



UNIVERSITÉ
CÔTE D'AZUR

Processus et Threads

Présentation: Stéphane Lavirotte

Auteurs: ... et al*

**(*) Cours réalisé grâce aux documents de :
Stéphane Lavirotte, Christophe Morvan**

Mail: Stephane.Lavirotte@unice.fr

Web: <http://stephane.lavirotte.com/>

Université de Nice - Sophia Antipolis



Processus vs Threads



Rappel Notion de Processus

- ✓ **Entité dynamique**
 - Contexte d'exécution d'un programme séquentiel
 - Entité active opérant sur son environnement en exécutant un programme

- ✓ **Etat courant de la « *machine virtuelle* » exécutant le programme associé au processus**
 - Inclut l'état [d'une partie] de la machine physique

- ✓ **Entité d'attribution du CPU**



Pourquoi avoir des Processus ?

- ✓ **La programmation d'un processus**
 - ne tient pas compte de l'existence simultanée d'autres processus
 - y compris 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 processus interagissent avec:
 - La mémoire
 - Les unités de stockage
 - Le réseau
 - Les utilisateurs
 - Lorsqu'un processus attend des données, un autre peut s'exécuter
- ✓ **Les principes de processus et de parallélisation sont anciens**



Pourquoi `fork()` ?

✓ Observation

- Dans la plupart des cas, `fork` est suivi de l'appel à `execxx()`
- Certains systèmes utilisent une combinaison des deux appels

✓ Néanmoins

- Il est parfois utile de faire un simple `fork`
- Le gain le plus manifeste est celui de la simplicité
 - Globalement on souhaiterait manipuler plusieurs aspects du processus fils
 - Descripteur de fichier, variables d'environnement, droits, ...
 - `fork()` n'utilise aucun paramètre

✓ En l'absence de `fork`

- La création de processus nécessite une multitude de paramètres



Se passer de `fork()` ?

✓ Interface Win32 de création de processus

```
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  
);
```



Limitation des Processus

- ✓ **Un processus regroupe 2 concepts**
 - Des ressources (données, fichiers...)
 - Une exécution
- ✓ **Les processus sont une abstraction pour grouper des ressources**
- ✓ **Les processus induisent une certaine lourdeur**
 - Changement de contexte
 - Contexte lourd, coûteux à créer et à détruire
 - Pas efficace (changement de contexte lent)
 - Contexte mémoire important
 - Pas (peu) de partage de mémoire
 - Pas intégré dans les langages de programmation
 - Offerts par services OS complexes, pas souples (IPC)
 - Un processus pourrait tirer parti d'une architecture multiprocesseurs/multi-cœurs

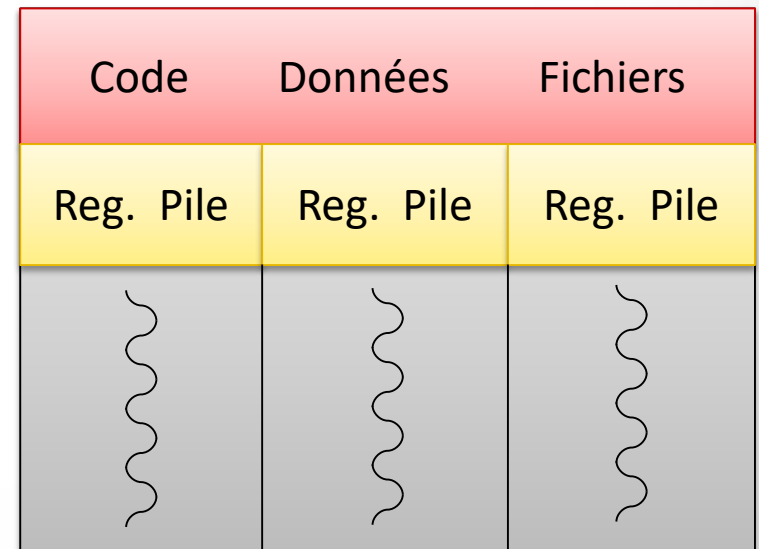
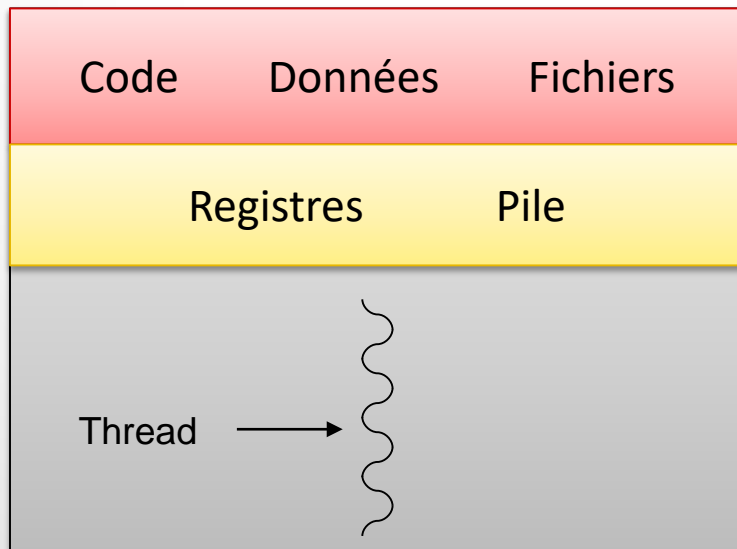


Introduction Notion de Threads

- ✓ **Un thread est une abstraction de l'exécution**
- ✓ **Notion combinant les avantages suivants :**
 - Exécution parallèle
 - Partage code et données applicatifs communs
- ✓ **Facile à mettre en œuvre**
 - Programmable
 - Contrôlable
 - Configurable
- ✓ **Efficace et passe à l'échelle**
- ✓ **Les Threads permettent donc :**
 - 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)



- ✓ **Un thread est un contexte d'exécution ordonnable**
 - Comprend un compteur ordinal, des registres et une pile
 - Chaque thread a son propre compteur de programme
 - Plusieurs threads partagent le même espace mémoire
 - Il n'y a pas de protections entre les threads d'un même processus
 - Les programmes simples comportent un seul thread → pas de surcoût





Données Processus vs Threads

✓ Données par processus

- Espace d'adressage
- Variables globales
- Fichiers ouverts
- Processus fils
- Alarmes en attente
- Signaux et gestionnaires de signaux
- Informations de comptabilité

✓ Données par thread

- Compteur d'instruction
- Registres
- Pile
- État



Intérêts des Threads

- ✓ **Dans une application, plusieurs activités s'exécutent en parallèle**
 - Les threads sont un bon modèle de programmation
 - Comme les processus
- ✓ **Les threads partagent le même espace mémoire**
 - Plus efficace que les mécanismes fournis par les OS (IPC)
 - Indispensable pour certaines applications
- ✓ **Les threads ont peu d'information propre**
 - Très faciles à créer/détruire
 - En général, 100 fois plus rapide à créer qu'un processus
- ✓ **Permettent de recouvrir le calcul et les I/Os**
 - Si les threads ne font que du calcul, peu d'intérêt
- ✓ **Très pratique sur les systèmes multi processeurs/cœurs**

Problèmes soulevés par les 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 partagées, 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



Mise en œuvre des Threads

Implémentation des
Threads
User Space / Kernel Space



User Space vs Kernel Space

user space vs. kernel space JULIA EVANS
@b0rk

drawings.jvns.ca

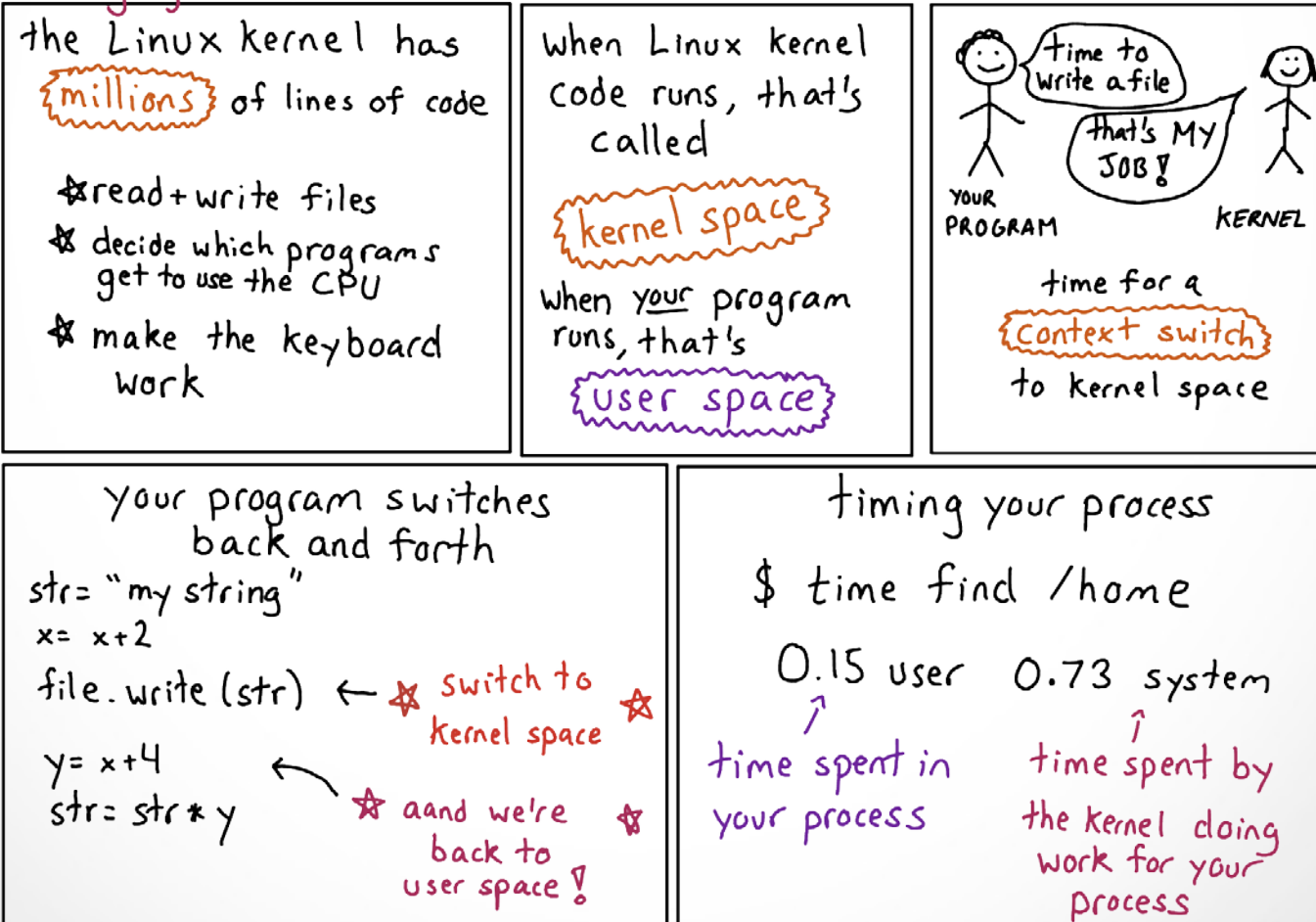
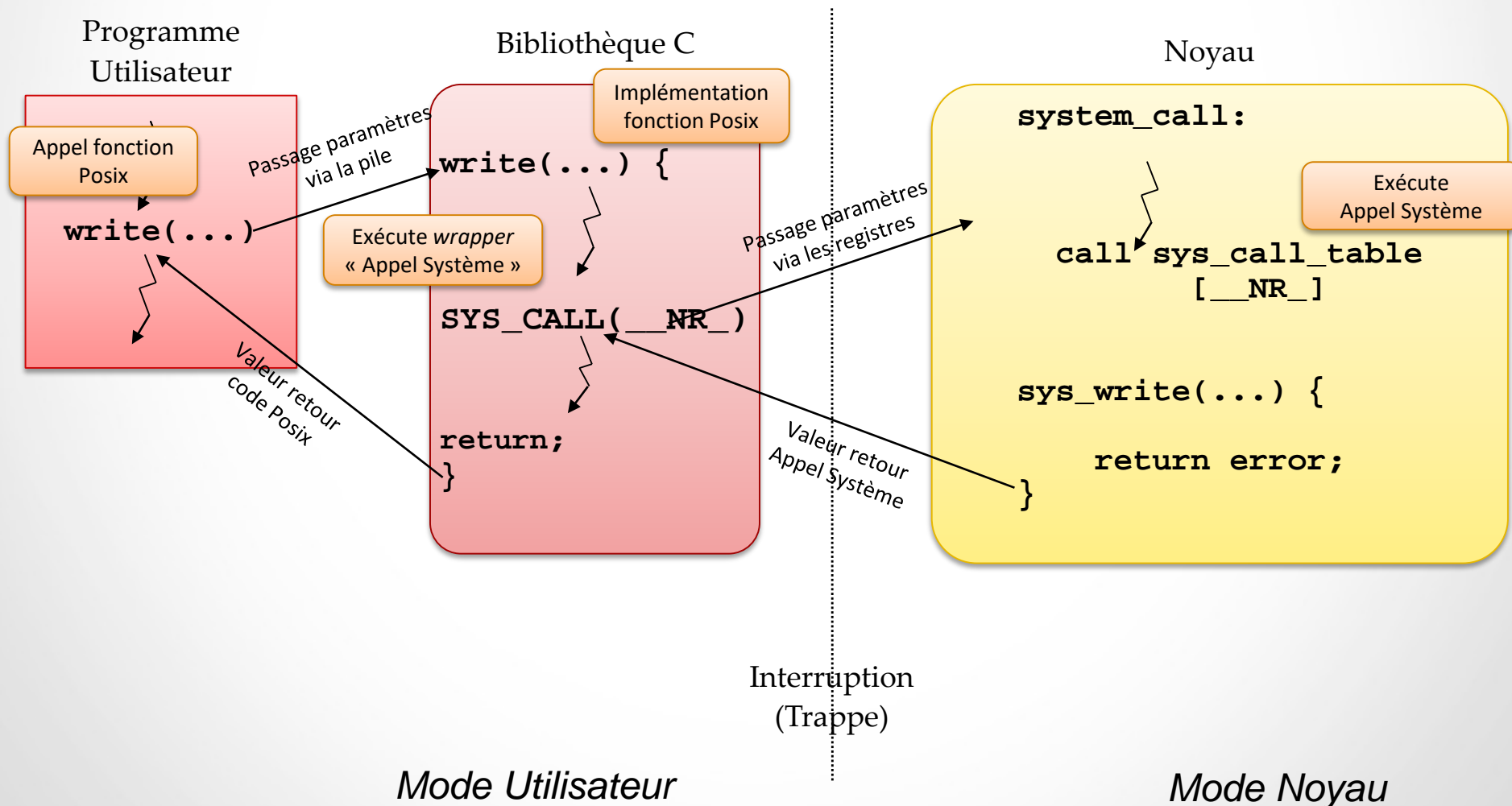


Illustration du User Space / Kernel Space



Implémentation des Threads en *User Space*

- ✓ 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
- ✓ Chaque processus a une table de threads
 - Contient les informations pour les threads du processus
 - Mise à jour quand changement d'état
- ✓ Si un thread va faire une opération potentiellement bloquante
 - Il appelle une méthode spéciale
 - Cette méthode vérifie si le thread doit être mis en attente
 - Si oui, modification de la table, recherche d'un autre thread, chargement du PC et registres...



Avantages et Inconvénients des Threads en *User Space*

✓ Avantages

- Changement de thread (« *thread switching* ») très rapide
 - Pas de passage en *kernel space*
- Permet à un programme d'avoir son propre ordonnanceur

✓ Inconvénients

- N'utilise pas les multiprocesseurs
- Les appels systèmes bloquant bloquent tous les threads
 - Ex: lire une touche d'un clavier, peut être bloquant ou pas
 - Solution : wrapper les appels système pour les rendre non bloquants
 - Il est possible d'utiliser ses propres fonctions pour les accès réseau
 - en général c'est impossible pour accéder au disque
- Un défaut de page bloque tous les threads
- Eventualité d'inter-blocages, les threads doivent collaborer
 - `thread_yield`

✓ En pratique beaucoup de travail/contraintes pour gérer le blocage et l'ordonnancement

- Alors que c'est l'intérêt des threads

Implémentation des Threads en *Kernel Space*

- ✓ Le noyau gère les threads
 - Il a une table des threads dans le système
- ✓ La création et la destruction sont effectués par le noyau à travers des appels système
 - Plus coûteux qu'en user space
 - Utilisation de pools de threads
- ✓ Quand un thread bloque, le noyau peut choisir un autre thread
 - du même processus
 - Ou pas

Avantages et Inconvénients des Threads en *Kernel Space*

✓ Avantages

- Les appels systèmes bloquants ne bloquent que le thread concerné

✓ Inconvénients

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



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 limitée (exemple traiter un message arrivant sur le réseau)



Threads POSIX

- ✓ **La norme POSIX définit l'API pthreads (POSIX 1.c)**

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

- ✓ **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**



Pthread sous Linux

- ✓ 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 une implémentation plus performante
 - Conforme à POSIX
 - Avec NPTL tous les threads d'un même processus ont le même identifiant



Conclusion

- ✓ **Sur le plan de l'utilisation la programmation multithread est bien plus complexe que l'approche mono-thread**
- ✓ **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