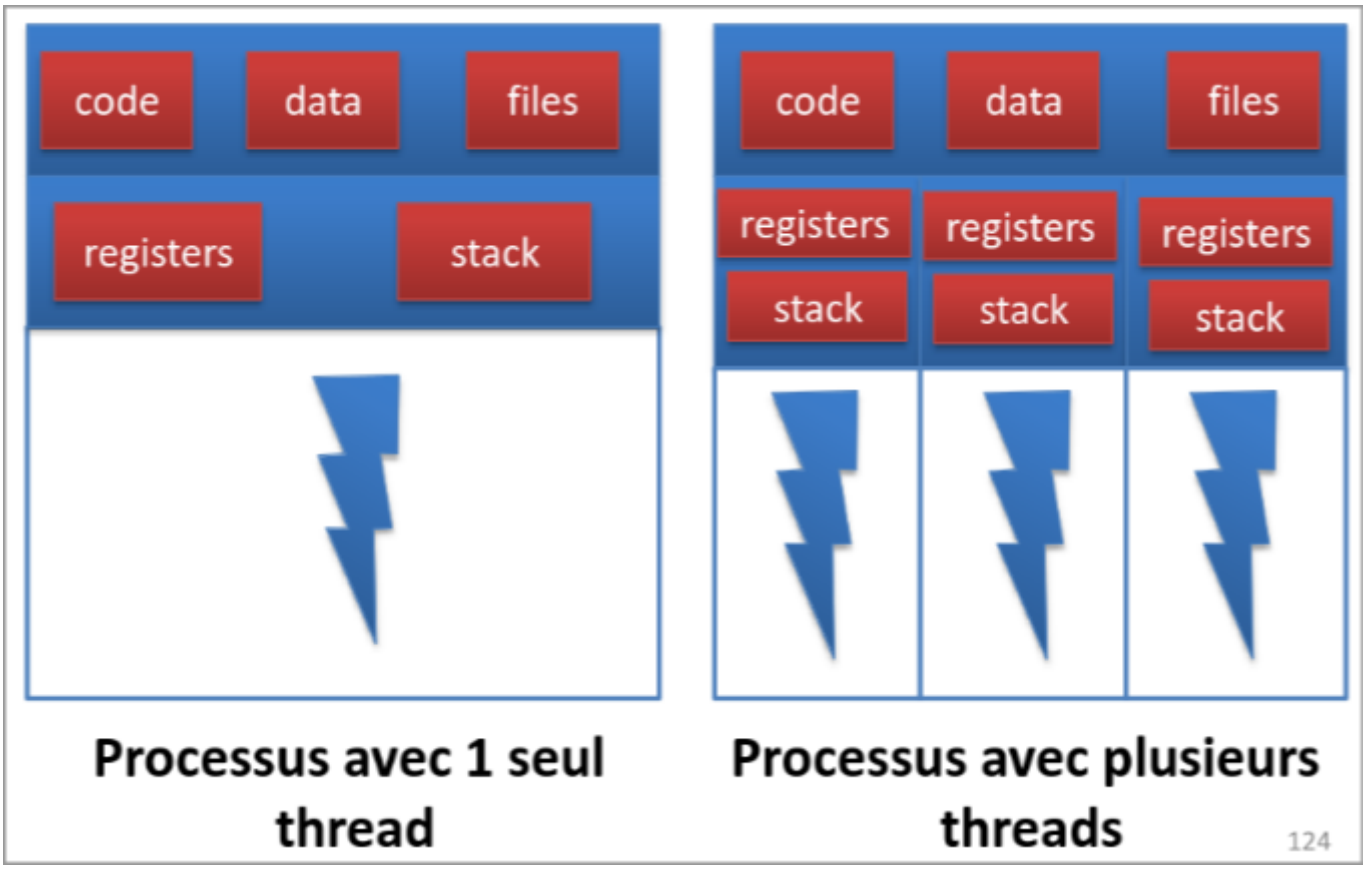
Les threads

Les processus que l'on a vu n'avait qu'un seul fil d'exécution (monothread) mais il est possible d'avoir un processus avec plusieurs fils d'exécutions (multithread).

Les threads sont en somme des sortes de "mini processus".

Avantages

Contrairement aux processus il est beaucoup plus rapide d'en créer un nouveau, également les threads d'un même processus partagent les informations. En plus sur un système avec plusieurs coeurs l'exécution des threads d'un même processus peut se faire en parallèle ce qui offre une performance intéressante.



Exemple

Par exemple on pourrait avoir un thread utilisé pour une saisie de texte, un autre thread pour l'affichage et encore un dernier thread pour vérifier les informations reçues.

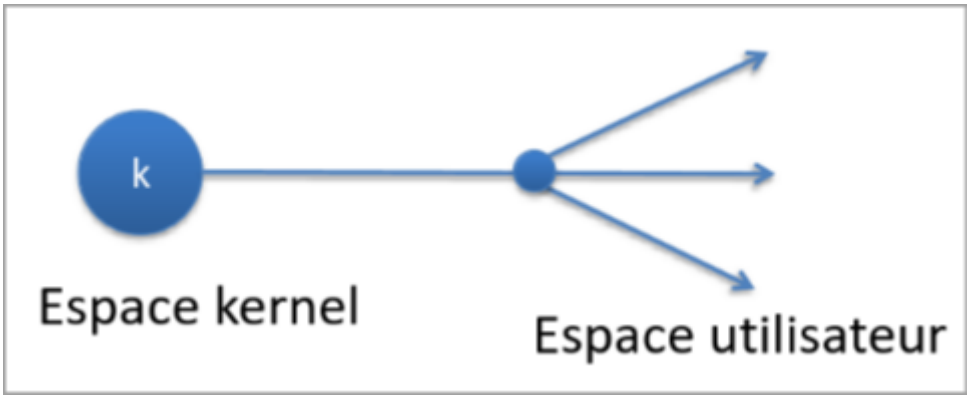
Modèles d'implémentations

Les threads peuvent être implémentés à deux niveaux :

- Dans **l'espace kernel**, il est alors pris en charge nativement par le système d'exploitation au même titre que les processus
- Dans **l'espace utilisateur**, il est alors supporté au travers de libraries externe

Les threads peuvent être implémentés selon plusiuers modèles :

Plusieurs à un

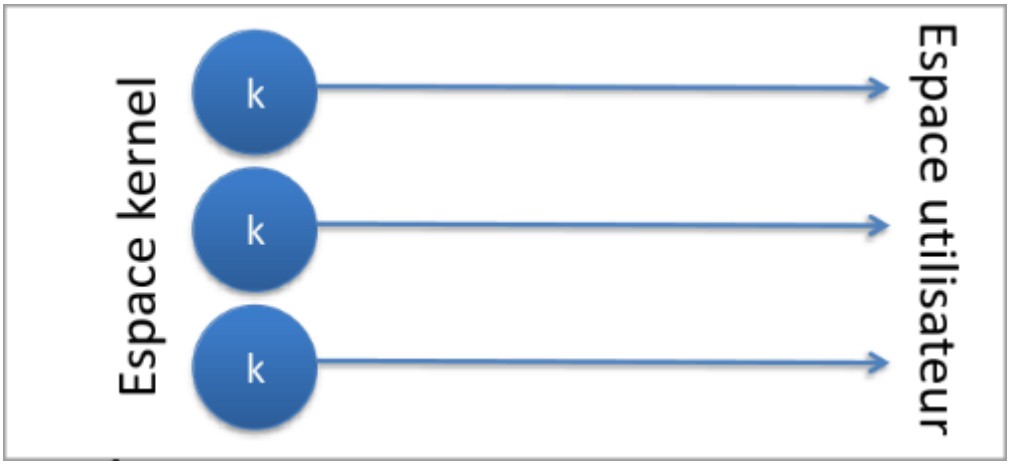


Dans ce modèle les threads sont supporté par une librarie externe, le système d'exploitation n'en a donc aucune connaissance et ne vois que le processus.

L'avantage est que sa création est rapide, cependant les inconvénients sont que un seul thread (le processus) est vu par le système, le scheduler du système n'est donc pas adapté. Si un thread réalse une opération bloquante, cela risque d'empécher tous les autres threads de travailler.

Enfin cette implémentation n'est plus vraiment courrante car elle n'est pas adaptées aux CPU multi-coeurs.

Un à un



Dans ce modèle chaque thread utilisateur est attaché à un thread kernel. Ainsi les threads sont complètement géré au niveau du système d'exploitation.

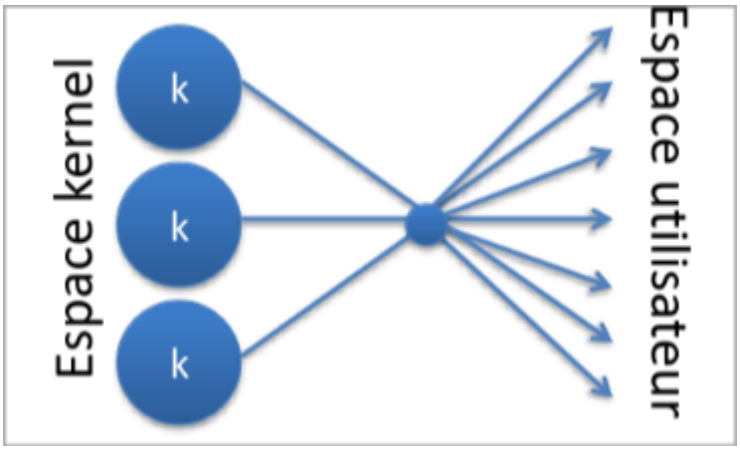
Cela a l'avantage de créer un scheduling plus avantageux et d'être compatible avec les processeurs multi-coeurs.

Cependant ce modèle est couteux pour le système car c'est lui qui doit tout gérer.

C'est ce modèle qui est nottament utilisé dans Linux. Voici par exemple la liste des threads associés au processus Firefox sur mon système.

```
[snowcode@snowcode:~]$ ps -T -p 608449
    PID    SPID TTY          TIME CMD
 608449  608449 tty1     00:00:00 Web Content
 608449  608453 tty1     00:00:00 IPC I/O Child
 608449  608455 tty1     00:00:00 Socket Thread
 608449  608456 tty1     00:00:00 HTML5 Parser
 608449  608457 tty1     00:00:00 JS Watchdog
 608449  608458 tty1     00:00:00 Backgro~Pool #1
 608449  608459 tty1     00:00:00 Timer
 608449  608463 tty1     00:00:00 RemVidChild
 608449  608464 tty1     00:00:00 ImageIO
 608449  608465 tty1     00:00:00 ImageBridgeChld
 608449  608466 tty1     00:00:00 RemoteLzyStream
 608449  608467 tty1     00:00:00 ProcessHangMon
 608449  608468 tty1     00:00:00 ProfilerChild
```

Plusieurs à plusieurs



L'idée du plusieurs à plusieurs est de créer un *pool* de thread au quel les threads utilisateurs vont être assigné à la volée au cours de l'exécution.

De cette façon cela combine les avantages des deux modèles précédents. Cependant ce modèle est assez peu courrant car il nécessite que le sysètme d'exploitation soit construit autour de ce modèle car il est plus complexe à gérer que les autres.

Problèmes

Il y a quelques difficultés à considérer pour les threads. Par exemple :

- Que se passe-t-il en cas de `fork()` dans un thread ? Certains OS vont dupliquer tous les threads, d'autres ne vont pas le faire.
- Et avec `execl()` ? L'appel execl remplace le code du processus pour charger celui d'un autre. Ainsi le code remplace tous les threads du processus
- Pour terminer l'exécution d'un thread il y a deux possibilités, dans tous les cas il faut faire très attention pour la libération des ressources
  - Le faire de manière **asynchrone**, un thread demande la terminaison d'un autre (cela est cependant rare)
  - Le faire de manière **différée**, chaque thread vérifie régulièrement s'il doit continuer ou s'arrêter
- Quand un signal est envoyé à un processus, quels threads recoivent le signal ? Tous, certains ou un en particulier ? Cela dépends du type de signal et cela est encore une fois pas comment dans tous les OS.

Librarie

Pour créer des threads dans les systèmes UNIX il existe la librarie standard `pthread` dont voici quelques fonctions intéressantes :

- `pthread_attr_init` qui permet de fixer certains attributs, mais pas utile dans le cours
- `pthread_create` pour créer et démarrer un nouveau thread
- `pthread_join` pour attendre la mort d'un thread
- `pthread_exit` pour terminer l'exécution d'un thread, cette fonction permet aussi de retourner une valeur de retour à pthread join via un pointeur générique `void*`

Exemple

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

void* thread1(void* args) {
  int i;

  /* On trasforme le void* args en pointeur de int avec un cast */
  int* resultat;
  resultat = (int*)args;

  for (i = 0; i < 10; i++) {
    printf("[THREAD 1] %d\n", i);
```

```
    /* On ajoute le nombre courrant au résultat, attention à ne pas oublier de déréferencer le pointeur */
    *resultat += i;
  }

  return resultat;
}

void* thread2(void* args) {
  int i;

  /* On trasforme le void* args en pointeur de int avec un cast */
  int* resultat;
  resultat = (int*)args;

  for (i = 0; i < 24; i++) {
    printf("[THREAD 2] %d\n", i);
    /* On ajoute le nombre courrant au résultat, attention à ne pas oublier de déréferencer le pointeur */
    *resultat += i;
  }

  return resultat;
}

int main(void) {
  pthread_t tid1, tid2;

  /* On initialise les résultats à 0 */
  int resultat1 = 0;
  int resultat2 = 0;

  /* Création des threads auquels on passe les pointeurs vers les variables resultat1 et resultat2 */
  pthread_create(&tid1, NULL, *thread1, &resultat1);
  pthread_create(&tid2, NULL, *thread2, &resultat2);

  /* On attends que tous les tests se finissent */
  /* Nous n'avons pas besoin ici de récupérer la valeur de retour car on a toujours accès aux variables dont on a passé les pointeurs plus tôt, surtout que cela rends les choses très compliquées de manipuler des void** (pointeur de pointeur de valeur de type inconnue) */
```

```
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  /* Nous pouvons ensuite simplement récupérer les valeurs des résultats */
  printf("Résultat du thread 1 = %d\n", resultat1);
  printf("Résultat du thread 2 = %d\n", resultat2);

  return EXIT_SUCCESS;
}
```

Revision #2
Created 2 January 2024 15:55:03 by SnowCode
Updated 2 January 2024 15:56:05 by SnowCode