Hilos y concurrencia 4

Sistemas operativos y distribuidos

Gustavo Distel
gd@cs.uns.edu.ar

DCIC - UNS

Hilos y concurrencia

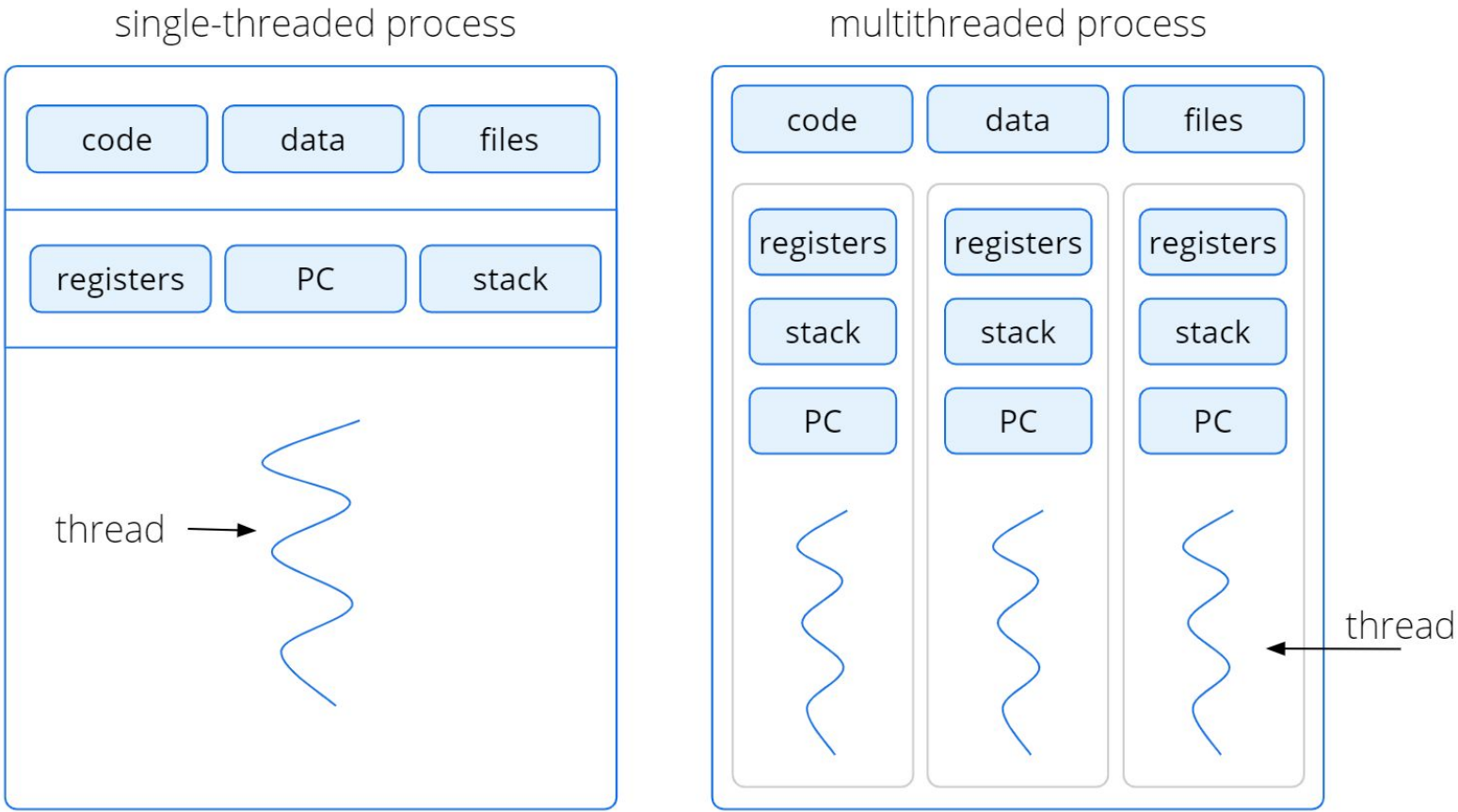
Contenido

- Generalidades.
- Programación *multicore*.
- Modelos multihilos.
- Bibliotecas de hilos.
- Hilos implícitos.
- Problemas de hilos.
- Ejemplos de sistemas operativos.

Introducción

- Prácticamente todos los SO modernos proveen características que permiten que un proceso contenga múltiples hilos de control.
- Es importante en los sistemas *multicore* identificar tareas paralelas mediante el uso de hilos.
- Un hilo es una unidad básica de utilización de la *CPU*; comprende:
  - Un *ID* de hilo (*thread ID*),
  - Un contador de programa (*PC*),
  - Un conjunto de registros y
  - Una pila.
  - Comparte con otros hilos que pertenecen al mismo proceso:
    - Sección de código,
    - Sección de datos y
    - Otros recursos del SO, como archivos abiertos y señales.
- Un proceso tradicional tiene un solo hilo de control. Si un proceso tiene múltiples hilos de control, puede realizar más de una tarea a la vez.

Introducción



Introducción

Motivación

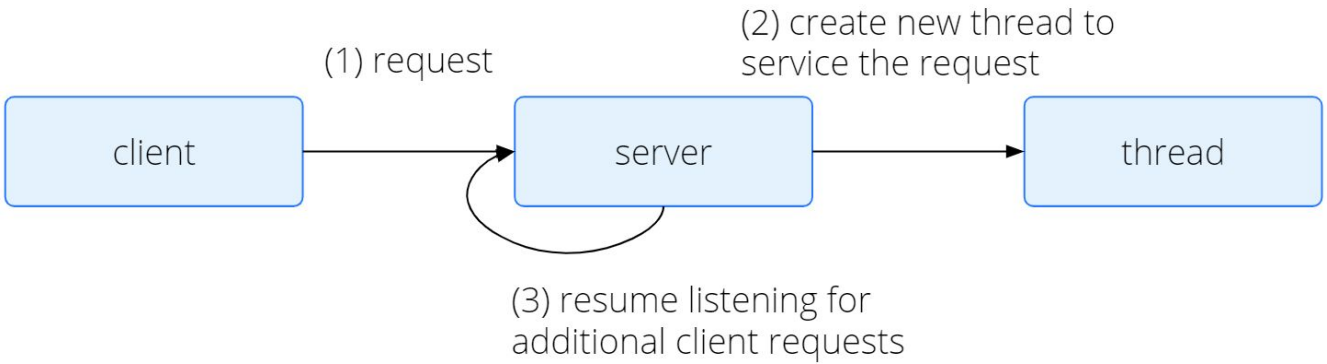
Ejemplos de aplicaciones multihilos:

- Una aplicación que genera *thumbnails* de fotos a partir de una colección de imágenes puede usar un hilo separado para generar un *thumbnails* de cada imagen separada.
- Un navegador *web* puede tener un hilo para mostrar imágenes o texto, mientras que otro hilo recupera datos de la red.
- Un procesador de texto puede tener un hilo para mostrar gráficos, otro hilo para responder al tecleo del usuario y un tercer hilo para realizar la revisión ortográfica y gramatical en segundo plano.
- Las aplicaciones también se pueden diseñar para aprovechar las capacidades de procesamiento en sistemas *multicore*. Dichas aplicaciones pueden realizar varias tareas intensivas de procesamiento en paralelo en múltiples *cores*, por ej. minería de datos, gráficos, inteligencia artificial, ordenamiento, etc.

Introducción

Motivación

- Un servidor *web* muy cargado puede tener miles de clientes accediendo simultáneamente.
- Una solución es que el servidor se ejecute en un proceso que acepta solicitudes y cuando recibe una solicitud, crea un proceso separado para atender dicha solicitud.
- Previo a que los hilos se hicieran populares crear procesos era común. Sin embargo, la creación de procesos requiere mucho tiempo y recursos. Si el nuevo proceso realizará las mismas tareas que el proceso existente, ¿por qué incurrir en toda esa sobrecarga?
- Por lo general, es más eficiente que un proceso tenga múltiples hilos. Cuando se realiza una solicitud, en lugar de crear otro proceso, el servidor crea un nuevo hilo para atender la solicitud y reanudar la escucha de solicitudes adicionales.



Introducción

Beneficios

- Capacidad de respuesta (*Responsiveness*).
  - Una aplicación multihilo interactiva permite que un programa continúe ejecutándose incluso si parte de él está bloqueada o está realizando una operación prolongada, lo que aumenta la capacidad de respuesta al usuario.
  - Esta cualidad es especialmente útil en el diseño de interfaces de usuario:
    - Por ej.: hacer clic en un botón de una operación que requiere mucho tiempo. Una aplicación de un solo hilo no responderá al usuario hasta que se complete la operación.
    - Por el contrario, si la operación que consume mucho tiempo se realiza por separado (hilo asincrónico), la aplicación seguirá respondiendo al usuario.

Introducción

Beneficios

- Compartir recursos.
  - Los procesos pueden compartir recursos solo a través de técnicas como la memoria compartida y el paso de mensajes.
  - Dichas técnicas deben ser organizadas explícitamente por el programador.
  - Sin embargo, los hilos por defecto comparten la memoria y los recursos del proceso al que pertenecen.
  - El beneficio de compartir código y datos permite que una aplicación tenga varios hilos diferentes dentro del mismo espacio de direcciones.

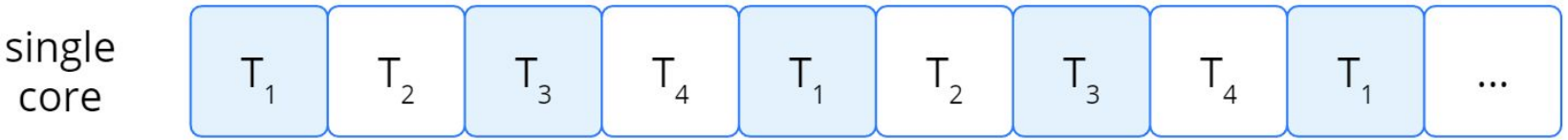
Introducción

Beneficios

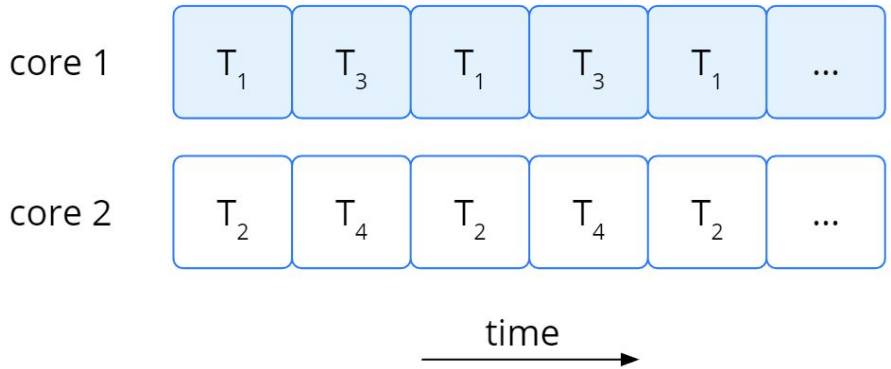
- Economía (*Economy*):
  - La asignación de memoria y recursos para la creación de procesos es costosa.
  - Debido a que los hilos comparten los recursos del proceso al que pertenecen, es más económico crear y hacer cambio de contexto de hilos.
  - La medición empírica de la diferencia en la sobrecarga puede ser difícil, pero en general la creación de hilos consume menos tiempo y memoria que la creación de procesos.
  - Además, el cambio de contexto suele ser más rápido entre hilos que entre procesos.
- Escalabilidad (*Scalability*):
  - Los beneficios del multihilo pueden ser aún mayores en una arquitectura multiprocesador, donde los hilos pueden ejecutarse en paralelo en diferentes *cores* de procesamiento.
  - Un proceso de un solo hilo puede ejecutarse en un solo procesador, independientemente de cuántos estén disponibles.

Programación *multicore*

- La programación multihilo proporciona un mecanismo para un uso eficiente de los *cores* en un sistema *multicore*.
- Ejecución concurrente en un sistema *single-core*:



- Paralelismo en un sistema *multi-core*:



Programación *multicore*

- Diferencia entre concurrencia y paralelismo:
  - Un sistema concurrente soporta más de una tarea al permitir que todas las tareas progresen.
  - Un sistema paralelo puede realizar más de una tarea simultáneamente.
  - Por lo tanto, es posible tener concurrencia sin paralelismo.
- Antes de la llegada de las arquitecturas multiprocesador y *multicore*, la mayoría de los sistemas informáticos tenían un solo procesador, y el planificador de la *CPU* era diseñado para proporcionar la ilusión de paralelismo al cambiar rápidamente entre procesos, permitiendo así que cada proceso progrese.
- Dichos procesos se ejecutaban concurrentemente, pero no en paralelo.

Programación *multicore*

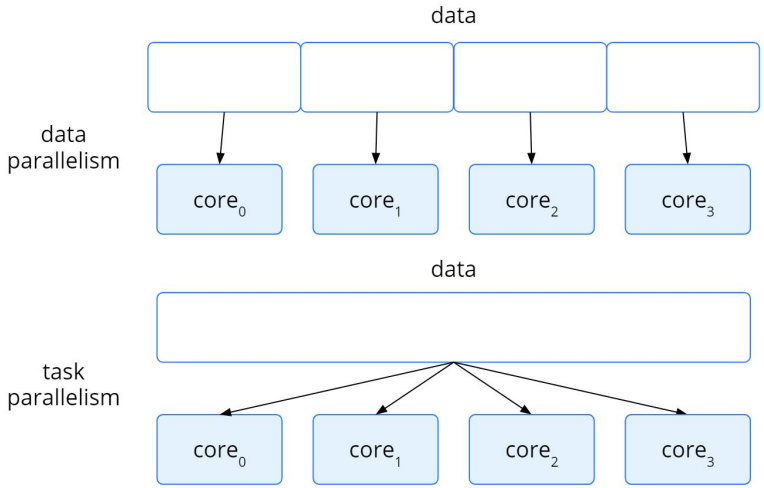
Desafíos de programación

- **Identificación de tareas**: examinar las aplicaciones para encontrar áreas que se puedan dividir en tareas separadas y concurrentes.
- **Equilibrio**: asegurarse de que las tareas realicen la misma cantidad de trabajo. En algunos casos, una tarea puede no aportar tanto valor al proceso general como otras tareas.
- **División de datos**: al igual que las aplicaciones se dividen en tareas separadas, los datos a los que las tareas acceden y manipulan deben dividirse para ejecutarse en *cores* separados.
- **Dependencia de datos**: los datos a los que acceden las tareas deben examinarse en busca de dependencias entre tareas.
  - Cuando una tarea depende de los datos de otra, los programadores deben asegurarse de que la ejecución de las tareas esté sincronizada.
- **Pruebas y depuración**: cuando un programa se ejecuta en paralelo en múltiples *cores*, son posibles muchas rutas de ejecución diferentes.
    - Probar y depurar estos programas concurrentes es inherentemente más difícil que probar y depurar aplicaciones de un sólo hilo.

Programación *multicore*

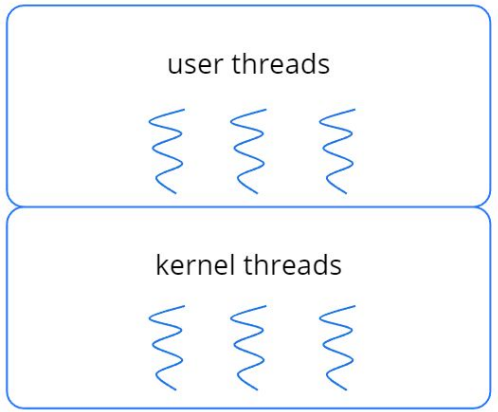
Tipos de paralelismo

- En general, hay dos tipos de paralelismo: paralelismo de datos y paralelismo de tareas.
  - **Paralelismo de datos**: se centra en distribuir subconjuntos de los mismos datos en múltiples *cores* y realizar la misma operación en cada *core*. Ej.: sumar el contenido de *arrays*.
  - **Paralelismo de tareas**: distribuye tareas (hilos) a través de múltiples *cores*. Cada hilo realiza una operación única. Diferentes hilos pueden estar operando con los mismos datos, o con diferentes datos.
- El paralelismo de datos implica la distribución de datos a través de múltiples *cores*, y el paralelismo de tareas involucra la distribución de tareas a través de múltiples *cores*.



Modelos de Multithreading

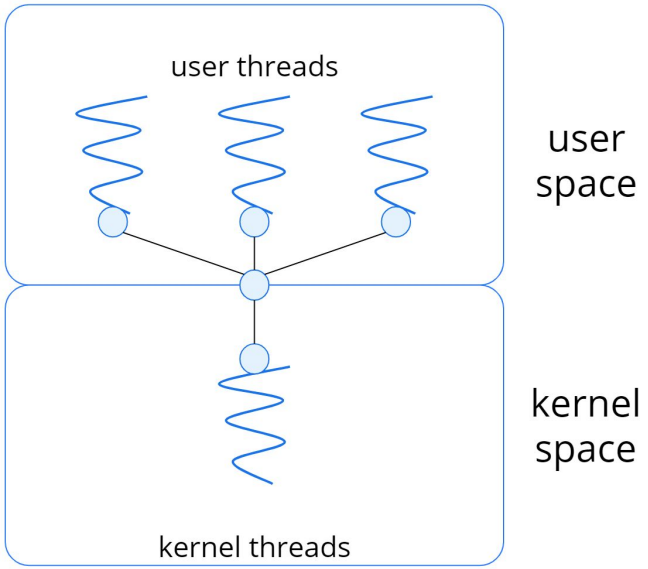
- El soporte de hilos puede ser:
  - **A nivel de usuario (*user threads*)**:
    - Los hilos de usuario se proveen por encima del *kernel* y se administran sin su soporte.
  - **A nivel de *kernel* (*kernel threads*)**:
    - Los hilos a nivel *kernel* son soportados y administrados directamente por el SO.
- Hay tres maneras de establecer una relación de este tipo:
  - el **modelo muchos a uno (*many-to-one*)**,
  - el **modelo uno a uno (*one-to-one*)** y
  - el **modelo muchos a muchos (*many-to-many*)**.



Modelos de Multithreading

muchos a uno (*many-to-one*)

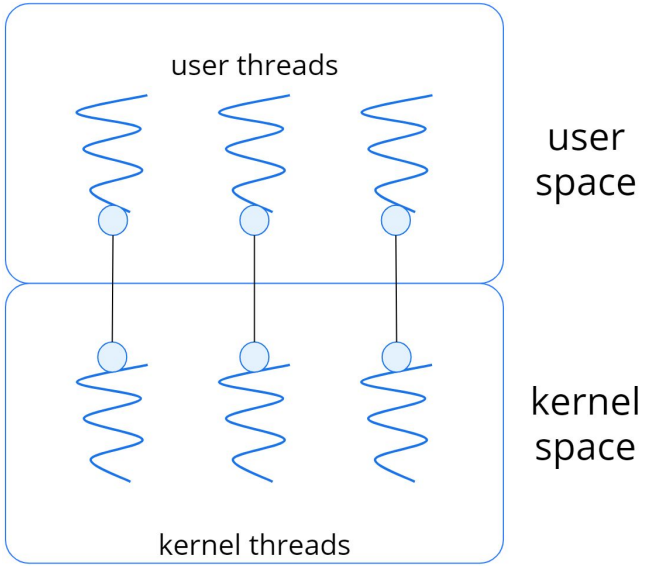
- Mapea muchos hilos de nivel de usuario a un hilo del *kernel*.
- La gestión de los hilos se realiza mediante la biblioteca de hilos en el espacio de usuario.
- Todo el proceso se bloqueará si un hilo realiza una llamada al sistema bloqueante.
- Dado que solo un hilo puede acceder al *kernel* a la vez, varios hilos no pueden ejecutarse en paralelo en sistemas *multicore*.
- Muy pocos sistemas continúan utilizando este modelo debido a su incapacidad para aprovechar múltiples *cores* de procesamiento, que se han convertido en un estándar en la mayoría de los sistemas informáticos.
- Ej.: *Solaris green threads.*



Modelos de Multithreading

uno a uno (*one-to-one*)

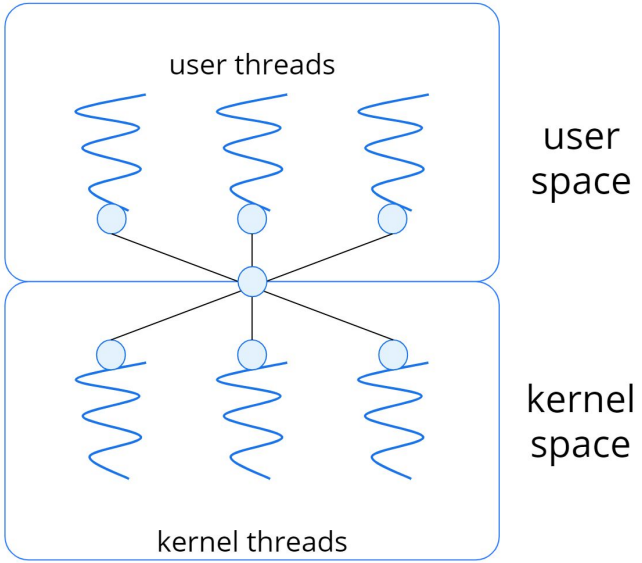
- Mapea cada hilo de usuario a un hilo del *kernel*.
- Proporciona más concurrencia que el modelo de **muchos a uno** al permitir que se ejecute otro hilo cuando un hilo realiza una llamada al sistema bloqueante.
- También permite que varios hilos se ejecuten en paralelo en multiprocesadores.
- El único inconveniente de este modelo es que la creación de un hilo de usuario requiere la creación del hilo del *kernel* correspondiente, y una gran cantidad de hilos del *kernel* puede afectar el rendimiento de un sistema.
- Linux y la familia Windows, implementan el modelo **uno a uno**.



Modelos de Multithreading

muchos a muchos (*many-to-many*)

- El modelo de **muchos a muchos** multiplexa muchos hilos de nivel de usuario a un número menor o igual de hilos del *kernel*.
- El número de hilos del *kernel* puede ser específico para una aplicación en particular o una máquina en particular:
  - Por ej.: a una aplicación se le pueden asignar más hilos en un sistema con ocho *cores* que en un sistema con cuatro *cores*.

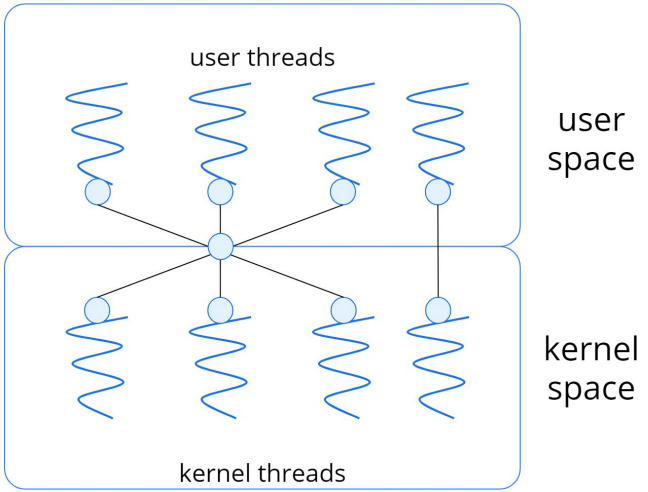


Modelos de Multithreading

- Consideraciones de concurrencia:
  - Mientras que el modelo **muchos a uno** le permite al desarrollador crear tantos hilos de usuario como desee, no produce paralelismo, porque el *kernel* puede planificar solo un hilo del *kernel* a la vez.
  - El modelo **uno a uno** permite una mayor concurrencia, pero el desarrollador debe tener cuidado de no crear demasiados hilos dentro de una aplicación (de hecho, en algunos sistemas, puede estar limitada la cantidad de hilos que se pueden crear).
  - El modelo de **muchos a muchos** no tiene ninguno de estos inconvenientes:
    - Los desarrolladores pueden crear tantos hilos de usuario como sea necesario y los correspondientes hilos del kernel pueden ejecutarse en paralelo en un multiprocesador.
    - Cuando un hilo realiza una llamada al sistema bloqueante, el *kernel* puede planificar otro hilo para su ejecución.

Modelos de Multithreading

- **Modelo de dos niveles:** una variación en el modelo de **muchos a muchos** multiplexa muchos hilos a nivel de usuario a un número menor o igual de hilos del *kernel*, pero también permite que un hilo de nivel de usuario se vincule a un hilo del *kernel*.
- Aunque el modelo de **muchos a muchos** parece ser el más flexible de los modelos discutidos, en la práctica es difícil de implementar.
- Además, con un número creciente de *cores* de procesamiento que aparecen en la mayoría de los sistemas, limitar el número de hilos del *kernel* se ha vuelto menos importante.
- Como resultado, la mayoría de los SO usan el modelo **uno a uno**.



Bibliotecas de hilos

- Una biblioteca de hilos proporciona al programador una *API* para crear y administrar hilos.
- Hay dos formas principales de implementarla:
  - **Primer enfoque**: proporcionar una biblioteca ubicada en el espacio de usuario sin soporte de *kernel*.
    - Códigos y estructuras de datos para la biblioteca en espacio de usuario.
    - Invocar una función en la biblioteca da como resultado una llamada de función local en el espacio del usuario y no una llamada de sistema.
  - **Segundo enfoque**: implementar una biblioteca a nivel de *kernel* soportada directamente por el SO.
    - Códigos y estructuras de datos para la biblioteca en espacio de *kernel*.
    - Invocar una función en la *API* para la biblioteca generalmente resulta en una llamada de sistema al *kernel*.
- Hoy en día las principales bibliotecas de hilos son: *POSIX Pthreads*, *Windows Threads*, y *Java Threads*.

Bibliotecas de hilos

- Existen dos estrategias generales para crear múltiples hilos:
  - **Hilos asincrónicos**: una vez que el padre crea un hilo hijo, el padre reanuda su ejecución, de modo que el padre y el hijo se ejecutan de manera simultánea e independiente uno del otro.
    - Debido a que los hilos son independientes, generalmente hay poco intercambio de datos entre ellos.
  - **Hilos sincrónicos**: se produce cuando el hilo principal crea uno o más hilos hijos y luego debe esperar a que todos sus hilos hijos finalicen antes de reanudarse.
    - Los hilos hijos realizan el trabajo en simultáneo, pero el padre no puede continuar hasta que este trabajo se haya completado.
    - Una vez finalizado cada hilo hijo se une (*joins*) con su padre. Solo después de que todos los hijos se hayan unido (*joined*), el padre puede reanudar la ejecución.
    - Por lo general, los hilos sincrónicos implican un intercambio de datos significativo entre ellos. Por ej.: el hilo padre puede combinar los resultados por sus hijos.

Bibliotecas de hilos

Pthreads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int sum;  /* this data is shared by the thread(s) */
/* The thread will execute in this function */
void *runner(void *param){
    long i,
    upper=(long)param;
    sum = 0;
    for (i=1; i<=upper; i++){
        sum += i;
    }
    pthread_exit(0);
}
int main(void){

    pthread_t tid;          /* the thread identifier */
    pthread_attr_t attr;    /* set of thread attributes */
    long x = 9;
    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, (void *)x);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
}
```

Ejemplo Linux

- Pthreads se refiere al estándar *POSIX* que define una *API* para la creación y sincronización de hilos.
- Esta es una especificación para el comportamiento del hilo, no una implementación.
- Los diseñadores de SO pueden implementar la especificación de la forma que deseen.

Bibliotecas de hilos

Windows *Threads*

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param) {
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
int main(int argc, char* argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    //Param = atoi(argv[1]);
    Param = 9;

    /* 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 */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);
    /* close the thread handle */
    CloseHandle(ThreadHandle);
    printf("sum = %d\n", Sum);
}
```

Ejemplo Windows

Compilado y ejecutado con *visual studio community 2019*

Bibliotecas de hilos

Java *Threads*

```java
package prueba01;
import java.util.concurrent.*;

class Summation implements Callable<Integer> {
    private int upper;
    public Summation(int upper) {
        this.upper = upper;
    }
    /* The thread will execute in this method */
    public Integer call() {
        int sum = 0;
        for (int i = 1; i <= upper; i++)
            sum += i;
        return new Integer(sum);
    }
}

public class Driver {
    public static void main(String[] args) {
        //int upper = Integer.parseInt(args[0]);
        int upper = 9;
        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));
        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}
```

Ejemplo Java

Compilado y ejecutado con *Eclipse IDE for Enterprise Java Developers.*

Hilos implícitos

- Con el continuo crecimiento del procesamiento *multicore*, las aplicaciones contendrán cientos o incluso miles de hilos.
- Diseñar tales aplicaciones no es una tarea trivial: los programadores deben abordar no solo los desafíos descritos anteriormente, sino también dificultades adicionales, como la correctitud.
- Una forma de abordar estas dificultades y mejorar el diseño de aplicaciones concurrentes y paralelas es transferir la creación y administración de hilos a los compiladores y bibliotecas en tiempo de ejecución.
- Estas estrategias generalmente requieren que los desarrolladores identifiquen tareas, no hilos, que pueden ejecutarse en paralelo.
- Una tarea puede escribirse como una función, que la biblioteca en tiempo de ejecución asigna a un hilo (generalmente usando el modelo de **muchos a muchos**).
- La ventaja de este enfoque es que los desarrolladores solo necesitan identificar tareas paralelas y las bibliotecas determinan los detalles específicos de la creación y administración de hilos.

Hilos implícitos

Thread Pools

- Consiste en crear un número de hilos al inicio y colocarlos en un *pool*, donde esperan por trabajo.
- Cuando el servidor recibe una solicitud, en lugar de crear un hilo, envía la solicitud al *thread pool* y reanuda la espera de solicitudes adicionales.
- Si hay un hilo disponible en el *pool*, se activa y la solicitud se atiende de inmediato. Si el *pool* no contiene ningún hilo disponible, la tarea se encola hasta que uno queda libre.
- Una vez que un hilo completa su servicio, vuelve al *pool* y espera más trabajo.
- Los *thread pools* ofrecen estos beneficios:
  - Atender una solicitud con un hilo existente es generalmente más rápido que crear uno.
  - Un *thread pool* limita el número de hilos existentes en cualquier momento, esto es importante en sistemas que no pueden soportar una gran cantidad de hilos concurrentes.
  - Separar la tarea a realizar de la mecánica de creación de dicha tarea nos permite utilizar diferentes estrategias para ejecutarla. Por ej., la tarea podría programarse para ejecutarse después de un retraso de tiempo o periódicamente.

Hilos implícitos

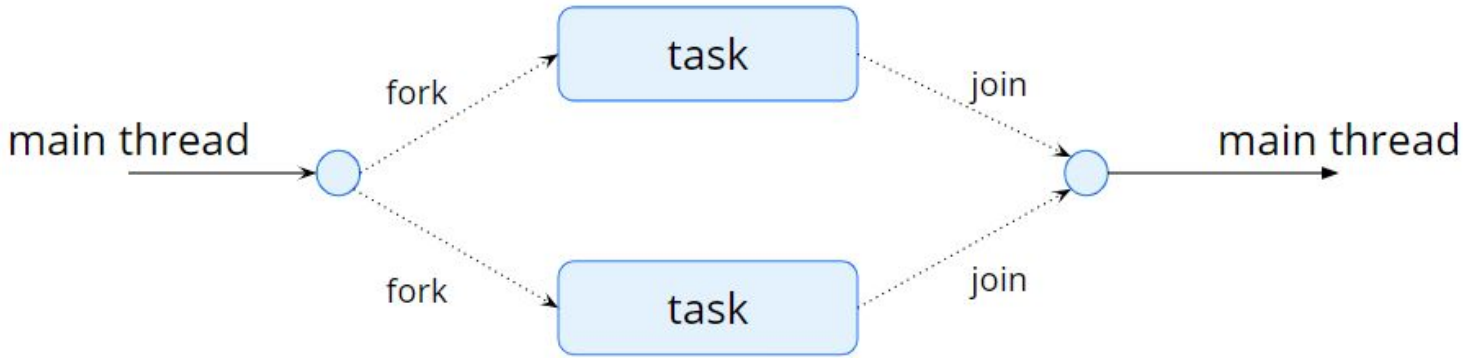
Thread Pools

- El número de hilos en el *pool* puede establecerse de forma heurística en función de factores como el número de *CPUs* en el sistema, la cantidad de memoria física y el número esperado de solicitudes de clientes concurrentes.
- Las arquitecturas de *thread pools* más sofisticadas pueden ajustar dinámicamente el número de hilos en el *pool* de acuerdo con los patrones de uso.
- Dichas arquitecturas brindan el beneficio adicional de tener un grupo más pequeño, consumiendo así menos memoria, cuando la carga en el sistema es baja.
- La *Windows API* provee varias funciones relacionadas a *thread pool*.
- El paquete `java.util.concurrent` de *Java* incluye una *API* para varias arquitecturas *thread-pool*.

Hilos implícitos

Fork Join

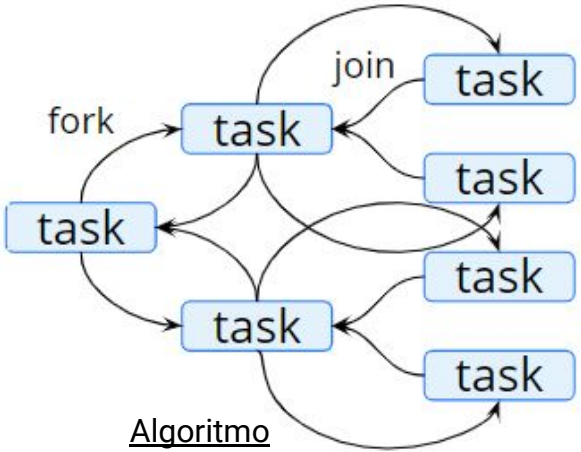
- En el **modelo *fork-join*** el hilo padre principal crea (*forks*) uno o más hilos hijos y luego espera a que los hijos terminen y se unan (*join*) con él, momento en el que puede recuperar y combinar sus resultados.
- Este modelo se caracteriza por la creación explícita de hilos, pero también se puede usar para hilos implícitos.
- Los hilos no se crean directamente durante la etapa de *fork*; sino que se designan tareas paralelas.
- Una biblioteca gestiona el número de hilos que se crean y también es responsable de asignar tareas a los hilos.



Hilos implícitos

Fork Join

- *Java* introdujo una biblioteca *fork-join* a la *API* que está diseñada para ser utilizada con algoritmos recursivos de divide y vencerás, como *Quicksort* y *Mergesort*.
- Al implementar algoritmos de divide y vencerás usando esta biblioteca, se bifurcan tareas separadas durante el paso de división, asignando subconjuntos más pequeños del problema original.
- Los algoritmos deben estar diseñados para que estas tareas separadas puedan ejecutarse simultáneamente.
- En algún momento, el tamaño del problema asignado a una tarea es lo suficientemente pequeño como para poder resolverlo directamente y no crear tareas adicionales.



Algoritmo

```
Task(problem)
 if problem is small enough
  solve the problem directly
 else
  subtask1 = fork(new Task(subset of problem))
  subtask2 = fork(new Task(subset of problem))

  result1 = join(subtask1)
  result2 = join(subtask2)

  return combined results
```

Hilos implícitos

OpenMP

- *OpenMP* es un conjunto de directivas de compilación, así como una *API* para programas escritos en C, C ++ o *FORTRAN,* que proporciona soporte para programación paralela en entornos de memoria compartida.
- *OpenMP* identifica regiones paralelas como bloques de código que pueden ejecutarse en paralelo.
- Los desarrolladores de aplicaciones insertan directivas de compilación en su código en regiones paralelas, y éstas directivas instruyen a la biblioteca de tiempo de ejecución de *OpenMP* para ejecutar la región en paralelo.
- El siguiente programa en C ilustra una directiva del compilador sobre la región paralela que contiene la instrucción `printf()`:
- Cuando *OpenMP* encuentra la directiva:
  - `#pragma omp parallel`
- Crea tantos hilos como *cores* haya en el sistema.

```c
/*gcc -fopenmp helloworld.c*/
#include <stdio.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Hilos implícitos

OpenMP

- Para un sistema de doble *core*, se crean dos hilos; para un sistema de cuatro *core*, se crean cuatro; etc.
- Todos los hilos ejecutan simultáneamente la región paralela y a medida que cada hilo sale de dicha región termina.
- *OpenMP* proporciona varias directivas adicionales para ejecutar regiones de código en paralelo, incluidos bucles de paralelización:

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

- *OpenMP* permite elegir entre varios niveles de paralelismo. Por ej.: es posible establecer el número de hilos manualmente.
- También permite identificar si los datos se comparten entre los hilos o son privados.

Hilos implícitos

Grand Central Dispatch

- *Grand Central Dispatch (GCD)* es una tecnología desarrollada por Apple para sus SOs *macOS* e *iOS*.
- Es una combinación de una biblioteca de tiempo de ejecución, una *API* y extensiones de lenguaje que permiten a los desarrolladores identificar secciones de código (*tasks*) para ejecutar en paralelo. Al igual que *OpenMP*, *GCD* gestiona la mayoría de los detalles de los hilos.
- *GCD* planifica las tareas para ser ejecutadas en tiempo de ejecución, colocándolas en una cola de despacho (*dispatch queue*). Cuando elimina una tarea de una cola, asigna la tarea a un hilo disponible de un *pool* de hilos que administra.
- *GCD* identifica dos tipos de colas de despacho: serial y concurrente.
- Las tareas colocadas en una cola en serie se eliminan en orden *FIFO*. Una vez que una tarea se ha eliminado de la cola, debe completar la ejecución antes de eliminar otra tarea. Cada proceso tiene su propia cola en serie. Las colas en serie son útiles para garantizar la ejecución secuencial de varias tareas.
- Las tareas colocadas en una cola concurrente también se eliminan en el orden *FIFO*, pero se pueden eliminar varias tareas a la vez, lo que permite que se ejecuten varias en paralelo.

Hilos implícitos

Intel Thread Building Blocks (TBB)

- Biblioteca de *templates* para el diseño de aplicaciones paralelas en C++. Aprovecha el rendimiento *multicore*, es portable y escalable.
- Como se trata de una biblioteca, no requiere un compilador especial. Los desarrolladores especifican tareas que pueden ejecutarse en paralelo y el planificador de tareas *TBB* las asigna en hilos.
- Además, el planificador de tareas proporciona equilibrio de carga y es compatible con la caché, lo que significa que dará prioridad a las tareas que tengan sus datos almacenados en la caché y, por lo tanto, se ejecutarán más rápidamente.
- *TBB* proporciona un amplio conjunto de características, que incluyen *templates* para estructuras de bucles paralelos, operaciones atómicas y exclusión mutua.

```
void SerialApplyFoo( float a[], size_t n )
{
     for( size_t i=0; i!=n; ++i )
          Foo(a[i]);
}
```

```
#include "tbb/tbb.h"
using namespace tbb;
void ParallelApplyFoo( float a[], size_t n ){
  tbb::parallel_for( size_t(0), n, [&]( size_t i ){
            Foo(a[i]);
  });
}
```

Problemas de hilos

Llamadas al sistema fork() y exec()

- La semántica `fork()` y `exec()` cambia en un programa multi-hilado.
- Si un hilo en un programa llama a `fork()`, ¿el nuevo proceso duplica todos los hilos o el nuevo proceso es mono-hilado? Algunos sistemas UNIX han optado por tener dos versiones de `fork()`, una que duplica todos los hilos y otra que duplica solo el hilo que invoca la llamada al sistema `fork()`.
- La llamada al sistema `exec()` generalmente funciona de la misma manera: si un hilo invoca a `exec()`, el programa especificado en el parámetro de `exec()` reemplazará todo el proceso, incluidos todos los hilos.
- El tipo de aplicación determina cuál de las dos versiones de fork() usar. Si se llama a `exec()` inmediatamente después del *fork*, no es necesario duplicar todos los hilos, ya que el programa especificado en los parámetros de `exec()` reemplazará el proceso.
- En el caso anterior, es apropiado duplicar solo el hilo que hace la llamada.
- Sin embargo, si el proceso separado no llama a `exec()` después del *fork*, se deberían duplicar todos los hilos.
- Queda como ejercicio para el alumno investigar que sucede en Linux usando *Pthreads*.

Problemas de hilos

Manejo de señales

- En sistemas UNIX se utiliza una señal para notificar a un proceso que se ha producido un evento en particular.
- Se puede recibir una señal de forma sincrónica o asincrónica, dependiendo de la fuente y el motivo del evento que se está señalando.
- Todas las señales, ya sean sincrónicas o asincrónicas , siguen el mismo patrón:
  - Una señal es generada por la ocurrencia de un evento particular.
  - La señal se entrega a un proceso.
  - Una vez entregada, la señal debe ser manejada.
- Ejemplos de señales sincrónicas incluyen el acceso ilegal a memoria y la división por 0.
- Si un programa en ejecución realiza cualquiera de estas acciones, se genera una señal. Estas señales se consideran sincrónicas porque se entregan al mismo proceso que realizó la operación que causó la señal.

Problemas de hilos

Manejo de señales

- Cuando una señal se genera por un evento externo al proceso que se está ejecutando, ese proceso recibe la señal de forma asincrónica.
- Ejemplos de tales señales incluyen la finalización de un proceso con pulsaciones de teclas específicas (como `<control><C>`) y la expiración de un temporizador.
- Por lo general, se envía una señal asincrónica a otro proceso.
- Una señal puede ser manejada por uno de dos posibles manejadores:
  - Manejador de señal por defecto.
  - Manejador de señal definido por el usuario.
- Cada señal tiene un manejador de señal predeterminado que el *kernel* ejecuta cuando maneja esa señal.
- Esta puede ser anulada por un manejador de señal definido por el usuario.
- Las señales se manejan de diferentes maneras; algunas señales pueden ignorarse, mientras que otras (por ej., un acceso ilegal a la memoria) se manejan al finalizar el programa.

Problemas de hilos

Manejo de señales

- El manejo de señales en programas de un solo hilos es sencillo: las señales siempre se entregan a un proceso.
- Sin embargo, en programas multi-hilados la entrega de señales es más complicada.
- ¿Dónde, entonces, se debe entregar una señal? En general, existen las siguientes opciones:
  - Entregar la señal al hilo al que se aplica la señal.
  - Entregar la señal a cada hilo en el proceso.
  - Entregar la señal a ciertos hilos en el proceso.
  - Asignar un hilo específico para recibir todas las señales del proceso.
- El método para entregar una señal depende del tipo de señal generada.
- Por ej., las señales sincrónicas deben entregarse al hilo que causa la señal y no a otros hilos en el proceso.
- Sin embargo, la situación con las señales asincrónicas no es tan clara. Algunas señales asincrónicas, como una señal que termina un proceso (`<control><C>`), deben enviarse a todos los hilos.

Problemas de hilos

Cancelación de hilos

- La cancelación de hilos implica terminar un hilo antes de que se haya completado. Por ej., si varios hilos buscan en una base de datos y un hilo devuelve el resultado, los hilos restantes podrían cancelarse.
- Un hilo que se cancelará se denomina hilo objetivo (*target thread*).
- La cancelación de un hilo objetivo puede ocurrir en dos escenarios diferentes:
  - **Cancelación asincrónica:** un hilo termina inmediatamente el hilo objetivo.
  - **Cancelación diferida:** el hilo objetivo verifica periódicamente si debe terminar, lo que brinda la oportunidad de terminar de manera ordenada.
- La dificultad con la cancelación se produce en situaciones en las que se han asignado recursos a un hilo cancelado o donde se cancela un hilo mientras se está actualizando datos que comparte con otros hilos. Esto es problemático con la cancelación asincrónica.
- En cambio, con la cancelación diferida, un hilo indica que se debe cancelar un hilo objetivo, pero la cancelación se produce sólo después de que el hilo objetivo haya verificado un indicador para determinar si se debe cancelar o no.
- En *Pthreads*, la cancelación de hilos se inicia utilizando la función `pthread_cancel()`.

Problemas de hilos

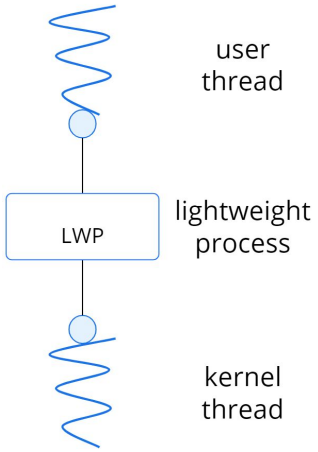
Thread-Local Storage

- Los hilos que pertenecen a un proceso comparten sus datos, este intercambio de datos proporciona uno de los beneficios de la programación multi-hilada.
- En algunas circunstancias, cada hilo puede necesitar una copia de ciertos datos propios, lo que se denomina almacenamiento local de hilo (*thread-local storage* o *TLS*).
- Ej.: en un sistema de procesamiento de transacciones con identificador único. Para asociar cada hilo con un identificador de transacción único podríamos usar *TLS*.
- Es fácil confundir *TLS* con variables locales. Las variables locales son solo visibles durante una invocación de una función, mientras que *TLS* son incluso visibles entre distintas invocaciones de funciones.
- Cuando el desarrollador no tiene control sobre el proceso de creación de hilo, por ej. con una técnica implícita, se necesita un enfoque alternativo.
- *TLS* es similar a los datos estáticos la diferencia es que los datos *TLS* son únicos para cada hilo, de hecho, *TLS* generalmente se declara como estático.
- La mayoría de las bibliotecas y compiladores de hilos brindan soporte para *TLS*.

Problemas de hilos

Scheduler Activations

- Un problema final que se debe considerar con los programas multiproceso se refiere a la comunicación entre el *kernel* y la biblioteca de hilos, que pueden ser requeridos por los modelos de **muchos a muchos** y de dos niveles.
- Dicha coordinación permite que el número de hilos del *kernel* se ajuste dinámicamente para ayudar a garantizar el mejor rendimiento.
- Muchos sistemas que implementan el modelo **muchos a muchos** o de dos niveles colocan una estructura de datos intermedia (*lightweight process*, o *LWP*) entre el usuario y los hilos del *kernel*.
- Para la biblioteca de hilos de usuario, el *LWP* se comporta como un procesador virtual en el que la aplicación puede programar la ejecución de un hilo de usuario.
- Cada *LWP* está conectado a un hilo del *kernel* y son los hilos del *kernel* los que el SO programa para ejecutarse en procesadores físicos.
- Si un hilo del *kernel* se bloquea el *LWP* también se bloquea. Lo que provoca que el hilo de nivel de usuario conectado al *LWP* también se bloquee.



Problemas de hilos

Scheduler Activations

- Un esquema para la comunicación entre la biblioteca de hilos de usuario y el *kernel* se conoce como activación del planificador (*scheduler activation*).
- Funciona de la siguiente manera: el *kernel* proporciona a una aplicación un conjunto de procesadores virtuales (*LWP*) y la aplicación puede planificar hilos de usuario en un procesador virtual disponible.
- Además, el *kernel* debe informar a una aplicación sobre ciertos eventos. Este procedimiento se conoce como una llamada ascendente (*upcall*). La biblioteca de hilos maneja las llamadas ascendentes con un controlador de llamadas ascendentes que debe ejecutarse en un procesador virtual.
- Un evento que desencadena una llamada ascendente ocurre cuando un hilo de aplicación está a punto de bloquearse.
- En este escenario, el *kernel* realiza una llamada ascendente a la aplicación informándole que un hilo está a punto de bloquearse. El *kernel* luego asigna un nuevo procesador virtual a la aplicación.

Ejemplos de sistemas operativos

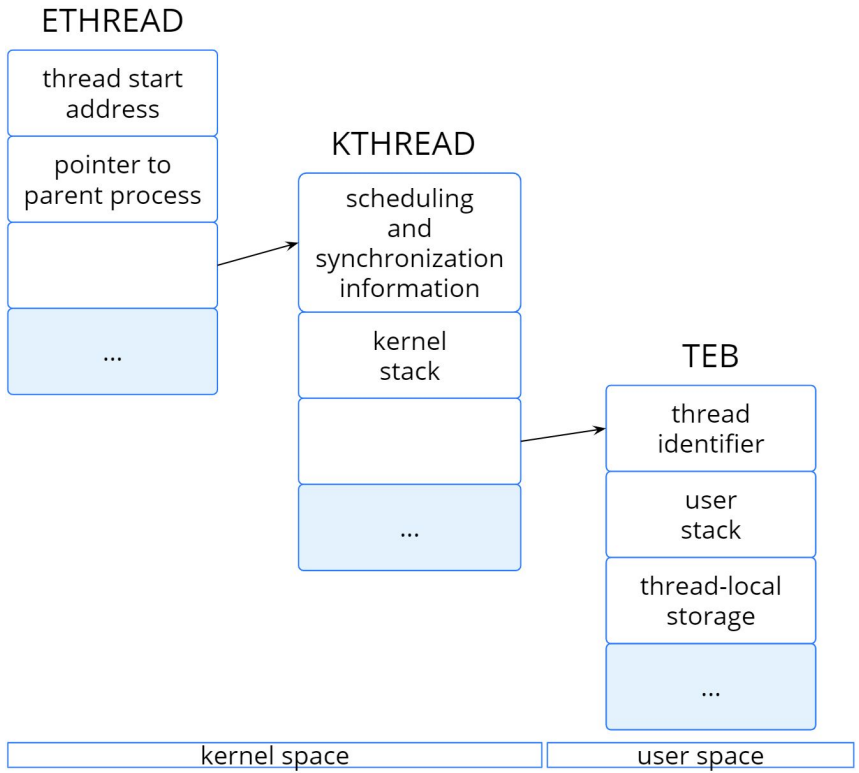
Windows *Threads*

- Una aplicación de Windows se ejecuta como un proceso separado y cada proceso puede contener uno o más hilos. Además, Windows utiliza la asignación **uno a uno** donde a cada hilo a nivel de usuario se le asigna un hilo del *kernel*.
- Los componentes generales de un hilo incluyen:
  - Una *ID* de hilo que lo identifica de forma exclusiva.
  - Un conjunto de registros que representa el estado del procesador.
  - Un contador de programa.
  - Una pila de usuarios, empleada cuando el hilo se ejecuta en modo de usuario, y una pila de *kernel*, empleada cuando el hilo se ejecuta en modo *kernel.*
  - El área de almacenamiento privado utilizada por varias bibliotecas de tiempo de ejecución y bibliotecas de enlaces dinámicos (*DLL*).
- El conjunto de registros, pilas y el área de almacenamiento privado se conocen como el contexto del hilo.

Ejemplos de sistemas operativos

Windows *Threads*

- Las estructuras de datos principales de un hilo incluyen:
  - *ETHREAD*: bloque de hilo ejecutivo.
  - *KTHREAD*: bloque de hilo del *kernel*.
  - *TEB*: bloque de entorno de hilos.



Ejemplos de sistemas operativos

Linux *Threads*

- Linux provee la llamada al sistema `fork()` con la funcionalidad tradicional de duplicar un proc., pero también proporciona la capacidad de crear hilos usando la llamada al sistema **`clone()`**.
- Linux no distingue entre procesos e hilos, y en su lugar usa el término tarea refiriéndose a un flujo de control dentro de un programa.
- Cuando se invoca a `clone()`, se pasa un conjunto de *flags* que determinan cuánto se compartirá entre las tareas padres e hijas.

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.

- Usar `clone()` con estos *flags* es equivalente a crear un hilo, ya que la tarea padre comparte la mayoría de sus recursos con la tarea hija.
- Sin embargo, si no se establece ninguno de estos *flags,* cuando se invoca `clone()` no se realiza ningún uso compartido, lo que resulta en una funcionalidad similar a la proporcionada por la llamada al sistema `fork()`.