Reconfigurable & embedded Digital Systems

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Systèmes d'exploitation (SYE)
Threads

Profs Daniel Rossier, Alberto Dassatti, Salvatore Valenza

Version 3.1 (2017-2018)

Plan

- Caractéristiques d'un *thread*
- *Threads POSIX*
- Types de *thread*

Caractéristiques d'un *thread* (1/3)

- Un processus peut contenir **un** ou **plusieurs** contextes d'exécution.
  - Un tel contexte d'exécution est appelé ***thread***.

Hyper-Threading

Cours SYE - Institut REDS/HEIG-VD

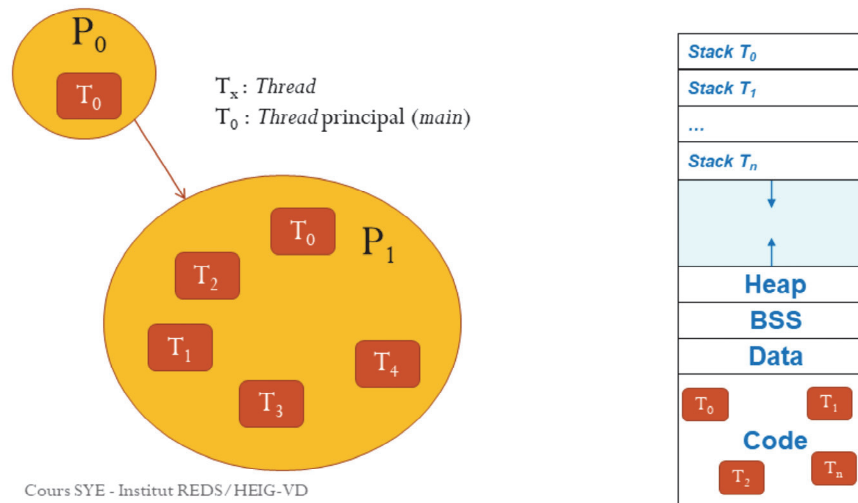
3

Nous avons vu lors de chapitres précédents que la notion de processus permettait de découper une application en plusieurs contextes d'exécution et contextes mémoire afin de *paralléliser* les traitements. Typiquement, le lancement de plusieurs instances d'un navigateur entraîne l'apparition de plusieurs processus, bien qu'il s'agisse d'une même application.

Cependant, la création et la gestion de processus au niveau du noyau est coûteuse (création: ~250 microsecondes). C'est pourquoi, il existe une autre approche de paralléliser les traitements avec la notion de *thread* (ou processus léger ou encore *fil d'exécution* – à noter que le terme *thread* est largement préféré et utilisé par les francophones!).

La notion de *thread* est également connue au niveau du matériel (processeur) avec le terme de *hyperthreading*. Ce néologisme veut simplement dire que la notion de *thread* n'est pas gérée d'une manière purement logicielle, mais qu'il existe un support matériel à l'intérieur d'un processeur pour gérer les *threads* de manière performante, notamment au niveau de la commutation entre eux et de la gestion des *caches*. La possibilité de gérer plusieurs *threads* au niveau matériel a dérivée sur le concept de *processeur logique*. Un processeur supportant l'*hyperthreading* est donc reconnu comme ayant (généralement) deux cœurs de calcul logiques, *CPU #0* et *CPU #1*.

La création d'un *thread* prend ~25 microsecondes, soit 10 fois moins qu'un processus.

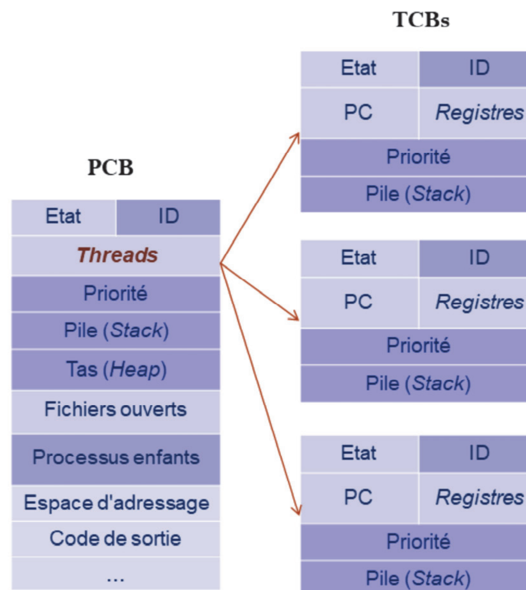


Comme un *thread* représente un contexte d'exécution au sein d'un processus, celui-ci doit pouvoir accéder à toutes les ressources allouées au processus (mémoire, ressources logiques, etc.). Il en va de même pour tous les autres *threads* générés à l'intérieur d'un même processus. C'est dire que tous les *threads* d'un processus ont **accès au même espace d'adressage**, en particulier les différentes *sections* du processus sont partagés, **à l'exception** de la pile qui fait partie du contexte d'exécution. En effet, chaque contexte d'exécution, donc chaque *thread*, doit disposer de sa propre pile afin de pouvoir s'exécuter correctement. En revanche, le tas (*heap*) est partagé par l'ensemble des *threads*, ainsi que les données se trouvant dans les sections *data* et *bss*. La section code contiendra alors le code *utilisé* par chaque *thread*.

Un bon exemple pour comprendre l'utilisation des *threads* est celui du traitement de texte. Durant la saisie, le traitement de texte *lance* un correcteur orthographique qui soulignera les fautes. La tâche de correcteur et la tâche de saisie sont deux tâches indépendantes, mais qui doivent accéder au même contenu; les deux *threads* associés à ces deux tâches devront gérer les accès concurrents qui pourront subvenir, en ayant recours à des objets de synchronisation classiques *(mutex,* sémaphore, etc.). Il est important de réaliser que les deux *threads* appartiennent au même processus.

Caractéristiques d'un *thread* (3/3)

- ***Thread Control Block*** *(TCB)*
  - Fiche signalétique du *thread*
- Informations du *TCB*
  - **Pointeur d'instruction** (PC)
  - **Registres** (données)
  - **Pointeur de pile**
  - Un **état**
  - Une **priorité**



5 Cours SYE - Institut REDS / HEIG-VD

A l'image du processus qui dispose de son *PCB* (fiche signalétique), chaque *thread* est lié à un ***TCB*** (*Thread/Task Control Block*). Il s'agit d'une structure de données contenant les informations propres au *thread*, comme son état, sa priorité, le pointeur de pile, etc.

Le ***TCB*** est fortement sollicité lors d'un changement de contexte entre *thread*.

Par ailleurs, on remarque que le *TCB* est beaucoup plus petit que le *PCB*; il ne contient aucune information particulière concernant les ressources qui pourraient être utilisées par le *thread,* comme des fichiers ouverts ou d'autres objets de synchronisation ou de communication.

Threads POSIX (1/6)

- ***Portable Operating System Interface*** *(X pour uniX/linuX/macosX)*
- API standardisée (IEEE 1003.1c) largement répandue pour les *threads*

- Création/terminaison
- Synchronisation
- Accès concurrents

- Supporté par Windows (sous-système POSIX)

6 Cours SYE - Institut REDS/HEIG-VD

La programmation des *threads* dans le langage C repose principalement sur une librairie connue sous le nom de *threads* ***POSIX***. *POSIX* est le nom d'une famille de standards définie depuis 1988 par *l'Institute of Electrical and Electronics Engineers* (*IEEE*) formellement désignée par *IEEE 1003*. Ces standards ont émergé d'un projet de standardisation des API (*Application Programming Interface*) des logiciels destinés à fonctionner sur des variantes du système d'exploitation UNIX.

Aujourd'hui, POSIX est présent dans la plupart des OS modernes, y compris les RTOS (*Realtime OS*). Les *threads* POSIX sont supportés sous *Linux, Windows, Mac OS X, etc.* La librairie définie un ensemble de fonctions permettant la création, la terminaison, la synchronisation des *threads*, ainsi que plusieurs fonctions permettant la gestion des accès concurrents sur des variables partagées. Ces fonctions utilisent des mécanismes comme des *mutex,* sémaphores, variables condition, etc. Ces différents mécanismes ne seront pas étudiés dans ce cours, mais seront approfondis dans un cours de *programmation concurrente*.

Les fonctions POSIX sont définies dans une librairie utilisateur où chaque fonction de manipulation de *thread* repose sur des appels système dépendants de l'OS. La librairie des *threads* POSIX constituent donc une couche d'abstraction idéale pour garantir une bonne portabilité de l'application (attention! les fonctions de gestion des *threads* POSIX ne sont pas directement des appels système).

Threads POSIX (2/6)

- **Création** d'un *thread*
  - La fonction "*threadée*" doit correspondre au prototype suivant:

```
void *start_routine(void *) { }
```

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- **Synchronisation** sur un *thread*

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

- **Terminaison** d'un *thread*
  - Retour de fonction (pas de valeur particulière à transmettre)
  - Fonction *pthread_exit()*

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

7 Cours SYE - Institut REDS/HEIG-VD

On se rappelle que les *threads* sont des contextes d'exécution à l'intérieur d'un processus. Un *thread* est directement lié à une **fonction C**. On dit que la fonction est *threadée* ou que l'on "*thread*" une fonction. Cette fonction constitue le point d'entrée du *thread* et le retour de la fonction (ou l'appel à la fonction *pthread_exit()*) terminera aussitôt le *thread*.
On peut considérer que tout processus dispose au minimum d'un *thread*: c'est le ***thread* principal** associé à la fonction *main()*; il s'agit du premier contexte d'exécution du processus.
La fonction ***pthread_create()*** permet la création d'un nouveau *thread* (à l'intérieur du processus qui exécute cette fonction). Cette fonction permet de passer une référence vers l'identifiant du *thread*, divers attributs (priorité, politique de synchronisation, etc**.), un pointeur de la fonction à *threader***, ainsi que d'éventuels arguments à transmettre à la fonction *threadée*.
La synchronisation entre *thread* (notamment entre le *thread* principal et les *threads* créés) s'effectuent avec la fonction ***pthread_join()***. Cette fonction met en attente le *thread* appelant jusqu'à ce que le *thread* à "joindre" **se termine**.
La fonction ***ptread_exit()*** permet à un *thread* de se terminer et de transmettre une valeur éventuelle au *thread* effectuant le *join*. L'appel à cette fonction n'est pas obligatoire; le retour de la fonction *threadée* provoquera un appel implicite à *pthread_exit()* en passant *NULL* comme argument. En outre, l'utilisation de *pthread_exit()* dans la fonction principale aura pour effet d'attendre la terminaison de tous les *threads* créés dans le processus; un appel implicite à l'appel système *exit(0)* sera ensuite effectué pour terminer le processus.

Threads POSIX (3/6)

```c
 2  #include <string.h>
 3  #include <pthread.h>
 4  #include <stdio.h>
 5
 6  int count = 0;
 7
 8  void *printHello(void *args) {
 9    char *th_name;
10    int i;
11
12    th_name = (char *) args;
13
14    printf("Hello! It's me, I am thread %s.\n", th_name);
15
16    for (i = 0; i < 100000; i++)
17      count++;
18
19    return NULL;
20  }
21
22  int main (int argc, char *argv[]) {
23    pthread_t hello_thread;
24    int ret;
25    char th_name[20];
26
27    strcpy(th_name, "Thread SYE");
28
29    ret = pthread_create(&hello_thread, NULL, printHello, (void *) th_name);
30    if (ret) {
31      printf("ERROR; return code from pthread_create() is %d\n", ret);
32      exit(-1);
33    }
34
35    pthread_join(hello_thread, NULL);
36
37    printf("count = %d\n", count);
38
39    exit(0);
40  }
```

Descripteur associé au *thread*

8 Cours SYE - Institut REDS/HEIG-VD

L'exemple ci-dessus illustre l'utilisation des *threads* avec les fonctions *POSIX*. La fonction *main()* – donc le *thread* principal – crée un nouveau *thread*. Le démarrage de ce nouveau *thread* est effectif au retour de la fonction *pthread_create()*. Nous verrons plus tard, dans le chapitre consacré aux ordonnancements, que les *threads* sont sélectionnés par l'ordonnanceur et peuvent s'exécuter dès que ceux-ci sont prêts (à être exécutés). Le programmeur ne peut faire aucune hypothèse sur les instants d'exécution des *threads*, à moins de "jouer" avec leurs priorités. Dans ce programme, il y aura donc deux *threads* : le *thread* principal et le *thread* nouvellement créé.

Dans cet exemple, la fonction *main()* effectue un appel à *pthread_join()* afin d'attendre le *thread* créé. On remarque que le second argument de cette fonction est le pointeur *NULL*: ce qui signifie que l'on n'attend aucun retour particulier du *thread.* En effet, la fonction *threadée printHello()* retourne la valeur *NULL* et ne fait pas utilisation de *pthread_exit()*.

En revanche, il n'aurait pas été judicieux de remplacer cet appel à *pthread_join()* par *pthread_exit()* car la fonction doit poursuivre son exécution par l'affichage de la valeur *count*. Un appel à *pthread_exit()* dans la fonction *main()* aurait bien eu comme effet l'attente de la terminaison du *thread* créé*,* mais le processus aurait alors terminé son exécution avant l'exécution du *printf()*.

Threads POSIX (4/6)

```c
 2 #include <string.h>
 3 #include <pthread.h>
 4 #include <stdio.h>
 5
 6 int count = 0;
 7
 8 void *printHello(void *args) {
 9   char *th_name;
10   int i;
11
12   th_name = (char *) args;
13
14   printf("Hello! It's me, I am thread %s.\n", th_name);
15
16   for (i = 0; i < 100000; i++)
17     count++;
18
19   return NULL;
20 }
21
22 int main (int argc, char *argv[]) {
23   pthread_t hello_thread1, hello_thread2;
24   char th_name1[20], th_name2[20];
25
26   strcpy(th_name1, "Thread SYE 1");
27   strcpy(th_name2, "Thread SYE 2");
28
29   pthread_create(&hello_thread1, NULL, printHello, (void *) th_name1);
30   pthread_create(&hello_thread2, NULL, printHello, (void *) th_name2);
31
32   pthread_join(hello_thread1, NULL);
33   pthread_join(hello_thread2, NULL);
34
35   printf("count = %d\n", count);
36
37   exit(0);
38 }
```

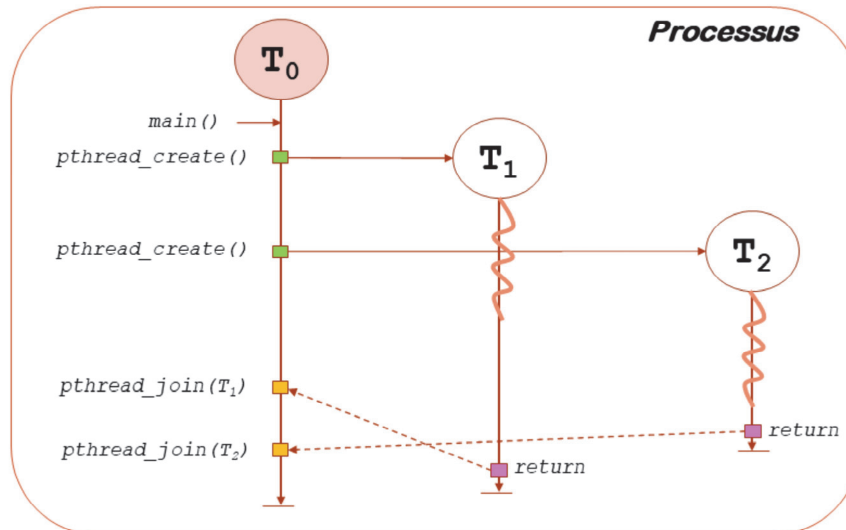
9 Cours SYE - Institut REDS/HEIG-VD

Le code précédent a été modifié afin de créer cette fois-ci un *thread* supplémentaire. Pour faciliter la lecture, les tests effectués au retour de *pthread_create()* ont été supprimés (bien qu'il soit indispensable de le faire).

Les deux *threads* vont ainsi s'exécuter quasi-simultanément, à tour de rôle. Une fois de plus, c'est l'ordonnanceur qui procédera au changement de contexte entre les *treads*.

Les deux appels à la fonction *pthread_join()* permet au *thread* principal d'attendre la fin de l'exécution des deux *threads* créés.

Ce code ne fonctionnera pas normalement. Pourquoi ?



Le diagramme de séquence (simplifié) de l'exemple précédent montre clairement les interactions entre les *threads.* De tels diagrammes sont très utiles et s'avèrent nécessaires lorsque plusieurs *threads* sont démarrés et doivent se synchroniser à un moment ou à un autre.

Dans cet exemple, on remarque le *thread* T_2 se termine *avant* le *thread* T_1. Mais, le premier appel à *pthread_join()* concerne bien le premier *thread*. Par conséquent, lorsque le *thread* T_2 se termine, le *thread* principal reste *bloqué* jusqu'à ce que le *thread* T_1 se termine. Le second appel à *pthread_join()* sera fera **sans suspension** du *thread* principal, le *thread* T_2 ayant terminé son exécution, la fonction retournera immédiatement.

Threads POSIX (6/6)

- Exécution…
  - Mais que se passe-t-il ?

```
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 123001
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 200000
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 200000
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 200000
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 200000
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 149549
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 123501
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 200000
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 200000
rossier@DRE-PC:~/sye/sye_sol/src$ ./app
Hello! It's me, I am thread Thread SYE 1.
Hello! It's me, I am thread Thread SYE 2.
count = 108533
rossier@DRE-PC:~/sye/sye_sol/src$
```

11 Cours SYE - Institut REDS/HEIG-VD

L'exécution de l'exemple précédent nous montre un résultat étonnant!

En effet, l'affichage montre des problèmes à l'exécution: les valeurs affichées par les *threads* ne sont pas celle auxquelles on s'attend, et varient d'une exécution à l'autre. Le compteur qui devrait afficher tout le temps la même valeur contient des valeurs qui semblent arbitraires.

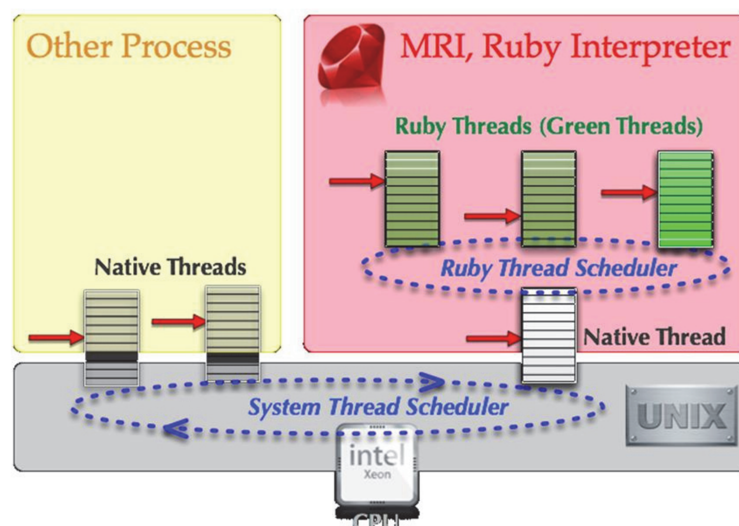
Ces problèmes sont directement liés à l'utilisation ***concurrente*** de variables partagées entre les *threads*. Dans un cas, on passe au *thread* l'adresse d'une variable locale utilisée par une boucle. On comprend aisément que la valeur de cette variable peut changer avant même que le *thread* n'ait eu le temps de la lire. Dans l'autre cas, c'est une variable globale qui sera modifiée tantôt par l'un, tantôt par l'autre. Le problème est qu'une telle affectation au niveau de la variable *count* peut être interrompue pendant son traitement, entre la lecture de la valeur actuelle et l'écriture de la nouvelle valeur (même s'il pourrait s'agit d'une instruction au niveau du langage C, cette affectation fait intervenir plusieurs instructions-machine et ne peut être en aucun cas considérée comme étant ***atomique***).
C'est un problème classique d'accès concurrent qui doit être géré à l'aide de mécanismes spécifiques. Ce genre de problèmes et les solutions liées seront étudiées en détails lors d'un cours de programmation concurrente.

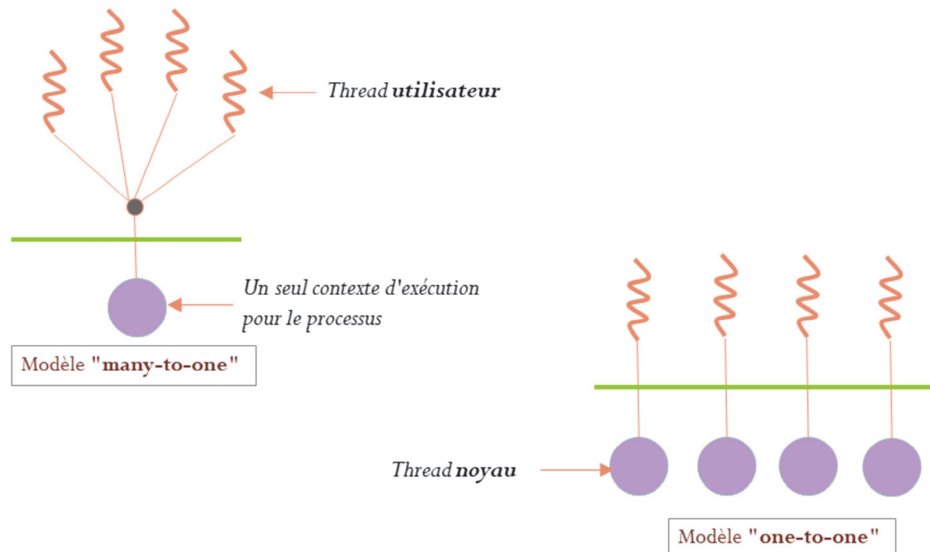
Types de thread (1/2)

- **Thread utilisateur**
  - La gestion des *threads* s'effectue complètement dans l'espace utilisateur.
  - Différents types de *threads* peuvent coexister.
    - Chaque processus peut utiliser des librairies de *threads* différents.
  - *Performance accrue*
- **Thread noyau**
  - La gestion des *threads* s'effectue dans l'espace noyau.
    - Utilisation d'appels système spécifiques
  - Généralement, un seul type de *thread* est disponible.

12 Cours SYE - Institut REDS / HEIG-VD

Différentes implémentations des *threads* existent. Généralement, dans les OS récents, la notion de *thread* est très bien supportée par le noyau, et l'utilisation de *thread* dans l'espace utilisateur repose sur les *threads* tels qu'ils sont définis dans le noyau. On parle alors de ***thread natif***. La création, l'ordonnancement et la destruction des *threads* est alors entièrement pris en charge par le noyau. Néanmoins, il existe une autre approche d'utiliser les *threads*: la gestion pourrait être faite entièrement dans l'espace utilisateur. Le noyau n'aurait ainsi plus aucun contrôle sur les *threads,* et, du point de vue du noyau**, le processus ne disposerait que d'un seul contexte d'exécution**. En d'autres termes, la gestion des *threads* est du ressort de l'application (utilisateur) elle-même. Par exemple, les anciennes version de *Java*, ainsi que certains langages comme *Ruby* (< 1.9) utilisaient la notion de *green threads* qui n'ont aucune interaction avec le noyau, et entièrement gérés dans l'espace utilisateur.





L'utilisation de *threads* utilisateurs non gérés par le noyau peut apporter un gain substantiel en performance puisqu'aucune interaction avec le noyau n'est nécessaire durant la création, changement de contexte, etc.

En revanche, lorsqu'un *thread* utilisateur effectue une requête I/O via un appel système (lecture, écriture dans un fichier par exemple), le processus sera complètement suspendu par le noyau (qui n'a aucune connaissance des autres *threads* en cours d'exécution dans ce processus), et pourra choisir d'exécuter un autre processus. Cette approche ne permet que difficilement d'équilibrer l'exécution des *threads* à l'intérieur d'un processus, et un *thread* dans ce cas peut affamer les autres *threads.* Une version avec des *threads* natifs permettrait au noyau, dans ce cas précis, de donner la main à d'autres *threads* du même processus, les *threads* étant "connus" par le noyau.

Références

- A. Silberschatz et al.:
  "**Operating Systems Concepts**", 8th edition, Wiley
- Andrew Tanenbaum
  "**Systèmes d'exploitation**", 3ème édition