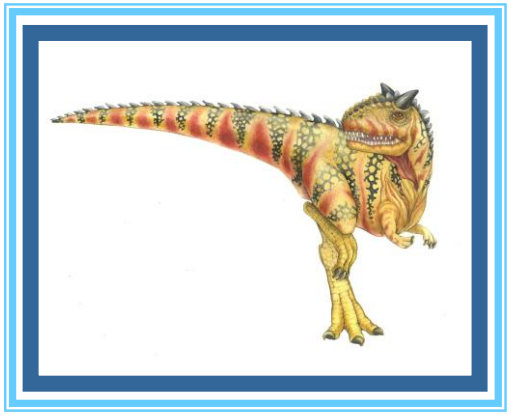
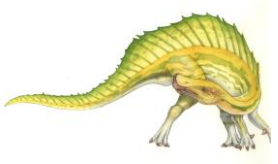
Chapter 4: Threads



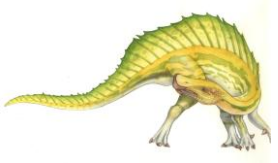
Plan

- Motivation
- Processus monothread et multithread
- Avantages
- Programmation Multicœur
- Modèles de Multithreading
- Bibliothèques de Threads
- Threading Implicite
- Problèmes liés au Threading
- Exemples de Systèmes d'Exploitation



Objectives

- Identifier les composants de base d'un thread et comparer les threads et les processus
- Décrire les avantages et les défis de la conception d'applications multithreads
- Illustrer différentes approches du threading implicite, y compris les pools de threads, fork-join et Grand Central Dispatch
- Décrire comment les systèmes d'exploitation Windows et Linux représentent les threads
- Concevoir des applications multithreadées en utilisant les API de threading Pthreads, Java et Windows



Motivation

- **Processus: programme en exécution** - Chaque processus a ses variables et fichiers indépendants des autres processus
- **Un thread est une subdivision d`un processus** : **Un fil d'exécution** dans un processus
- **Les différents threads** d'un processus **partagent les variables et les ressources** d'un même processus
  - lorsqu'un thread modifie une variable (non locale à lui), tous les autres threads voient la modification
  - un fichier ouvert par un thread est accessible aux autres threads (du même processus)

Exemple

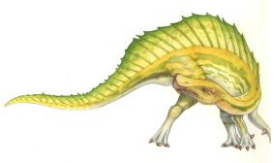
- Le processus MS-Word peut impliquer plusieurs threads:
  - Interaction avec le clavier
  - Rangement de caractères sur la page
  - Sauvegarde régulière du travail fait
  - Contrôle orthographe
  - Etc.
- Ces threads partagent tous le même fichier .doc et autres données

Pourquoi dits processus légers ?

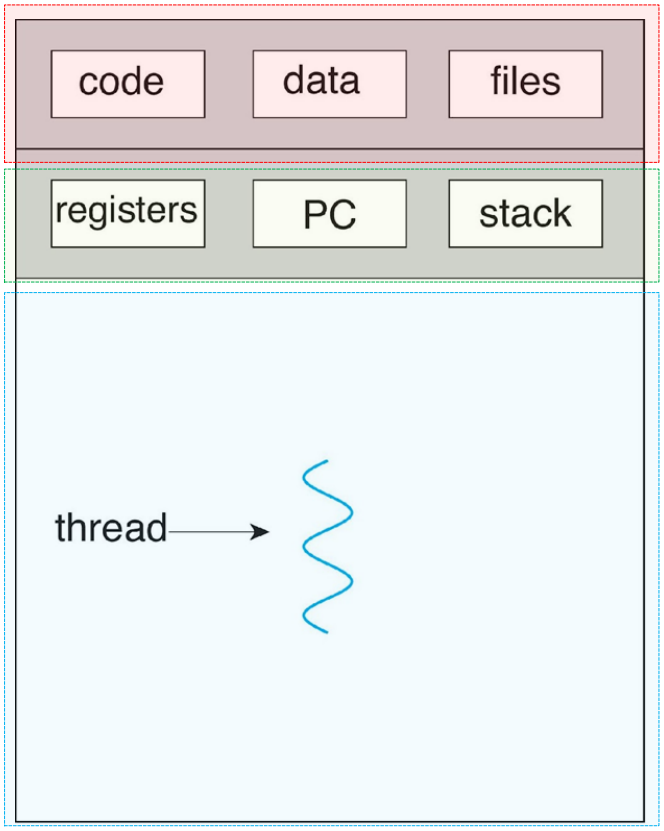
- Les threads possèdent une **structure plus légère** que les processus, car ils partagent la plupart des ressources du processus principal.
- Seules les données essentielles sont dupliquées, réduisant ainsi la consommation mémoire.
  - **Deux threads au sein d'un même processus** utilisent moins de mémoire que deux processus distincts.

- **Avantages des threads**
  - **Création rapide** : Moins de ressources à allouer par rapport à un processus.
  - **Communication efficace** : Les échanges entre threads sont plus rapides car ils ne nécessitent pas d'appels système.

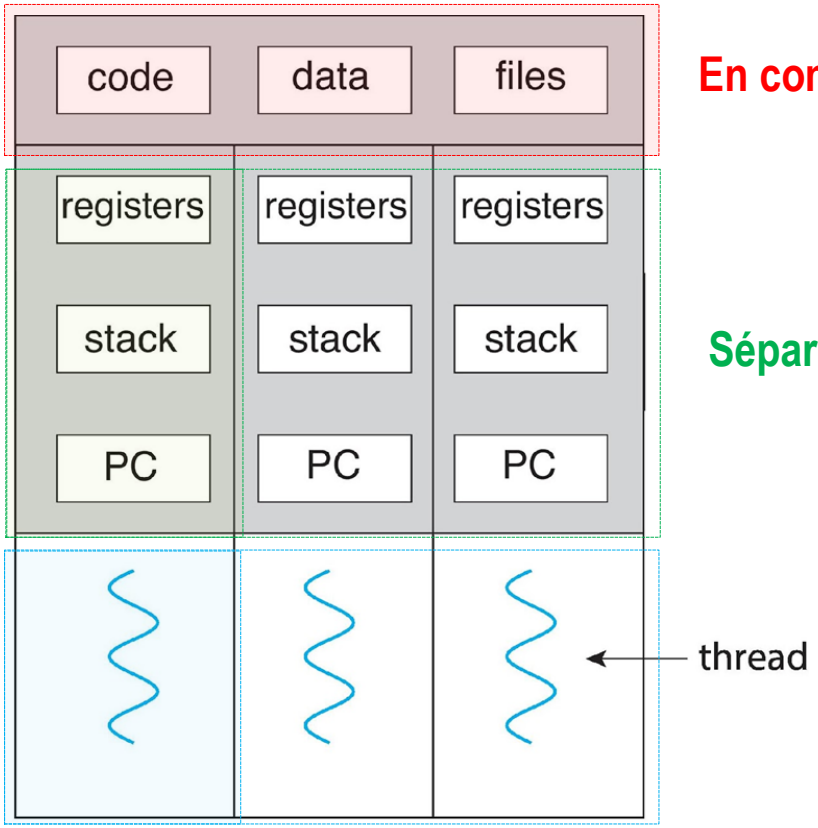
- **Importance du multithreading**
  - **Présent dans la plupart des applications modernes** pour optimiser l'exécution.
  - **Simplifie le code et améliore les performances** en exploitant le parallélisme.
  - **Les noyaux des systèmes d'exploitation sont généralement multithreads**, permettant une meilleure gestion des tâches concurrentes.



Processus monothread et multithread



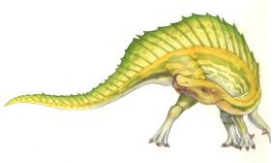
single-threaded process



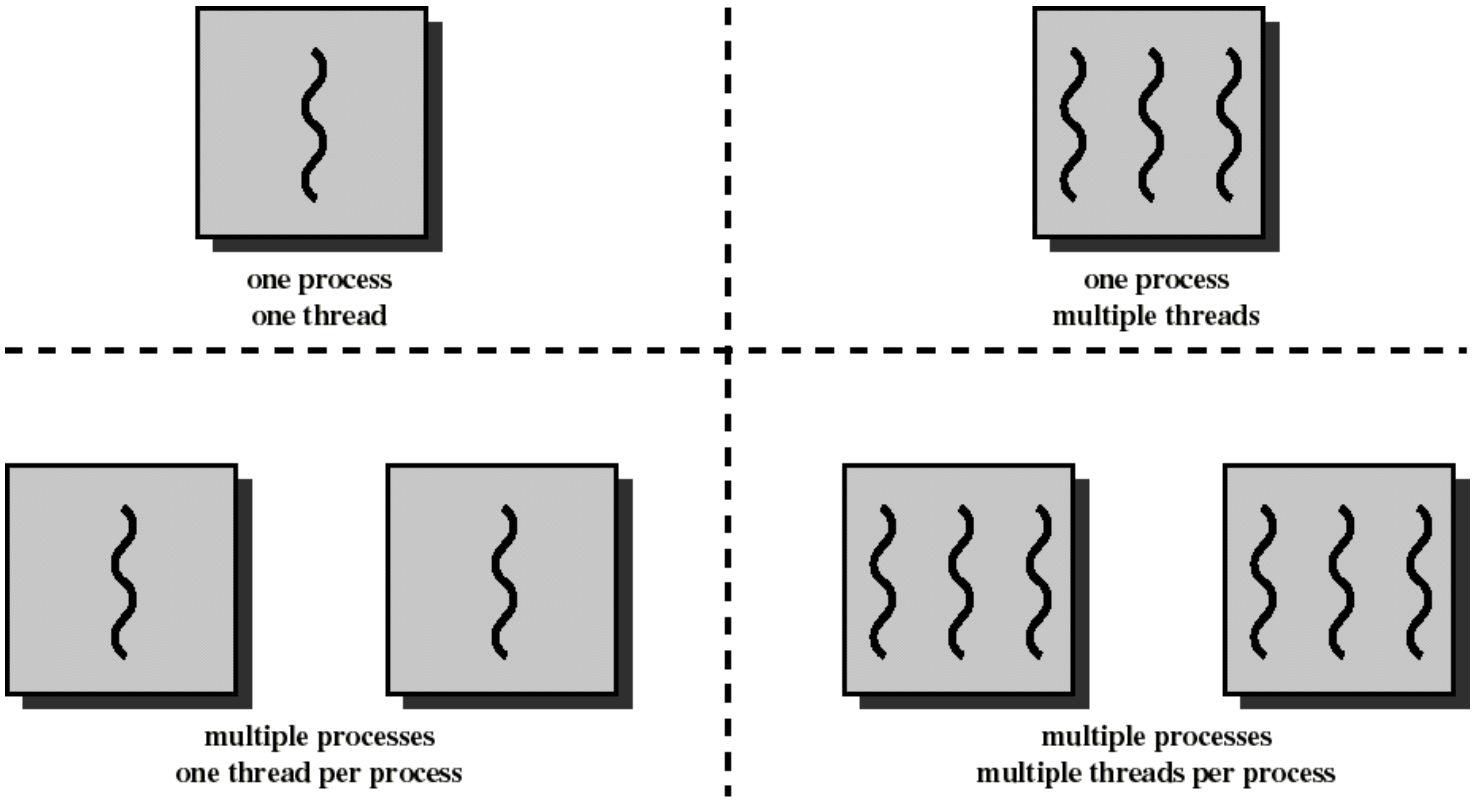
multithreaded process

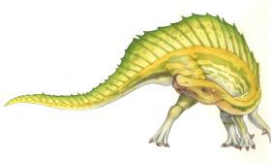
En commun

Séparés

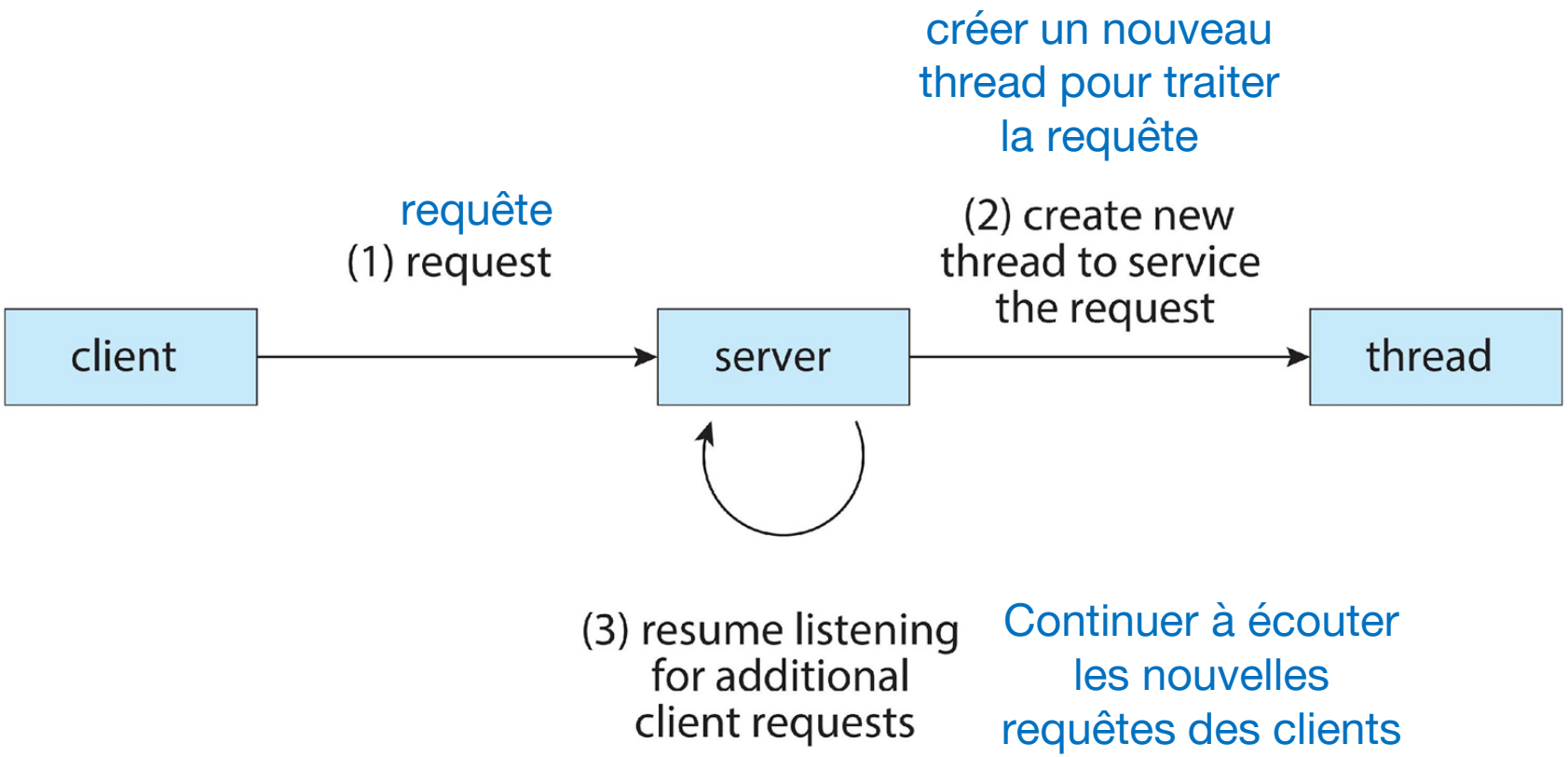


Processus monothread et multithread



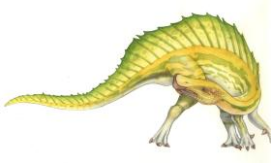


Architecture de serveur multithreads



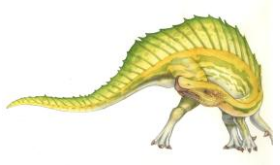
Avantages

- **Réactivité** – permet de continuer à exécuter certaines tâches même si une partie du processus est bloquée, important pour les interfaces utilisateur
- **Partage de ressources** – les threads partagent les ressources du processus, ce qui est plus simple que d'utiliser de la mémoire partagée ou de passer des messages
- **Économie** – créer des threads coûte moins cher que de créer des processus, et changer de thread est plus rapide que de changer de contexte
- **Scalabilité** – un processus peut mieux utiliser les architectures multicœurs



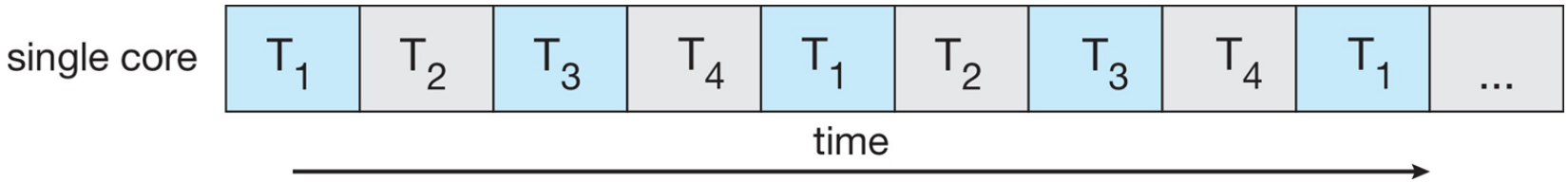
Programmation multicœur

- Les systèmes **multicœurs** ou **multiprocesseurs** imposent des défis aux programmeurs, notamment :
  - **Division des tâches**
  - **Équilibre**
  - **Division des données**
  - **Dépendances de données**
  - **Tests et débogage**
- Le **parallélisme** permet à un système d'effectuer plusieurs tâches simultanément
- La **concurrence** permet à plusieurs tâches de progresser simultanément: Cela signifie que plusieurs tâches peuvent avancer à différents moments, même si elles ne s'exécutent pas exactement en même temps. Le système gère le passage entre ces tâches, donnant l'impression qu'elles progressent simultanément.
- *Processeur/cœur unique*, le planificateur assurant la concurrence: Sur un seul processeur ou cœur, la concurrence est gérée par un planificateur (ordonnanceur) qui attribue des périodes d'exécution aux différentes tâches, créant l'illusion qu'elles s'exécutent en même temps.

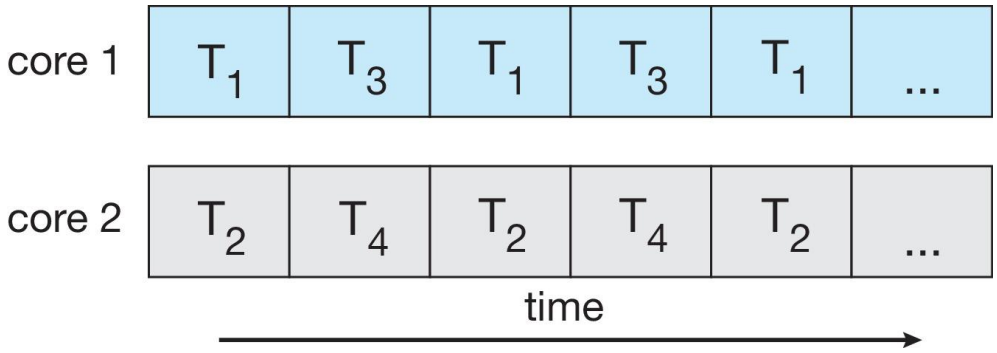


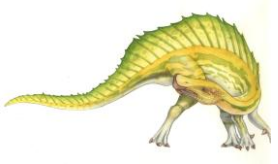
Concurrence vs. Parallélisme

- **Exécution concurrente sur un système monocœur :**



- **Parallélisme sur un système multicœur :**

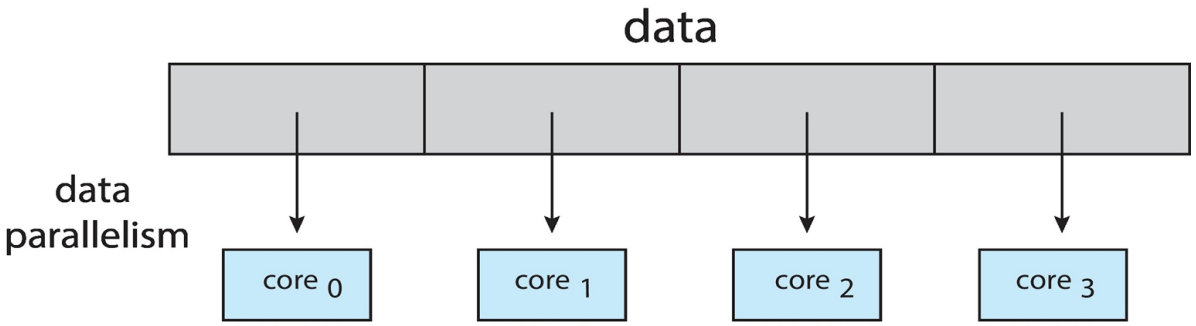


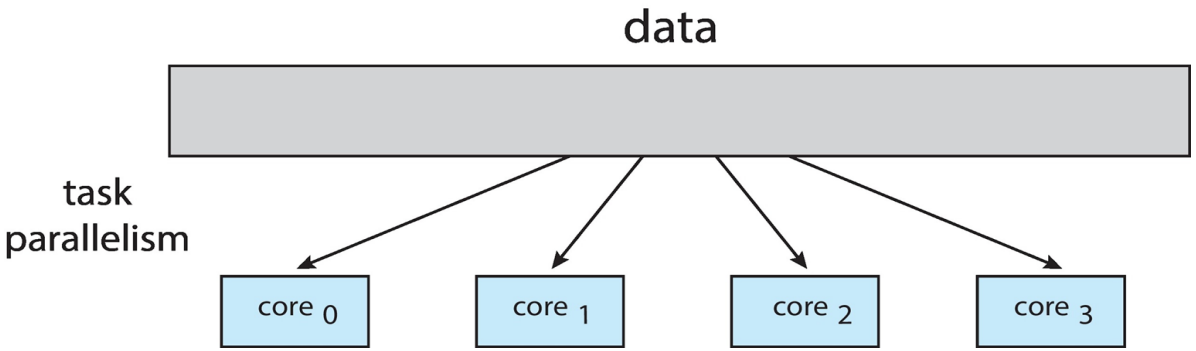


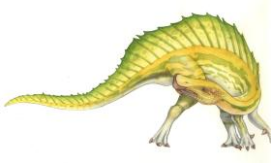
Programmation multicœur

Parallélisme des données et des tâches

- **Types de Parallélisme**
  - **Parallélisme de données** **Data parallelism** : chaque cœur traite une partie des mêmes données avec la même opération.
  - **Parallélisme de tâches** **Task parallelism** : chaque cœur exécute une tâche différente en parallèle.



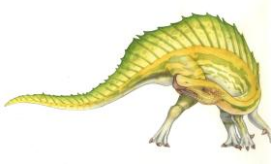




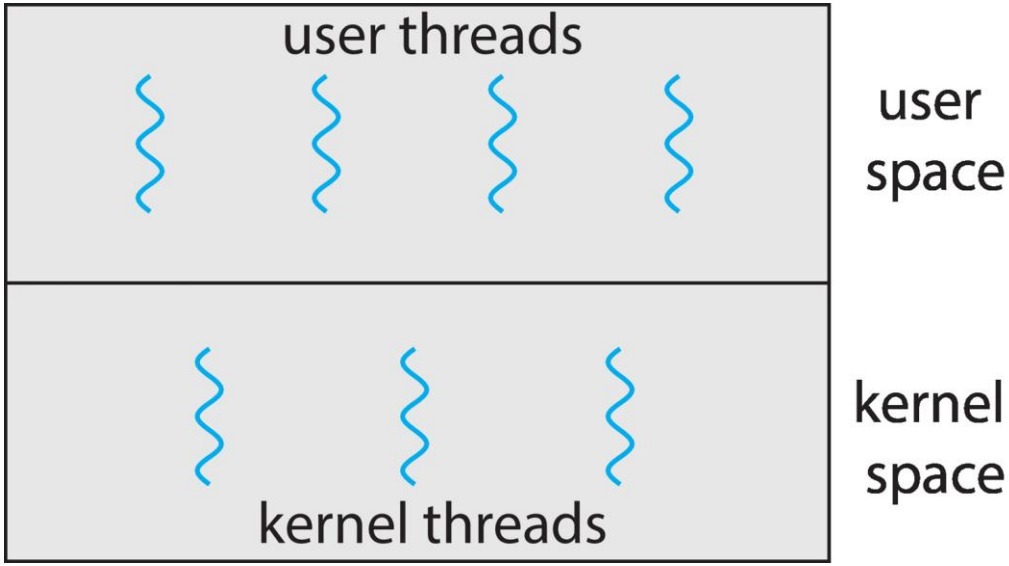
Threads Utilisateur et Threads Noyau

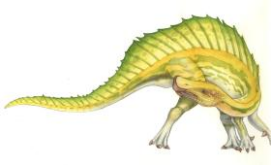
- **Threads utilisateur User threads :** gérés par une bibliothèque de threads au niveau utilisateur.
- Principales bibliothèques de threads :
  - POSIX **Pthreads**
  - Windows Threads
  - Java Threads

- **Threads noyau Kernel threads** : gérés directement par le noyau du système d'exploitation.
- Utilisés dans presque tous les OS, tels que :
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android



Threads Utilisateur et Threads Noyau





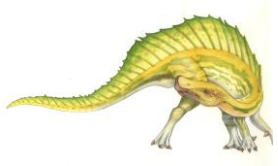
Implémentation des threads

Où implémenter les threads:

L'implémentation du (multithreading) varie considérablement d'une plateforme à une autre (threads de Win32, threads de Solaris et threads de POSIX).

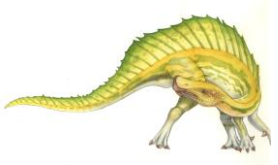
Le multithreading peut être implémenté :

- au niveau utilisateur (**threads utilisateur)** -> modèle **plusieurs-à- un**,
- au niveau noyau (**threads noyau**) -> modèle **un-à-un**, ou
- aux deux niveaux (**threads hybrides**) -> modèle **plusieurs-à-plusieurs**.



Implémentation des threads

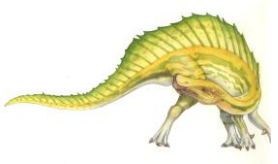
- **Threads d'utilisateur**: supportés par des bibliothèques d'usager ou le langage de programmation (p.ex Java)
  - les opérations sur les threads ne demandent pas des appels du système
  - sont gérées par le *système d'exécution du langage* de programmation (p.ex. Java)
  - le noyau ne peut pas les gérer directement
- **Threads de noyau**: supportés directement par le noyau du SE (Windows, Linux)
  - les opérations sur les threads sont des appels au système
  - le noyau est capable de gérer directement les états des threads
  - Il peut affecter différents threads à différentes UCTs



Implémentation des threads

- **Relation entre threads utilisateur et threads noyau**
  - plusieurs à un
  - un à un
  - plusieurs à plusieurs (v. Lightweight Processes)

- **Nous devons prendre en considération plusieurs niveaux:**
  - Processus
  - Thread usager
  - Thread noyau
  - Processeur (UCT)

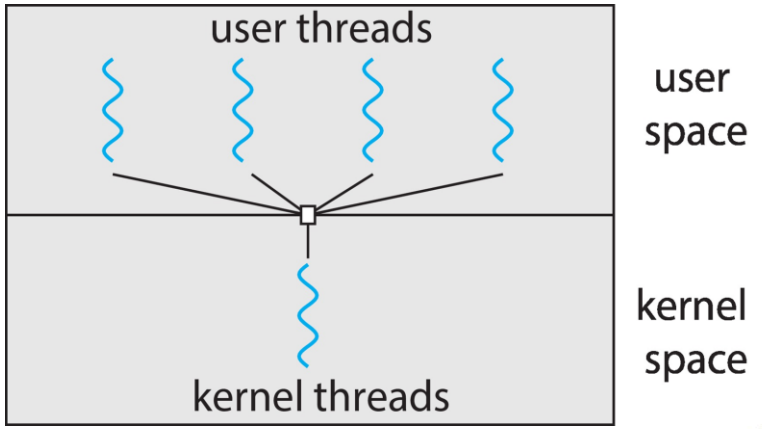


Plusieurs à un (Many-to-One)

Plusieurs threads utilisateur pour un thread noyau: l'usager ou son langage de programme contrôle les threads

- De nombreux threads au niveau utilisateur sont mappés sur un seul thread noyau.
- Le blocage d'un thread entraîne le blocage de tous les autres.
- Les threads ne peuvent pas s'exécuter en parallèle sur un système multi-cœur, car un seul peut être dans le noyau à la fois.
- Peu de systèmes utilisent actuellement ce modèle.

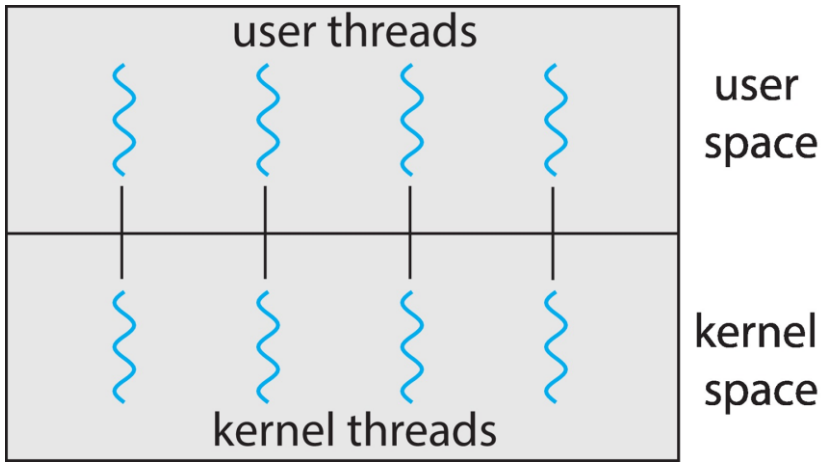
- **Exemples**:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

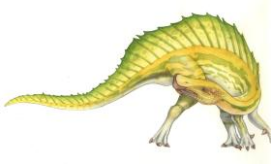


Un à Un (One-to-One)

- **Chaque thread au niveau utilisateur est mappé sur un thread noyau**.
- La création d'un thread utilisateur entraîne la création d'un thread noyau.
- Offre plus de concurrence que le modèle **many-to-one**.
- Le nombre de threads par processus est parfois limité en raison du surcoût.

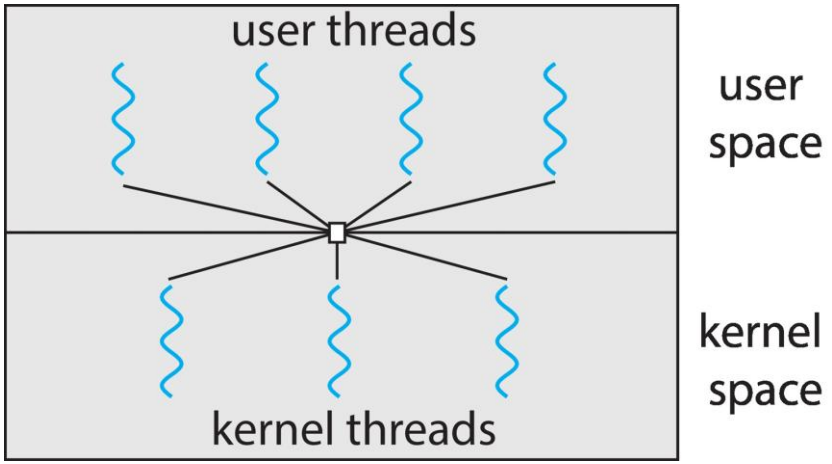
- **Exemples**
  - **Windows**
  - **Linux**

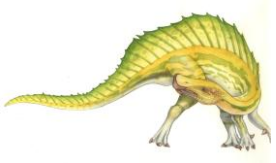




Plusieurs à Plusieurs (Many-to-Many Model)

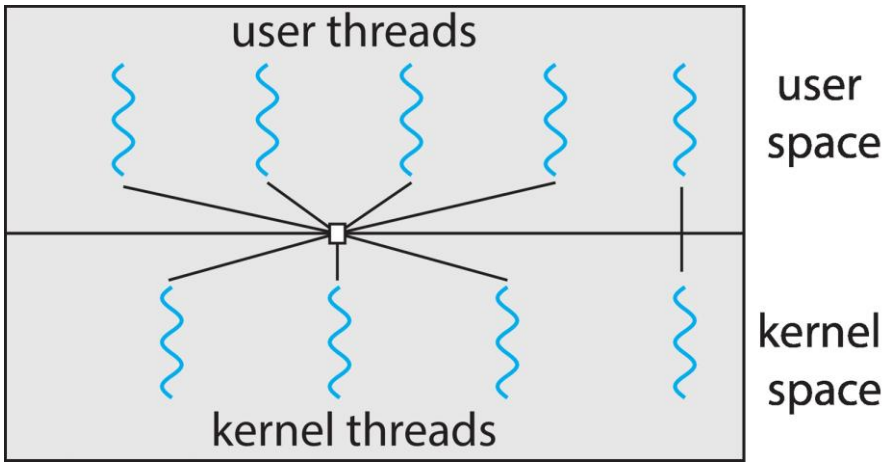
- Permet de mapper plusieurs threads utilisateur sur plusieurs threads noyau.
- Laisse le système d'exploitation créer un nombre suffisant de threads noyau.
- Utilisé sous Windows avec le package *ThreadFiber*.
- Sinon, ce modèle reste peu répandu.

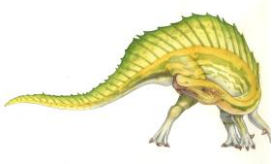




Modèle à deux niveaux (Two-Level Model)

- Combine les modèles **many-to-many** et **one-to-one**.
- Permet de mapper plusieurs threads utilisateur sur plusieurs threads noyau.
- Offre aussi la possibilité de lier certains threads utilisateur à des threads noyau spécifiques.
- Plus flexible que **many-to-many**, mais moins courant.



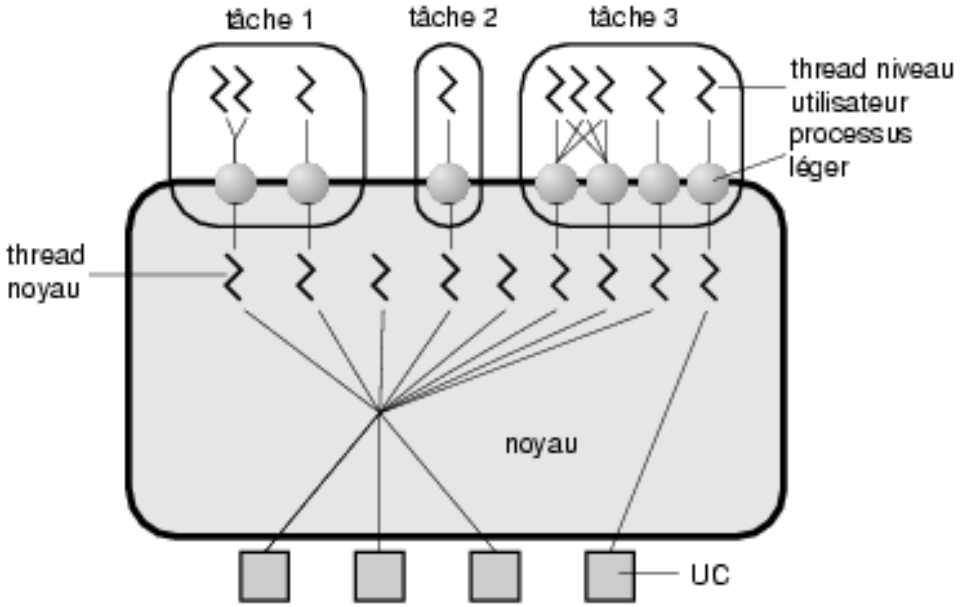


Concept de Light-Weight Process (LWP ou PL)

- Les **PL** sont des UCT virtuelles
- Représentent des threads de noyau qui peuvent exécuter des threads d'usager

- Concept implémenté dans plusieurs systèmes d'exploitation
  - **LWP=PL** dans **Unix-Linux**
  - **ThreadFiber** ou **fibres dans Windows**
  - **Hyperthreading**

- Paraît un concept semblable, mais peu de documentation disponible



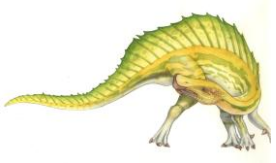
9 thread niveau usager

7 processus légers

9 threads noyau (2 inactifs)

4 Unités Centrales

Plusieurs à plusieurs, usager et noyau

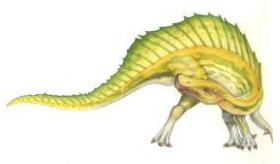


Bibliothèques de threads (Thread Libraries)

- **Thread library** Fournissent une interface API pour la création et la gestion des threads.
- Masquent les détails bas niveau liés à l'implémentation des threads.
- Offrent des fonctions pour :
  - Créer, détruire et synchroniser des threads
  - Gérer les priorités et le scheduling

- **Exemples :**
  - **POSIX Threads (pthreads)** – Standard sous UNIX/Linux
  - **Windows Threads** – API native de Windows
  - **Java Threads** – Intégrés dans la JVM, orientés objet

☞ Ces bibliothèques peuvent être implémentées au niveau utilisateur, noyau ou hybride.



Bibliothèque pthreads

- Bibliothèque normalisée POSIX (ISO/IEC 9945-1:1996)
- Existe sous Linux et ses dérivés, et sous Windows (installation depuis http://sourceware.org/pthreads-win32/)

- Fichier d'en-tête :

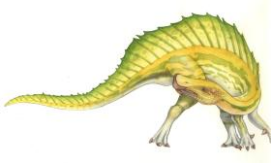
```
#include <pthread.h>
```

- Compilation et édition des liens :

```
gcc app.c -lpthread
```

- Déclaration d'un thread

```
pthread_t thread;
```



Primitives de manipulation de threads Unix

- **Création d'un thread**

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg);
```

- **Terminaison d'une thread**

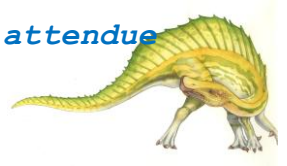
```
void pthread_exit ( void *status );
                // *status : pointeur sur le résultat retourné par la thread
```

- **Libération des ressources d'une thread :**

```
int pthread_detach( pthread_t *thread );
                // *thread : pointeur sur l'identité de la thread
```

- **Attente de la terminaison d'une thread :**

```
int pthread_join( pthread_t thread, void *status );
                // thread : identité de la thread attendue
                // *status : pointeur sur le code retour de la thread attendue
```



Bibliothèques pthreads

- **Création d'un thread**

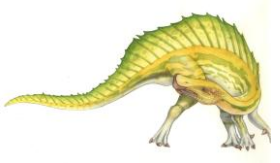
```
int pthread_create(
      pthread_t *thread,
      const pthread_attr_t *attr,
      void *(*start_routine)(void *),
      void *arg);
```

- ***thread*** est un pointeur sur une variable de type ***pthread_t*** argument de sortie
- ***attr*** permet de définir les attributs du thread (***NULL*** = attributs par défaut)
- ***arg*** est un pointeur passé à la fonction
- ***start_routine*** correspond à la fonction qui est exécutée par le thread créé. La fonction exécutée par le thread créé devra avoir le prototype suivant:

```
void *fonction(void *data);
```

Le service **pthread_create()** crée un processus léger qui exécute la fonction **start_routine** avec l'argument **arg** et les attributs **attr**.

Les attributs permettent de spécifier la taille de la pile, la priorité, la politique de planification, etc. Il y a plusieurs formes de modification des attributs.



Premier thread

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

void *thread_1(void *arg) {
    printf("Nous sommes dans le thread.\n");
    // Arrêt propre du thread
    pthread_exit(EXIT_SUCCESS);
}

int main(void) {
    // Création de la variable qui va contenir le thread
    pthread_t thread1;
    printf("Avant la création du thread.\n");
    // Création du thread
    pthread_create(&thread1, NULL, thread_1, NULL);
    printf("Après la création du thread.\n");
    return EXIT_SUCCESS;
}
```

Dans cet exemple, nous utilisons la fonction suivante:

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

- ***thread** est une référence vers la variable qui va contenir le thread;
- ***attr** correspond aux attribues de création du thread, passer à la fonction *pthread_create*;
- ***(*start_routine)** est un pointeur vers la fonction exécutée par le thread;
- ***arg** est un pointeur vers les arguments passés en paramètres à la fonction *start_routine*;
- **Le code retour de la fonction varie entre :**
  - *0* → **lorsque tout s'est bien passé;**
  - **EAGAIN** → lorsque les ressources sont insuffisantes ou le nombre de threads maximum est atteint;
  - **EINVAL** → lorsqu'un argument invalide est passé en paramètre;
  - **EPERM** → lorsqu'il y a un problème de droit sur l'ordonnanceur (passé en paramètre);

Premier thread

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

void *thread_1(void *arg) {
    printf("Nous sommes dans le thread.\n");
    // Arrêt propre du thread
    pthread_exit(EXIT_SUCCESS);
}

int main(void) {
    // Création de la variable qui va contenir le thread
    pthread_t thread1;
    printf("Avant la création du thread.\n");
    // Création du thread
    pthread_create(&thread1, NULL, thread_1, NULL);
    printf("Après la création du thread.\n");
    return EXIT_SUCCESS;
}
```

Lorsque l'on exécute cet exemple, on a le résultat suivant :

```
ayounes@vm-ubuntu:~/exemples/threads$
ayounes@vm-ubuntu:~/exemples/threads$ gcc ex1.c  -lpthread -o ex1.o
ayounes@vm-ubuntu:~/exemples/threads$ ./ex1.o
Avant la création du thread.
Après la création du thread.
ayounes@vm-ubuntu:~/exemples/threads$
```

On ne voit pas le message du thread car le programme se termine sans attendre la fin de son exécution.

Premier thread

Pour attendre la fin de l'exécution du thread, il faut utiliser la fonction suivante :

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  void *thread_1(void *arg) {
6          printf("Nous sommes dans le thread.\n");
7          // Arrêt propre du thread
8          pthread_exit(EXIT_SUCCESS);
9  }
10
11 int main(void) {
12         // Création de la variable qui va contenir le thread
13         pthread_t thread1;
14         printf("Avant la création du thread.\n");
15         // Création du thread
16         pthread_create(&thread1, NULL, thread_1, NULL);
17         pthread_join(thread1, NULL);
18         printf("Après la création du thread.\n");
19         return EXIT_SUCCESS;
20 }
21
```

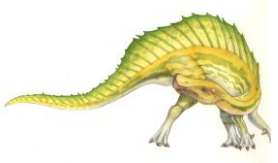
```
ayounes@vm-ubuntu:~/exemples/threads$
ayounes@vm-ubuntu:~/exemples/threads$ gcc ex1.c  -lpthread -o ex1.o
ayounes@vm-ubuntu:~/exemples/threads$ ./ex1.o
Avant la création du thread.
Après la création du thread.
ayounes@vm-ubuntu:~/exemples/threads$
ayounes@vm-ubuntu:~/exemples/threads$
ayounes@vm-ubuntu:~/exemples/threads$ gcc ex1.c  -lpthread -o ex1.o
ayounes@vm-ubuntu:~/exemples/threads$ ./ex1.o
Avant la création du thread.
Nous sommes dans le thread.
Après la création du thread.
ayounes@vm-ubuntu:~/exemples/threads$
```

Premier thread

Pour attendre la fin de l'exécution du thread, il faut utiliser la fonction suivante :

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

- **thread* est une référence vers la variable qui va contenir le thread;
- ***retval* est un pointeur vers un entier qui contiendra la valeur de retour du thread;
- le code retour de la fonction varie entre :
  - *0* → lorsque tout s'est bien passé;
  - EDEADLK → lorsqu'il y a un *deadlock*;
  - EINVAL → lorsque le thread n'est pas joignable;
  - ESRCH → si le thread n'existe pas;



Différence entre les processus et les threads

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

int x = 2;

void* routine1() {
    x += 5;
    sleep(2);
    printf("Value of x: %d\n", x);
}

void* routine2() {
    sleep(2);
    printf("Value of x: %d\n", x);
}

int main(int argc, char* argv[]) {
    pthread_t t1, t2;
    if (pthread_create(&t1, NULL, &routine1, NULL)) {
        return 1;
    }
    if (pthread_create(&t2, NULL, &routine2, NULL)) {
        return 2;
    }
    if (pthread_join(t1, NULL)) {
        return 3;
    }
    if (pthread_join(t2, NULL)) {
        return 4;
    }
    return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char* argv[]) {

    int x = 2;
    int pid = fork();

    if (pid == -1) {
        return 1;
    }

    if (pid == 0) {
        x++;
    }

    sleep(2);
    printf("Value of x: %d\n", x);

    if (pid > 0) {
        wait(NULL);
    }
    return 0;
}
```

```
ayounes@vm-ubuntu: ~/exemples/threads
ayounes@vm-ubuntu:~/exemples/threads$ gcc main-processus.c -o main-processus.o
ayounes@vm-ubuntu:~/exemples/threads$ ./main-processus.o
Value of x: 2
Value of x: 3
ayounes@vm-ubuntu:~/exemples/threads$ gcc main-threads.c -o main-threads.o
ayounes@vm-ubuntu:~/exemples/threads$ ./main-threads.o
Value of x: 7
Value of x: 7
ayounes@vm-ubuntu:~/exemples/threads$
```

End of Chapter 4