UNIVERSITÉ PARIS SUD

IUT D'ORSAY

M3101 · Principes des systèmes d'exploitation

Threads

IUT d'Orsay · DUT Informatique · 2015 / 2016

Les threads

Threads (processus légers) unités d'exécution dans un processus

- exécutent le même programme
- utilisent les piles indépendantes — plusieurs flux d'exécution
- partagent la mémoire du processus — code, données, tas...
- partagent les fichiers ouverts, les gestionnaires de signaux...

La mémoire est partagée

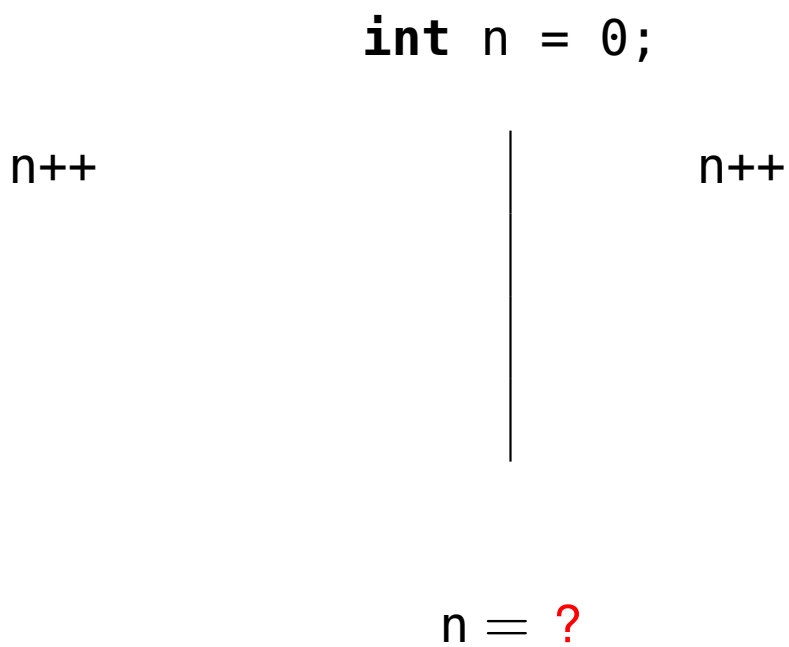
- création légèrement plus rapide

  pas besoin de copier la table de pages
- changement de contexte légèrement plus rapide

  pas besoin de remettre à zero le TLB (*Translation Lookaside Buffer*)
- communication beaucoup plus rapide

  pas besoin de passer par des tubes, sockets, ...
- programmation beaucoup plus délicate

Mémoire partagée



Mémoire partagée

```
int n = 0;
```

`load n`	`load n`
`store n+1`	`store n+1`

$n = ?$

Mémoire partagée

```
int n = 0;
```

`load 0`	
`store 1`	
	`load 1`
	`store 2`

$n = 2$

Mémoire partagée

```
int n = 0;
```

`load 0`	
	`load 0`
	`store 1`
`store 1`	

$n = 1$

Mémoire partagée

```
int n = 0;
```

```
for (i=0; i<100; i++)
    n++;
```

```
for (i=0; i<100; i++)
    n++;
```

$n = ?$

Mémoire partagée

```
int n = 0;
```

`load 0`	
	`load 0`
	`store 1`
	…
	`load 98`
	`store 99`
`store 1`	
	`load 1`
`load 1`	
`store 2`	
…	
`load 99`	
`store 100`	
	`store 2`

$n = 2$

Type **pthread_t** — analogue de PID, type opaque

```
pthread_t pthread_self(void);
```

Renvoie l'identifiant du thread courant

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Renvoie une valeur non-nulle si `tid1` est égal à `tid2`, sinon 0

```
int pthread_create(
  pthread_t *tid,                  /* identité du thread */
  NULL,                            /* attributs du thread */
  void* (*routine) (void*),        /* routine de départ */
  void* arg);                      /* argument de routine() */
```

Crée un nouveau thread qui exécutera `routine(arg)` :

```
void* routine(void* arg) { /* le code du thread */ }
```

Écrit dans `tid` l'identifiant du nouveau thread

Renvoie 0 au succès, sinon un code d'erreur non-nul

```
void pthread_exit(void * valeur_de_retour);
```

Termine le thread courant

Équivalent au **return** de la routine de départ

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

Attend la terminaison du thread d'identifiant `tid`

Retourne immédiatement si `tid` a déjà terminé

Met dans `*retval` la valeur de retour du thread

On peut appeler `pthread_join(tid, NULL)` si on n'en a pas besoin

Libère les ressources du thread terminé

Renvoie 0 au succès, sinon un code d'erreur non-nul

Exclusion mutuelle

Accès concurrent et non-atomique aux données partagées $\Rightarrow$ désastre

Solution : rendre atomique toute opération sur les données partagées

Outil : objet d'exclusion mutuelle — mutex

Peut être en deux états :

- pris par un (et un seul) thread — verrouillé
- pris par aucun thread — déverrouillé

Tout thread qui souhaite prendre un mutex verrouillé doit attendre

Rend possible l'utilisation de sections critiques

> partie du code qui ne peut pas être exécutée
> par plusieurs threads en même temps

Initialisation et destruction de mutex

```
int pthread_mutex_init(pthread_mutex_t * mutex, NULL);
```

Le deuxième paramètre indique les attributs du mutex (NULL par défaut)

```
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Attention : le mutex doit être déverrouillé

Renvoient 0 au succès, sinon un code d'erreur non-nul

Prise et libération de mutex

```
int pthread_mutex_lock(pthread_mutex_t * mutex);
```

Verrouille le mutex

Si mutex est déjà verrouillé par un autre thread

$\Rightarrow$ suspend le thread appelant jusqu'au déverrouillage du mutex

Si mutex est déverrouillé

$\Rightarrow$ retourne immédiatement

```
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

Déverrouille le mutex

Attention : le mutex doit être verrouillé par le thread appelant

Renvoient 0 au succès, sinon un code d'erreur non-nul

Accès protégé

```
int n = 0;
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

```
for (i=0; i<100; i++) {
  pthread_mutex_lock(&m);
  n++;
  pthread_mutex_unlock(&m);
}
```

```
for (i=0; i<100; i++) {
  pthread_mutex_lock(&m);
  n++;
  pthread_mutex_unlock(&m);
}
```

n = ?

Accès protégé

```
int n = 0;
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

load 0 **store** 1	
	load 1 **store** 2
	load 2 **store** 3
load 3 **store** 4	
...	...

$n = 200$

Écueils de l'exclusion mutuelle

Exclusion mutuelle coûte cher :

- les primitives de synchronisation sont coûteuses
- tue le parallélisme de l'application

  les threads passent leur temps en attente

Une implémentation incorrecte provoque des erreurs graves :

- famine (*starvation*) : un ou plusieurs threads n'avancent pas

  car toujours doublés par d'autres threads
- interblocage (*deadlock*) : plusieurs threads se bloquent mutuellement

  cf. le problème de « repas de philosophes »