur

Il est possible de créer une bibliothèque en espace utilisateur pour les threads



Principes

Un seul processus
Un seul thread noyau
Les fonctions de création de thread sont des appels de bibliothèque

Pthreads

L'API `Pthread` sous Linux est implémentée en espace noyau.

Historiquement (avant le noyau 2.6) *LinuxThreads* était l'implémentation

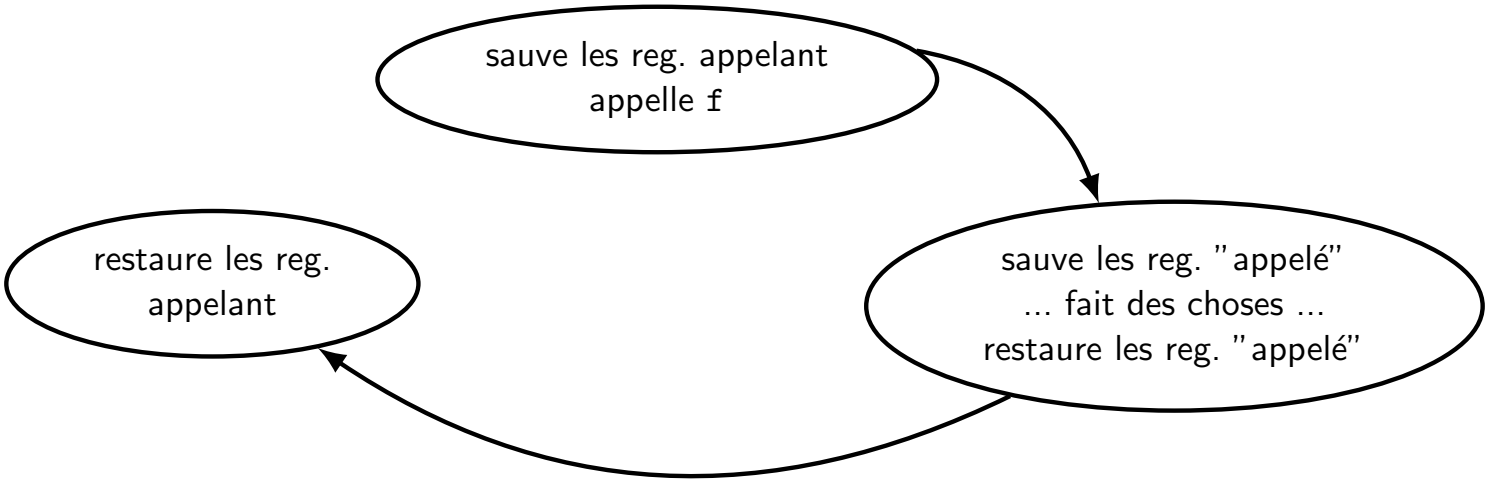
Elle n'était pas conforme avec POSIX

Depuis le noyau 2.6 la NPTL (*Native Posix Thread Library*) offre un implémentation plus performante *et* plus conforme à POSIX.

Avec NTPL tous les threads d'un même processus ont le même identifiant

Appels de fonctions

Le principe d'un thread est proche de celui d'un appel de fonction du point de vue du système



Principes

- Mémoriser des éléments sur la pile (adresse de retour, registres de l'appelant)
- Mémoriser les informations de l'appelé (n'existera pas pour les threads)

Des fonctions aux threads

Nombreuses différences

- Les threads peuvent reprendre dans un ordre quelconque :
  - Fonctions : une pile (pour tous les appels)
  - Threads : une pile par thread
- Les threads changent moins souvent
- Les threads peuvent être interrompus : le compilateur ne suffit pas à déterminer l'état antérieur. Le code d'échange stocke les registres
- Parallélisme : dans les appels de fonctions l'ordonnancement est trivial. Pour les threads l'ordonnancement est nécessaire.

Limites des threads utilisateurs

Réciproque des threads noyaux

- N'utilise pas les multi-processeurs
- Les appels systèmes bloquant bloquent tous les threads
  - Il est possible d'utiliser ses propres fonctions pour les accès réseau
  - en général c'est impossible pour accéder au disques
- Un défaut de page bloque tous les threads
- Éventualité d'interblocages

On verra ultérieurement les mécanismes pour prendre en charge la concurrence

Autres approches

Entrelacement

Le principe est de permettre les deux modes de fonctionnement : exécuter n threads utilisateurs sur p threads noyau

Le programmeur est maître de ses priorités

Le noyau ignore l'importance de chaque thread noyau

Autres approches

Entrelacement

Le principe est de permettre les deux modes de fonctionnement : exécuter n threads utilisateurs sur p threads noyau
Le programmeur est maître de ses priorités
Le noyau ignore l'importance de chaque thread noyau

Threads spontanés

Il s'agit d'associer un événement à la création d'un thread
Cette approche est légère car les threads ainsi créés ont une durée de vie limité (exemple traiter un message arrivant sur le réseau)

Monothread → multithread

Sur le plan de l'utilisation la programmation multithread est bien plus complexe que l'approche monothread.

Réentrance

Cette notion caractérise le fait de pouvoir être utilisé *simultanément* par plusieurs tâches
Une fonction réentrante permet d'être appelée dans un programme multithread et de toujours fournir un résultat consistant

Dans ce type de programmation :

- soit mécanismes de synchronisation
- soit fonctions réentrantes