# Threads

Programmation multicore

Modèles de multithreading

# Cores, Processeurs, Multithreading

Il y un continuum entre ces différentes alternatives

Le SE peut généralement traiter un “hardware thread” comme un CPU

4 processeurs dual-core $\simeq$ 1 octa-core

# Motivations

Mêmes que communication entre processus:

- Pour profiter du parallélisme
- Pour des raisons de modularité
- Efficacité du partage d'information

Mais aussi

- Un *thread* est plus "léger" qu'un processus
- Des fois il n'y a pas de processus (e.g. à l'intérieur du noyau)
- Éviter de se bloquer sur une opération (*responsiveness*)

# Exemples d'usage

Serveur web: beaucoup de requêtes indépendantes

- Créer un processus pour chaque requête est inefficace
- Utiliser un seul thread empêche de profiter du parallélisme

Découpler les tâches indépendantes:

- S'occuper de l'interface utilisateur
- Chercher des données
- Faire de la correction orthographique
- Réorganiser les données pour optimiser les accès futurs

# Parallélisme vs concurrence

Termes cousins mais distincts

- *Parallélisme*: accélérer l'exécution d'une tâche en la divisant en éléments exécutables en même temps
- *Concurrence*: Décomposition en plusieurs tâches indépendantes

Types de parallélisme

- *parallélisme de données*: les données du calcul sont réparties entre les sous-tâches qui font toutes plus ou moins la même opération
- *parallélisme de tâches*: chaque tâche fait une opération distincte

# Programmation multithreaded

Avenir peu brillant pour la performance single-threaded

Moins difficile d'augmenter le nombre de processeurs

Plus de travail pour nous

- Diviser le travail en sous-tâches indépendantes
- Équilibre entre la quantité de travail des tâches
- Répartition des données pour éviter les conflits
- Synchronisation
- Tester, vérifier, déboguer

# Loi d'Amdahl

Les portions de code *séquentielles* sont déterminantes

$N$, nombre de processeurs; $S$, portion d'exécution séquentielle

$$\textit{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

75% d'exécution en parallèle sur 1024 processurs $\Rightarrow 3.99x$

# Threads noyau et threads utilisateur

Threads utilisées par une application viennent d'une librairie

Threads du noyau viennent... du noyau

Les deux sont en partie indépendants

- *One-to-one*: un thread de l'application = un thread du noyau
- *Many-to-one*: un seul thread noyau partagé
- *Many-to-many*: N threads application, M threads noyau
- *One-to-many*: usage interne de threads noyau

# *POSIX threads* (Pthreads)

Créer un nouveau thread:

```
err = pthread_create (&tid, attr, func, arg);
```

Terminer un thread:

```
pthread_exit (retval);
```

Attendre la fin d'un thread:

```
err = pthread_join (tid, &retval);
```

Demande de terminaison:

```
err = pthread_cancel (tid);
```

# Détails sémantiques

`fork` et `exec`: que faire avec les autres threads?

Mémoire *thread-local*

Intéractions avec la conversion user-thread $\leftrightarrow$ kernel-thread

Gestion des signaux

# Signaux POSIX

Utilisés pour notifier les processus d'événements particuliers

- Examples: SIGFPE, SIGCHILD, SIGSTOP, ...

Un *signal handler* répond à un *signal*

- Par défaut, tuer le processus
- Peut être bloqué, ou ignoré
- Peut appeler une fonction au choix du programme

Signal reçu de manière asynchrone, comme une interruption

Quel thread reçoit quels signaux? [ + user/kernel thread? ]