Threads

Menu

- Introduction au threads
- Les modèles pour multithreading
- Les Librairies pour les threads
- Threading implicite
- Les problèmes avec les threads

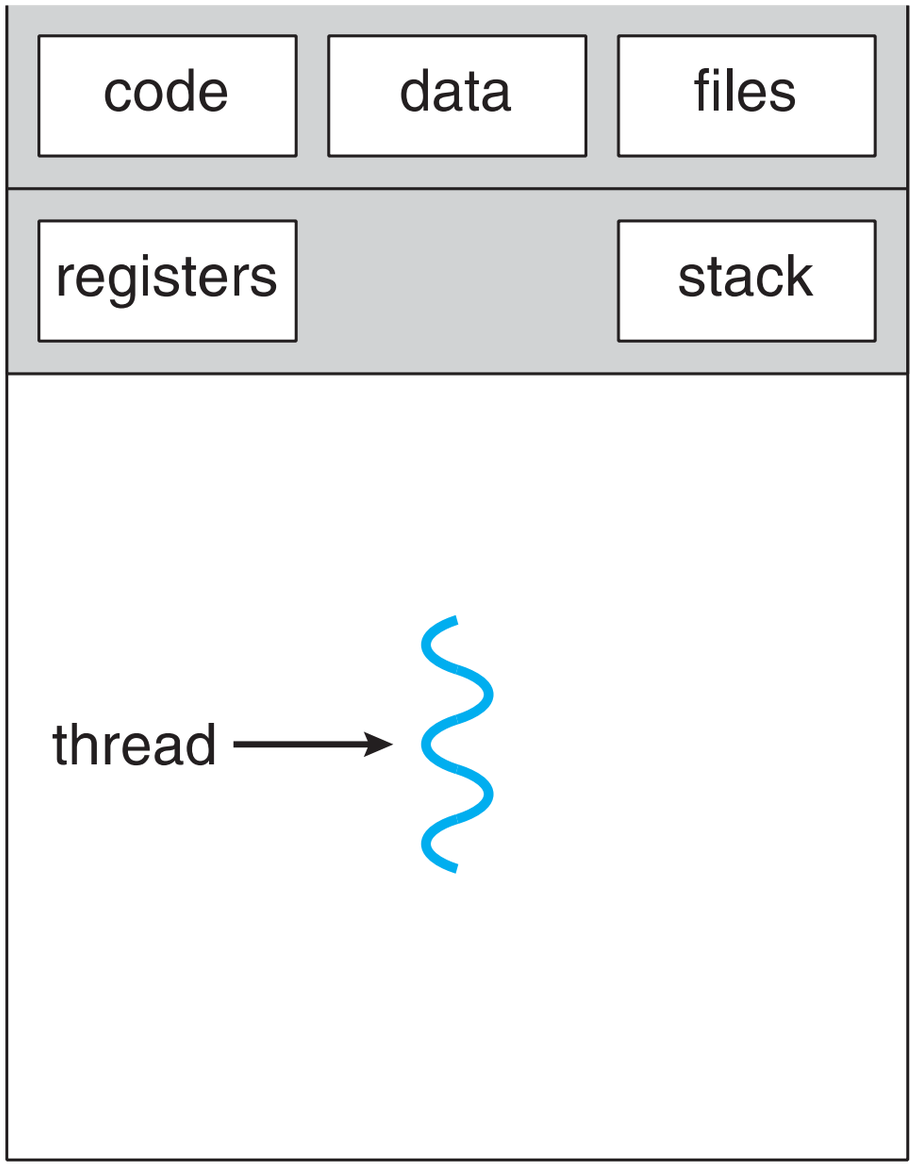
Menu

- **Introduction au threads**
- Les modèles pour multithreading
- Les Librairies pour les threads
- Threading implicite
- Les problèmes avec les threads

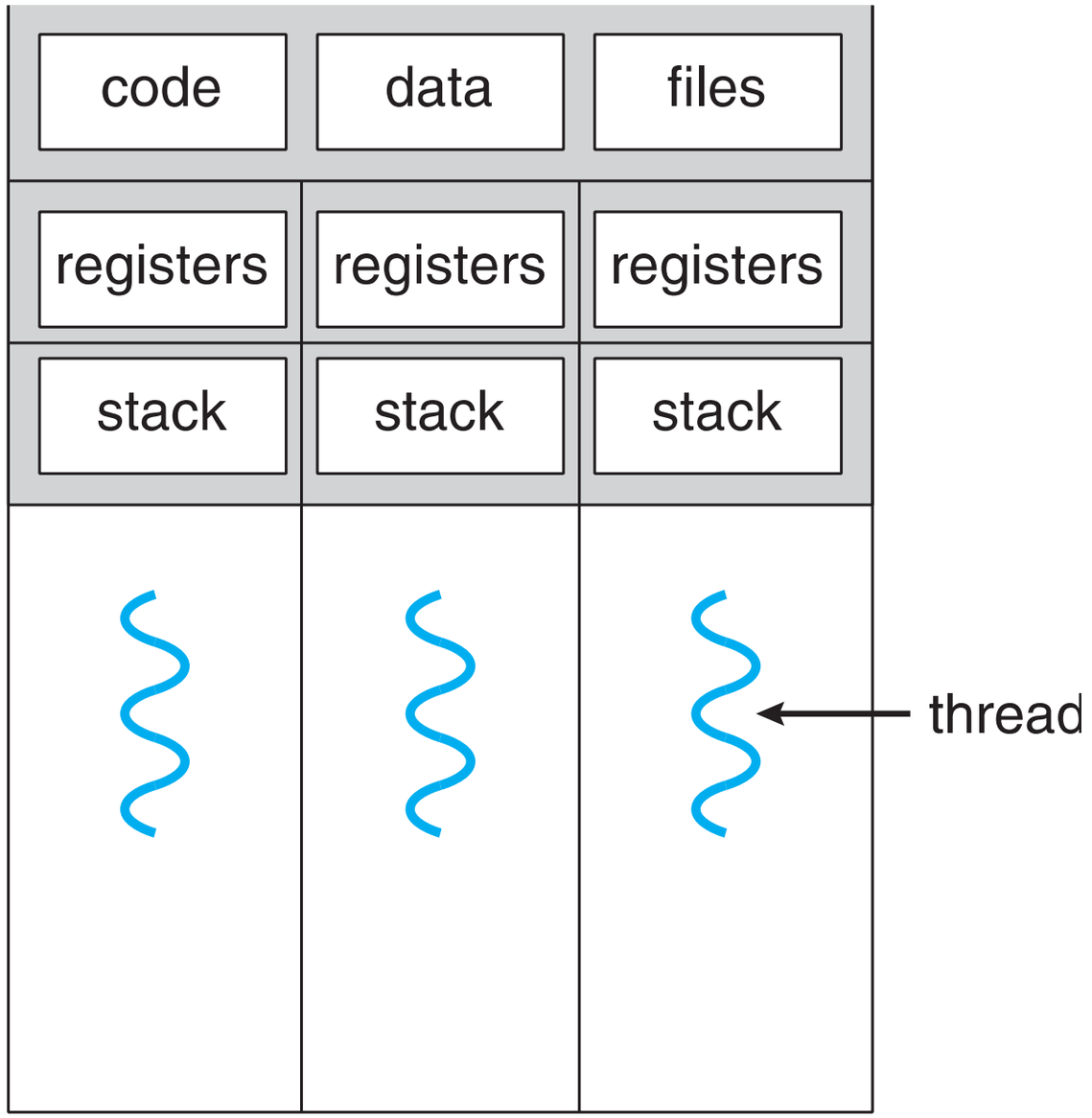
Motivation

- La plupart des applications modernes sont multithread
- Les threads s'exécutent dans l'application
- Plusieurs tâches avec l'application peuvent être implémentées par des threads séparés, par exemple,
  - Afficher l'affichage
  - Récupérer des données
  - Vérification orthographique
  - Répondre à une demande de réseau
- La création de processus est lourde tandis que la création de threads est légère
  - Pas besoin de contexte switch
  - Peut partager les resources (pas besoin de copier)
- Peut simplifier le code, augmenter l'efficacité
- Les noyaux sont généralement multithread

Les Processus “Multithreaded”

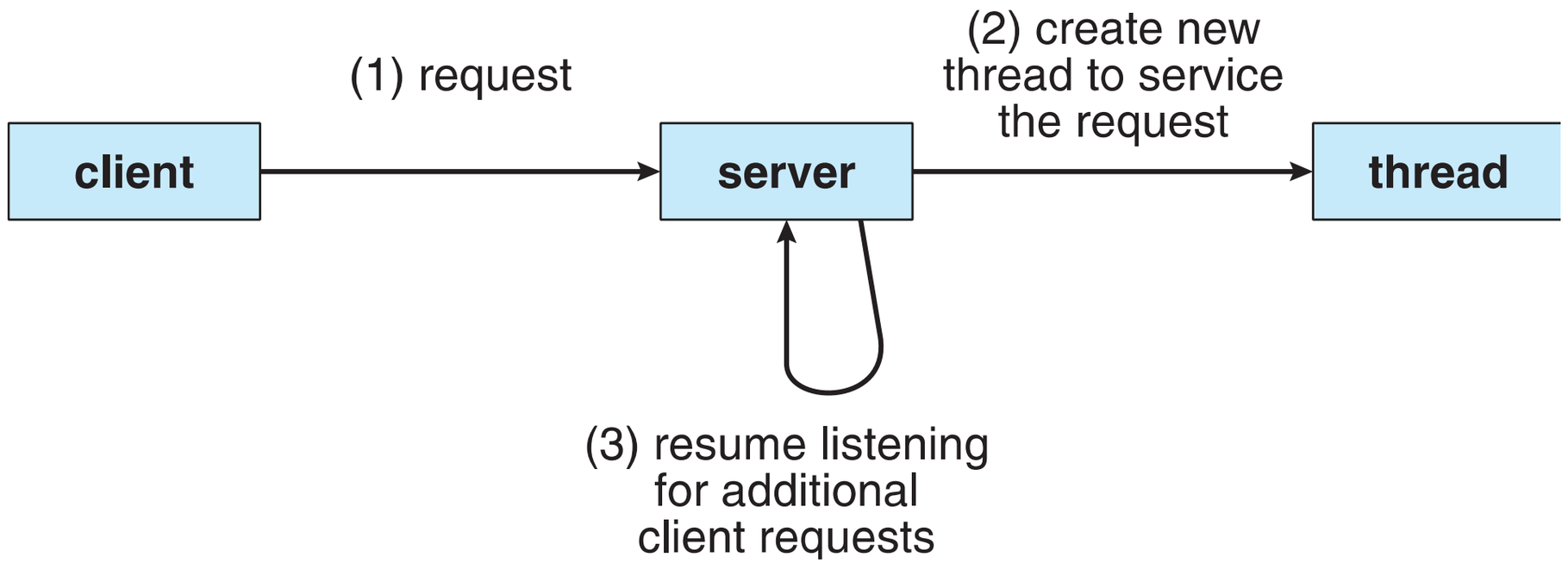


single-threaded process



multithreaded process

Architecture Server “Multithreaded”

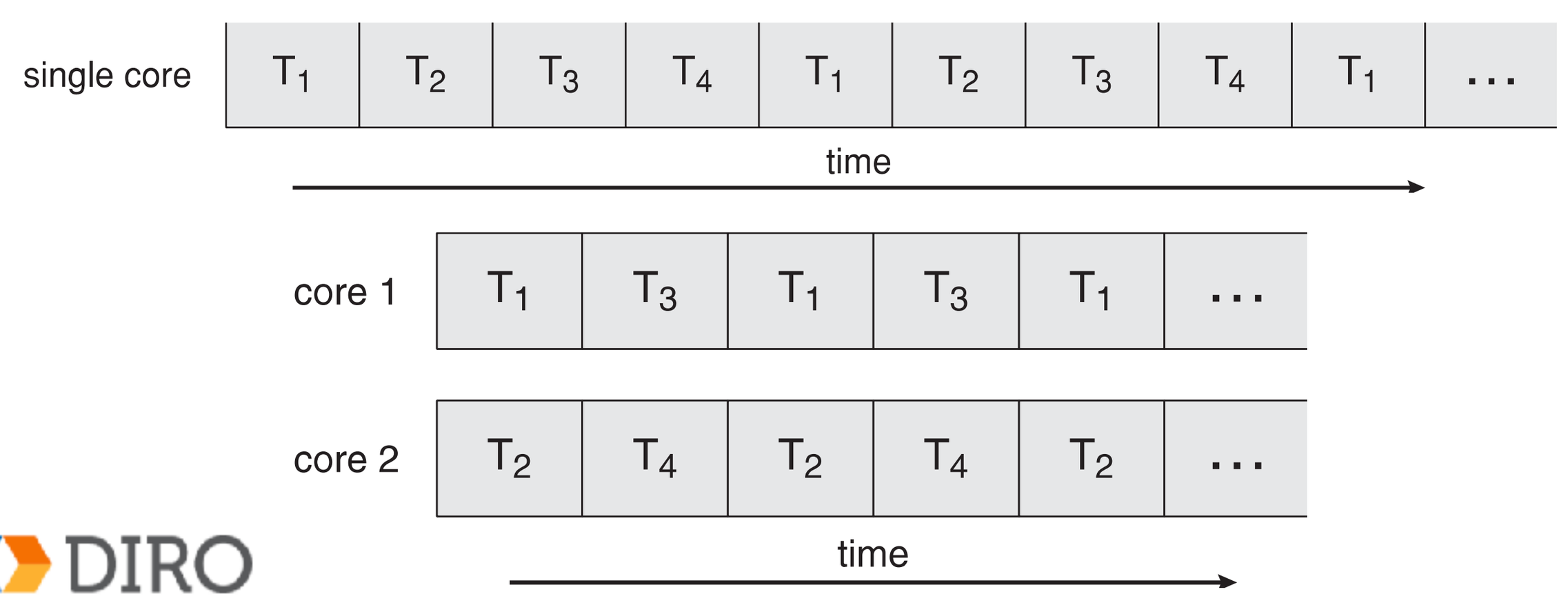


Avantages

- **Réactivité -** peut permettre une exécution continue si une partie du processus est bloquée, particulièrement importante pour les interfaces utilisateur
- **Partage de ressources -** les threads partagent des ressources de processus, plus faciles pour le programmeur que la mémoire partagée ou les mécanismes de passage de message
- **Economie -** moins cher que la création de processus (par exemple, 30x dans Solaris), la commutation de threads (par exemple, 5x dans Solaris) est plus faible que la commutation de contexte
- **Évolutivité -** le processus peut tirer parti des architectures multiprocesseurs

Concurrence vs. Parallélisme

- Le **parallélisme** implique qu'un système peut effectuer plus d'une tâche simultanément
- La **concurrence** prend en charge plus d'une tâche en cours
  - Processeur unique / noyau, ordonnanceur fournissant la concurrence et seulement une illusion de parallélisme



Types de Parallélisme

- **Parallélisme des données -** distribue des sous-ensembles des mêmes données sur plusieurs cœurs, la même opération sur chaque

- **Parallélisme des tâches -** distribution des threads à travers les cœurs, chaque thread effectuant une opération unique

Hardware Threads

- À mesure que le nombre de threads augmente, le support de l'architecture pour le threading augmente également
  - Les processeurs ont des “cores” ainsi que les “**hardware threads**”
  - “Hyper-threading”

Programmation Multicore

- Systèmes “**multi-core**” ou **multiprocesseurs** font la pression sur les programmeurs pour qu'ils adaptent leurs applications pour supporter le multithreading
- Les défis comprennent:
  - **Diviser les activités** en tâches distinctes et simultanées
  - **Balance** - doit essayer d'obtenir un travail équivalent pour les tâches
  - **Division des données** - pour éviter les collisions de données
  - **Dépendance des données** - synchronisation pour la dépendance des données
  - **Test et déboggage**

Loi d'Amdahl

- Identifie les gains de performances résultant de l'ajout de cœurs supplémentaires à une application qui comporte des composants série et parallèle
- S portion d'exécution séquentielle
- *N* nombre de processeurs

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- E.g., si application est 75% parallèle / 25% séquentielle
  - de 1 à 2 cores entraîne une accélération de ?? 1.6 times
  - de 1 à 4 cores entraîne une accélération de ?? 2.29 times

- As *N* approaches infinity, speedup approaches ?? 1/S

La partie séquentielle d'une application a un effet disproportionné sur les performances obtenues en ajoutant plus de cores

Menu

- Introduction au threads
- **Les modèles pour multithreading**
- Les librairies pour les threads
- Threading implicite
- Les problèmes avec les threads

Threads Utilisateur vs. Threads Noyau

- **Threads utilisateur** - gestion effectuée par la bibliothèque de threads de niveau utilisateur
- Trois bibliothèques de threads primaires:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads

- Les **threads du noyau** - pris en charge par le noyau
- Exemples - pratiquement tous les systèmes d'exploitation à usage général, y compris:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Modèles Multithreading

- Doit établir une relation entre les threads utilisateur et les threads du noyau

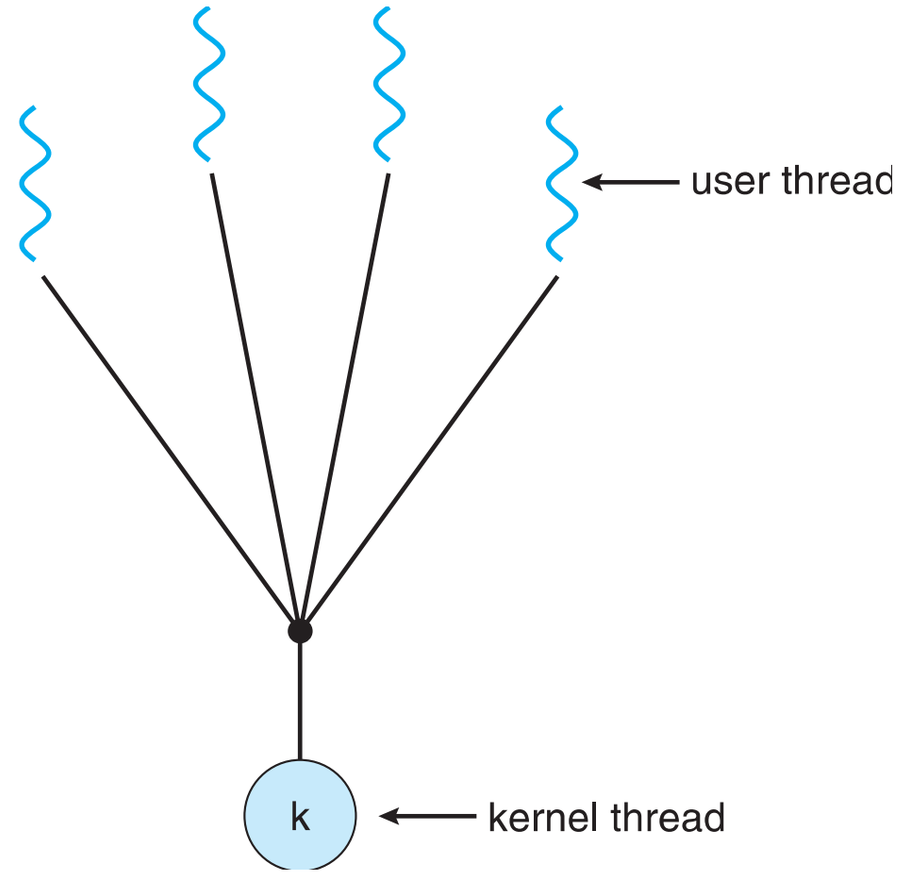
- Les threads utilisateur peuvent être gérés entre eux, mais un thread utilisateur ne peut pas être exécuté par lui-même

- Un thread utilisateur a besoin d'un thread de noyau à exécuter, mais un thread utilisateur est léger par rapport à un thread de noyau

- Trois modèles:
  - “one-to-one”
  - “many-to-one”
  - “many-to-many”

Many-to-One

- De nombreux threads de niveau utilisateur mappés à un thread de noyau unique
- Un blocage de thread (par exemple, un appel système) provoque le blocage de tous
- Plusieurs threads peuvent ne pas s'exécuter en parallèle sur un système multicore car un seul peut accéder au noyau à la fois

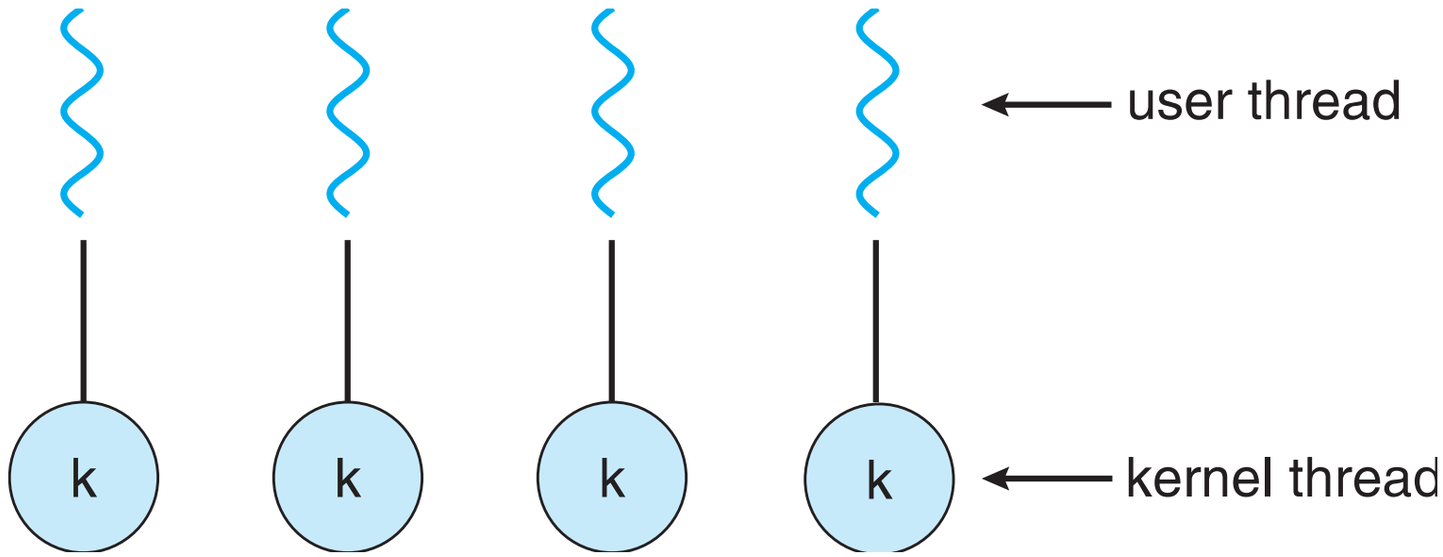


- Peu de systèmes utilisent actuellement ce modèle

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

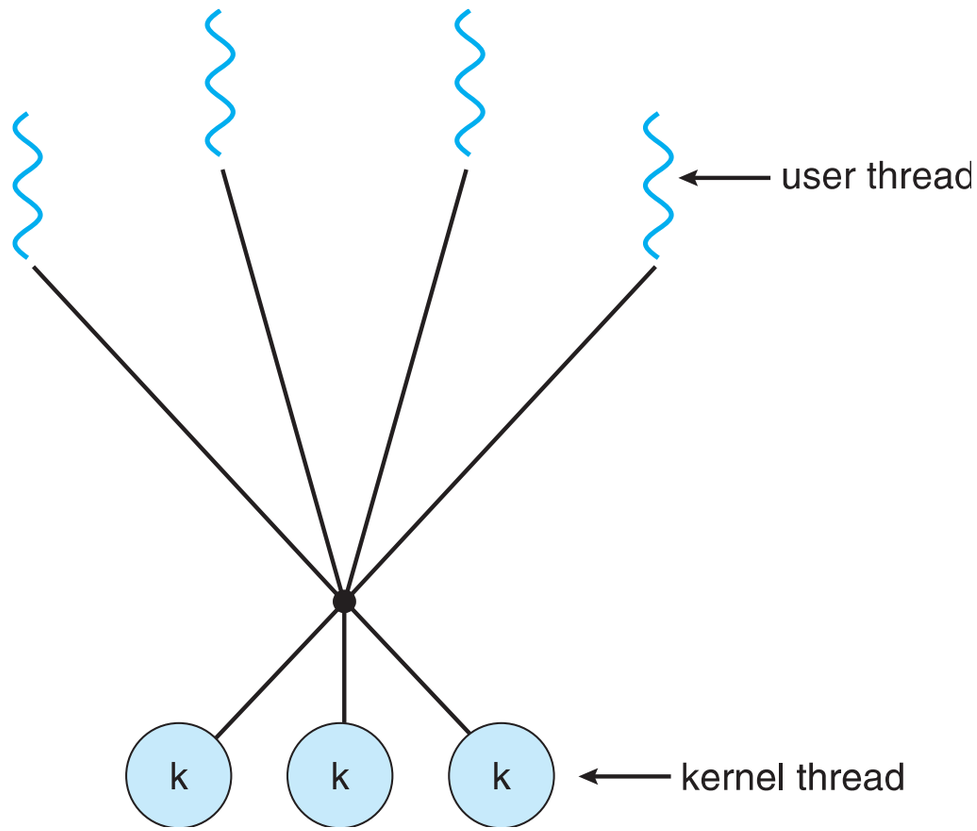
One-to-One

- Chaque thread de niveau utilisateur correspond a un thread du noyau
- La création d'un thread de niveau utilisateur crée un thread de noyau
- Plus d'accès simultané que plusieurs à un
- Le nombre de threads par processus est parfois limité en raison du temps de création des threads du noyau
- Exemples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9+



Many-to-Many

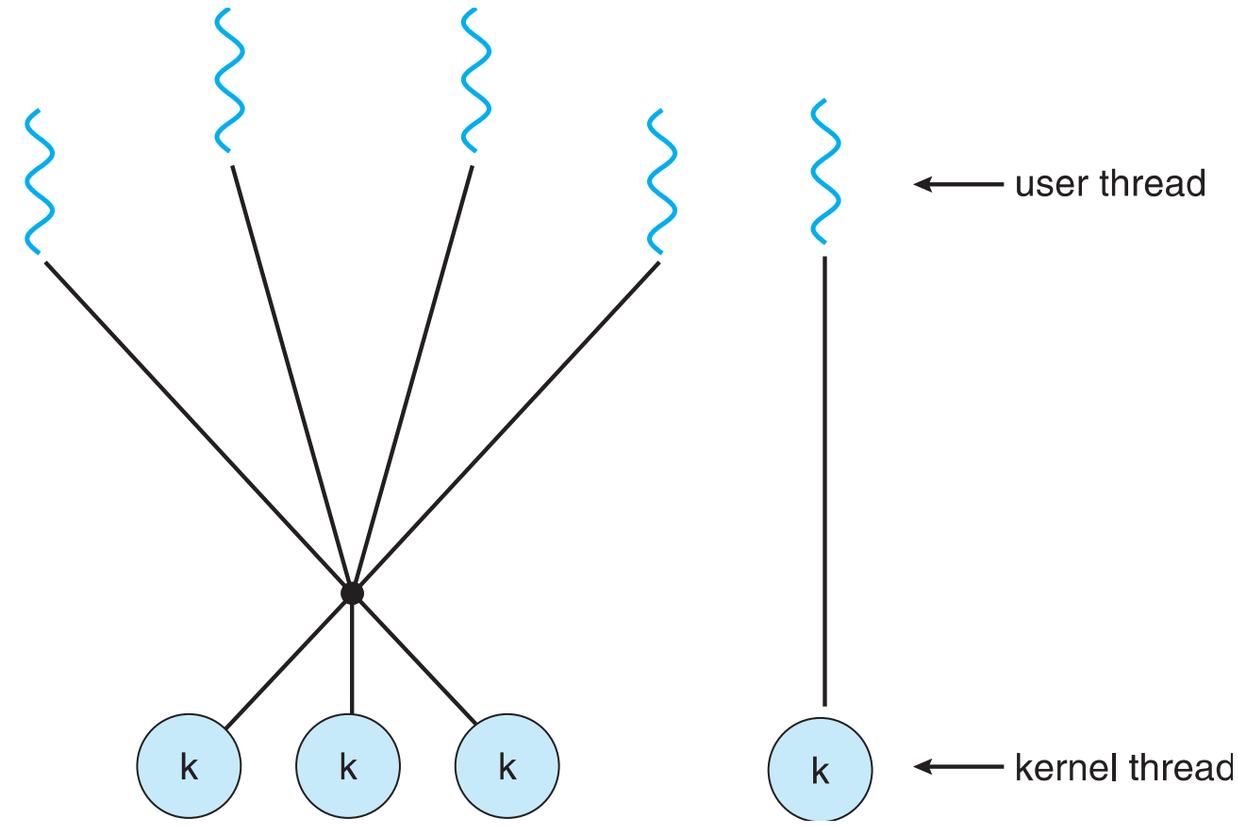
- Permet à de nombreux threads de niveau utilisateur d'être mappés vers de nombreux threads du noyau (plus petits ou égaux)
- Permet au système d'exploitation de créer un nombre suffisant de threads du noyau, en fonction de sa configuration matérielle, mais donne la possibilité à l'utilisateur de créer autant de threads utilisateur que souhaité
- Solaris avant 9
- Windows NT/2000 with the *ThreadFiber* package



Modèle à deux niveaux

- Similaire à Many-to-Many, sauf qu'il permet à un thread utilisateur d'être lié à un thread du noyau

- Exemples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 et avant



Menu

- Introduction au threads
- Les modèles pour multithreading
- **Les Librairies pour les threads**
- Threading implicite
- Les problèmes avec les threads

Librairie de Thread

- La **librairie de threads** fournit au programmeur une API pour créer et gérer des threads
- Deux principales façons de mettre en œuvre
  - Librairie entièrement dans l'espace utilisateur
  - Librairie de niveau noyau supportée par le système d'exploitation (un appel de fonction de thread entraîne un appel système au noyau)
- Trois principales bibliothèques de threads utilisées aujourd'hui
  - Pthreads POSIX (utilisateur ou noyau)
  - Librairie de threads Windows (noyau)
  - Librairie de threads Java (interfaces JVM avec la librairie de threads du système d'exploitation)
- Les threads parent-enfant peuvent être
  - synchrone: le thread parent doit attendre la fin de tous les threads enfants (par exemple, partage des données et combinaison des résultats)
  - asynchrone: le parent poursuit son exécution, ce qui est souvent le cas pour les services de type serveur

Exemple Pthreads

globale

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Exemple Pthreads (cont.)

un seul thread est commencé

le parent attend le fin d'exécution

globale

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

Win32 API Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Win32 API Multithreaded C Program (cont.)

```
/* create the thread */
ThreadHandle = CreateThread(
   NULL, /* default security attributes */
   0, /* default stack size */
   Summation, /* thread function */
   &Param, /* parameter to thread function */
   0, /* default creation flags */
   &ThreadId); /* returns the thread identifier */
```

```
   if (ThreadHandle != NULL) {
      /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Java Threads

- Les threads en Java sont géré par le JVM
- Généralement implementer en utilisant le modèle de threads fourni par le système d'exploitation sous-jacent

- Java threads peuvent être créer par:
  - Extension du Thread class
  - En utilisant l'interface Runnable

```
public interface Runnable
{
    public abstract void run();
}
```

Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (cont.)

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Menu

- Introduction au threads
- Les modèles pour multithreading
- Les Librairies pour les threads
- **Threading implicite**
- Les problèmes avec les threads

Threading Implicite

- De plus en plus populaire avec l'augmentation du nombre de threads, l'exactitude du programme est plus difficile avec des threads explicites
- La création et la gestion des threads peuvent (et devraient peut-être) être faites de préférence par des compilateurs et des bibliothèques d'exécution plutôt que par des programmeurs
  - la création de thread et la destruction nécessite du temps
  - sans limites, trop de threads pourraient épuiser les ressources du système d'exploitation
- Trois méthodes seront explorées (de nombreuses méthodes existent)
  - Pools de threads
  - OpenMP
  - Grand Central Dispatch
- Les autres méthodes incluent Microsoft Building Blocks (TBB), package `java.util.concurrent`

Thread Pools

- Lorsque le processus démarre, créez un nombre de threads dans un “pool” où ils attendent le travail
  - thread réveillé pour l'utilisation, puis retourné au pool une fois terminé
  - si aucun n'est disponible, attendez qu'un thread retourne dans le pool
- Avantages:
  - Habituellement un peu plus rapide pour traiter une requête avec un thread existant que de créer un nouveau thread
  - Permet de limiter le nombre de threads dans la ou les applications à la taille de la piscine
  - Séparation des tâches à effectuer à partir de la mécanique de création de tâche permet différentes stratégies pour exécuter la tâche
    - par exemple, les tâches peuvent être planifiées pour être exécutées après un délai ou périodiquement
- Windows API:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

OpenMP

- Ensemble de directives de compilation et une API pour C, C ++, FORTRAN
- Fournit un support pour la programmation parallèle dans des environnements à mémoire partagée
- Identifie les régions parallèles - des blocs de code pouvant fonctionner en parallèle

```
#pragma omp parallel
```

- Créez autant de threads que de cœurs, qui seront exécutés simultanément et qui se termineront lorsque le thread sera fermé. Le programmeur peut également contrôler le nombre de threads.

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

- Exécuter pour une boucle en parallèle, et le programmeur peut déterminer si les données sont partagées ou privées

```
#include <omp.h>
#include <stdio.h>

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

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Grand Central Dispatch

- Technologie Apple pour les systèmes d'exploitation Mac OS X et iOS
- Extensions aux langages C, C ++, API et bibliothèque d'exécution
- Permet l'identification des sections parallèles
- Gère la plupart des détails de l'enfilage
- Un **bloc** est dans "^ {}" - {printf ("Je suis un bloc"); }
- Blocs placés dans la queue d'attente
  - supprimé de la file d'attente et affecté au thread disponible dans le pool de threads
- Deux types de files d'attente de répartition:
  - les blocs en **série** sont supprimés dans l'ordre FIFO (le bloc suivant est exécuté après que le bloc précédent est terminé), la file d'attente est par processus, appelée **file d'attente principale**
    - Les programmeurs peuvent créer des files d'attente série supplémentaires dans le programme
  - **concurrent** - supprimé dans l'ordre FIFO mais plusieurs blocs peuvent être supprimés à la fois
  - Trois files d'attente à l'échelle du système se distinguent par les priorités: faible, défaut, élevé

Menu

- Introduction au threads
- Les modèles pour multithreading
- Les Librairies pour les threads
- Threading implicite
- **Les problèmes avec les threads**

Les Problèmes avec les Threads

- Les appels système `fork()` et `exec()`
- Traitement du signal
  - Synchrone et asynchrone
- Annulation de thread du thread cible
  - Asynchrone ou différée
- Filetage-stockage local (thread-local storage)
- Activations d'ordonnanceur

Les fork() et exec()

- Est-ce que `fork()` fait un copie de juste le thread où c'est appeler ou tous les threads dans le processus?
  - Certains UNIX ont deux versions de fork (c'est-à-dire, le thread appelant OU le processus appelant)

- `exec()` fonctionne normalement comme d'habitude - remplace le processus en cours, y compris tous les threads

Traitement du signal

- **Les signaux** sont utilisés dans les systèmes UNIX pour notifier un processus qu'un événement particulier s'est produit
- Un **gestionnaire de signal** (signal handler) est utilisé pour traiter les signaux
  1. Le signal est généré par un événement particulier
  2. Le signal est délivré à un processus
  3. Le signal est géré par l'un des deux gestionnaires de signaux:
     - défaut
     - défini par l'utilisateur
- Chaque signal a un **gestionnaire de signal par défaut** (default signal handler) que le noyau exécute lors de la gestion du signal
  - Le gestionnaire de signal **défini par l'utilisateur** peut remplacer le défaut
  - Pour un seul thread, signal délivré au processus (trivial)
- Où un signal devrait-il être fourni pour multi-thread? (dépend du type de signal)
  - Livrer le signal au fil auquel le signal s'applique (par exemple, synchrone)
  - Livrer le signal à chaque thread dans le processus (par exemple, ctrl-c)
  - Fournit le signal à certains threads du processus (par exemple, l'élément de base de données a été récupéré)
  - Affecter un thread spécifique pour recevoir tous les signaux du processus

L’annulation de threads

- Terminer un fil avant qu'il ne soit terminé
- Le thread à annuler est appelé **thread cible** (target thread)
- Deux approches générales:
  - **L'annulation asynchrone** met fin au thread cible immédiatement
  - **L'annulation différée** permet au thread cible de vérifier périodiquement s'il doit être annulé
- Des précautions doivent être prises pour les données partagées, les ressources allouées, etc.

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Annulation de threads

- L'annulation de demandes d'annulation de thread demande l'annulation, mais l'annulation dépend de l'état du thread

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

← non recommandé

- Si l'annulation du thread est désactivée, l'annulation reste en attente jusqu'à ce que le thread l'active
- Le type d'annulation par défaut est différé
  - L'annulation ne se produit que lorsque le thread atteint le **point d'annulation**
    - i.e., `pthread_testcancel()`
    - Ensuite, le **gestionnaire de nettoyage** est appelé
- Sur les systèmes Linux, l'annulation des threads est gérée par des signaux

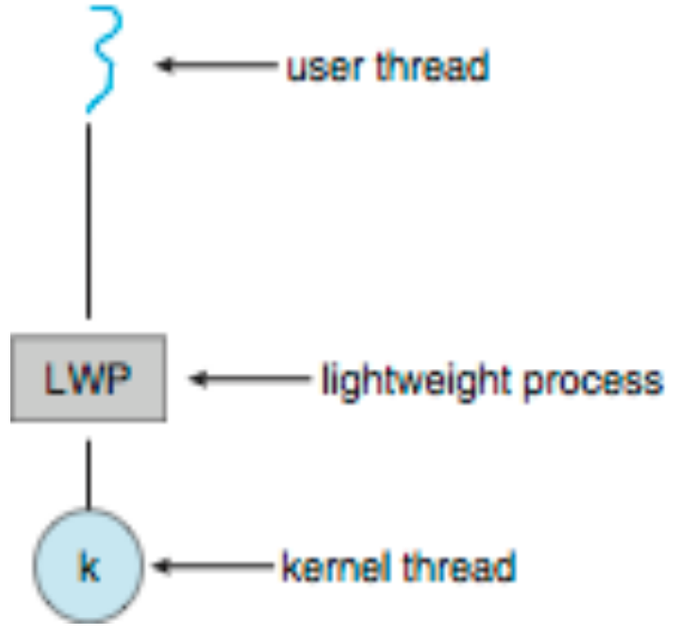
Stockage “Thread-Local” (TLS)

- Les fils d'un processus partagent les données du processus
- Stockage Thread-local (TLS) permet à chaque thread d'avoir sa propre copie de données
- Utile lorsque vous n'avez aucun contrôle sur le processus de création de thread (c'est-à-dire lorsque vous utilisez un pool de threads)
- Différent des variables locales:
  - Variables locales visibles uniquement lors d'une invocation de fonction unique
  - TLS visible à travers les invocations de fonctions
- Similaire aux données `static`
  - TLS est unique à chaque fil

Activations d'Ordonnanceur

- Les modèles Many-to-Many et deux niveaux nécessitent une communication pour gérer le nombre approprié de threads noyau alloués à l'application

- Utilisez généralement une structure de données intermédiaire entre les threads de l'utilisateur et du noyau - **processus léger** (LWP)
  - Semble être un processeur virtuel sur lequel le processus peut planifier le thread utilisateur à exécuter
  - Chaque LWP attaché au thread du noyau
  - Combien de LWP créer?



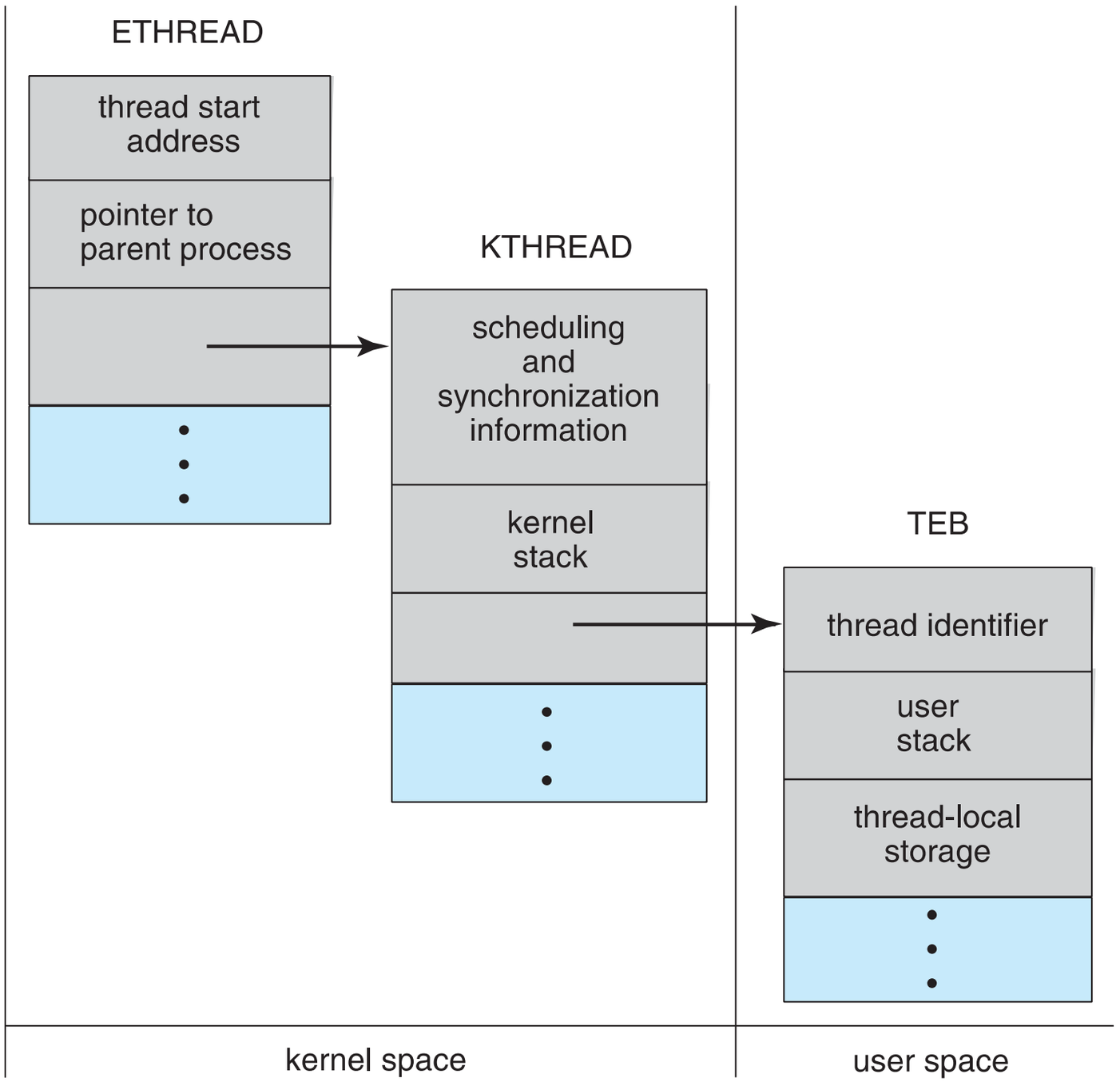
- Les activations du planificateur fournissent des **upcalls** - un mécanisme de communication du noyau vers **le gestionnaire upcall** de la librairie de threads

- Cette communication permet à une application de maintenir le bon nombre de threads du noyau

Windows Threads

- Windows implémente l'API Windows - API principale pour Win 98, Win NT, Win 2000, Win XP et Win 7
- Implémente le mappage un-à-un, au niveau du noyau
- Chaque fil contient
  - Un identifiant de thread
  - Registre ensemble représentant l'état du processeur
  - Séparez les piles de l'utilisateur et du noyau lorsque le thread s'exécute en mode utilisateur ou en mode noyau
  - Zone de stockage de données privée utilisée par les bibliothèques d'exécution et les bibliothèques de liens dynamiques (DLL)
- Le jeu de registres, les piles et la zone de stockage privée sont connus comme le **contexte** du thread
- Les structures de données primaires d'un thread incluent:
  - ETHREAD (bloc de thread exécutif) - inclut le pointeur vers le processus auquel appartient le thread et vers KTHREAD, dans l'espace noyau
  - KTHREAD (bloc de thread du noyau) - informations de planification et de synchronisation, pile en mode noyau, pointeur vers TEB, dans l'espace du noyau
  - TEB (bloc d'environnement de thread) - ID de thread, pile en mode utilisateur, stockage local de thread, dans l'espace utilisateur

Windows XP Threads Data Structures



Linux Threads

- Linux se réfère à eux comme des **tâches** plutôt que des **threads**
- La création de thread est effectuée via l'appel système de `clone()`
- `clone()` permet à une tâche enfant de partager l'espace d'adressage de la tâche parente (processus)
- Les flags contrôlent le comportement

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

- `struct task_struct` pointe vers des structures de données de processus (partagées ou uniques)

Sommaire

- Les threads peut nous donner plusieurs chemins d'exécution dans un processus
- Les threads sont plus léger a créer car les données ne sont pas copier et nous faisons pas de contexte switch
- 3 modèles: one-to-one, many-to-one, many-to-many
- Les libraries de threads: pthreads, Windows threads, Java threads
- La threading implicite nous donne un API est transfert la gestion au compilateur
- Les threads rendent la programmation et déboggage plus difficile