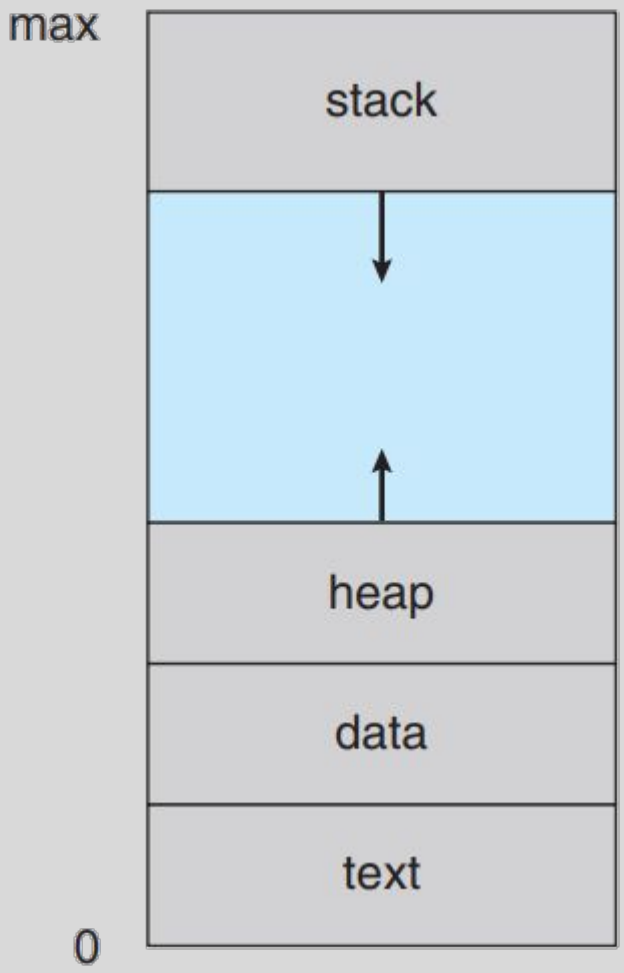
ARQUITECTURA Y SISTEMAS OPERATIVOS

CLASE 4

HILOS (SUBPROCESOS) Y CONCURRENCIA

REPASO



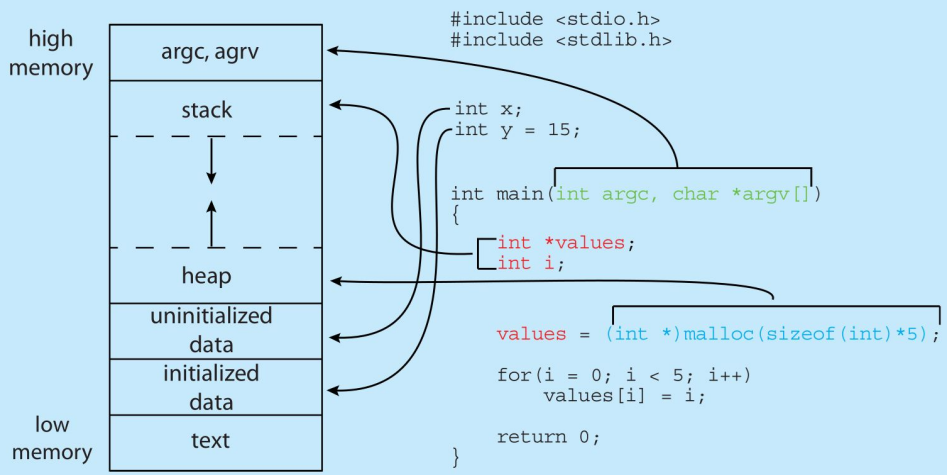
EL PROCESO

El diagrama de un proceso en memoria se divide en secciones:

- *Text*: el código ejecutable.
- *Data*: variables globales.
- *Heap*: memoria asignada dinámicamente durante la ejecución del programa.
- *Stack*: almacenamiento temporal al invocar funciones (parámetros, direcciones de retorno, variables locales).

Notar las secciones de texto y datos son fijas, no cambian durante la ejecución del programa. En cambio el stack y el heap se pueden expandir y contraer dinámicamente durante la ejecución del programa.

MEMORY LAYOUT OF A C PROGRAM



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to ***block started by symbol***.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

REPRESENTACIÓN DE LA MEMORIA DE UN PROGRAMA EN C

Esta imagen ilustra de manera más real la disposición de un programa en C en memoria. Notar las diferencias con la imagen anterior:

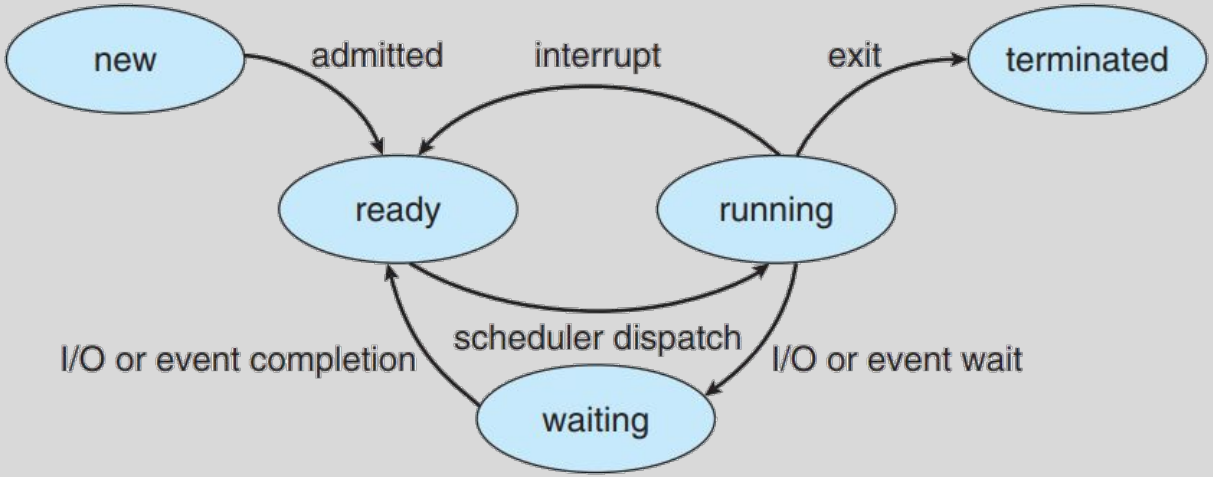
- La sección *data* (donde se almacenan las variables globales) se divide en (a) datos inicializados y (b) datos no inicializados.
- Además, existe una sección separada para los parámetros argc y argv pasados a la función `main()`.

ESTADOS

Mientras se ejecuta, el proceso cambia de estado.

- **Nuevo**: el proceso es creado.
- **Ejecutando**: las instrucciones están siendo ejecutadas.
- **Esperando**: el proceso espera la ocurrencia de un evento.
- **Listo**: el proceso espera ser asignado a un procesador.
- **Terminado**: el proceso ha finalizado su ejecución.

Es importante notar que solo un proceso puede estar ejecutándose en un núcleo del procesador en un instante. Sin embargo, muchos procesos pueden estar listos y esperando.



BLOQUE DE CONTROL (PCB)

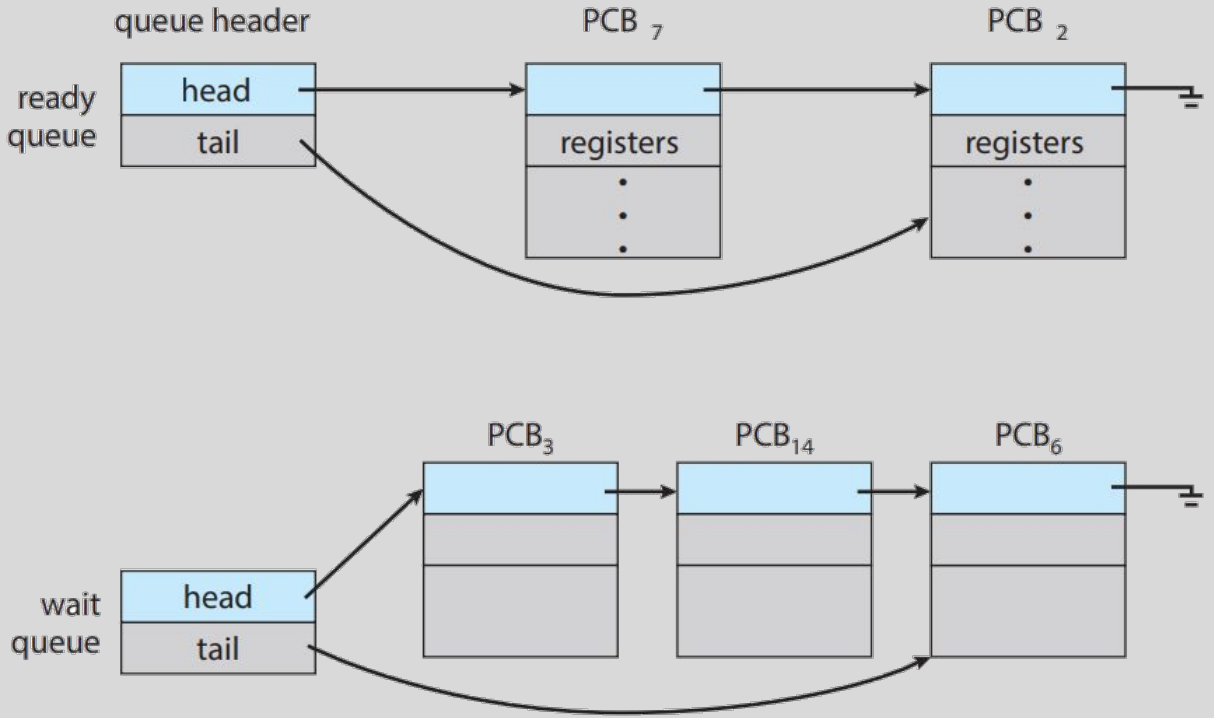
Cada proceso es representado en el sistema operativo por un **Bloque de Control de Procesos** (*Process Control Block*). El cual contiene mucha información asociada a un proceso:

- Estado del Proceso (new, ready, etc.)
- PC
- Registros del CPU.
- Información sobre la Planificación del CPU.
- Información sobre la Administración de Memoria.
- Información contable (cantidad de CPU, pid, etc).
- Información del Estado E/S.

COLAS DE PLANIFICACIÓN

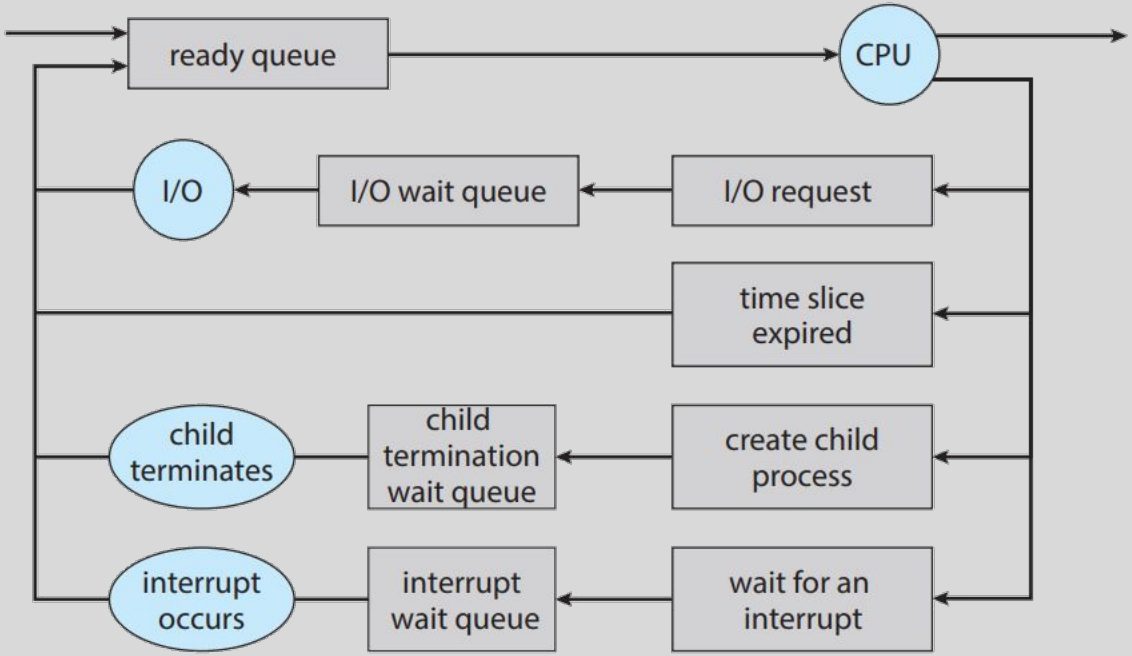
Cuando un proceso entra al sistema, se suma a la **cola de listos** (*ready*), donde esperan la asignación en un núcleo de la CPU.

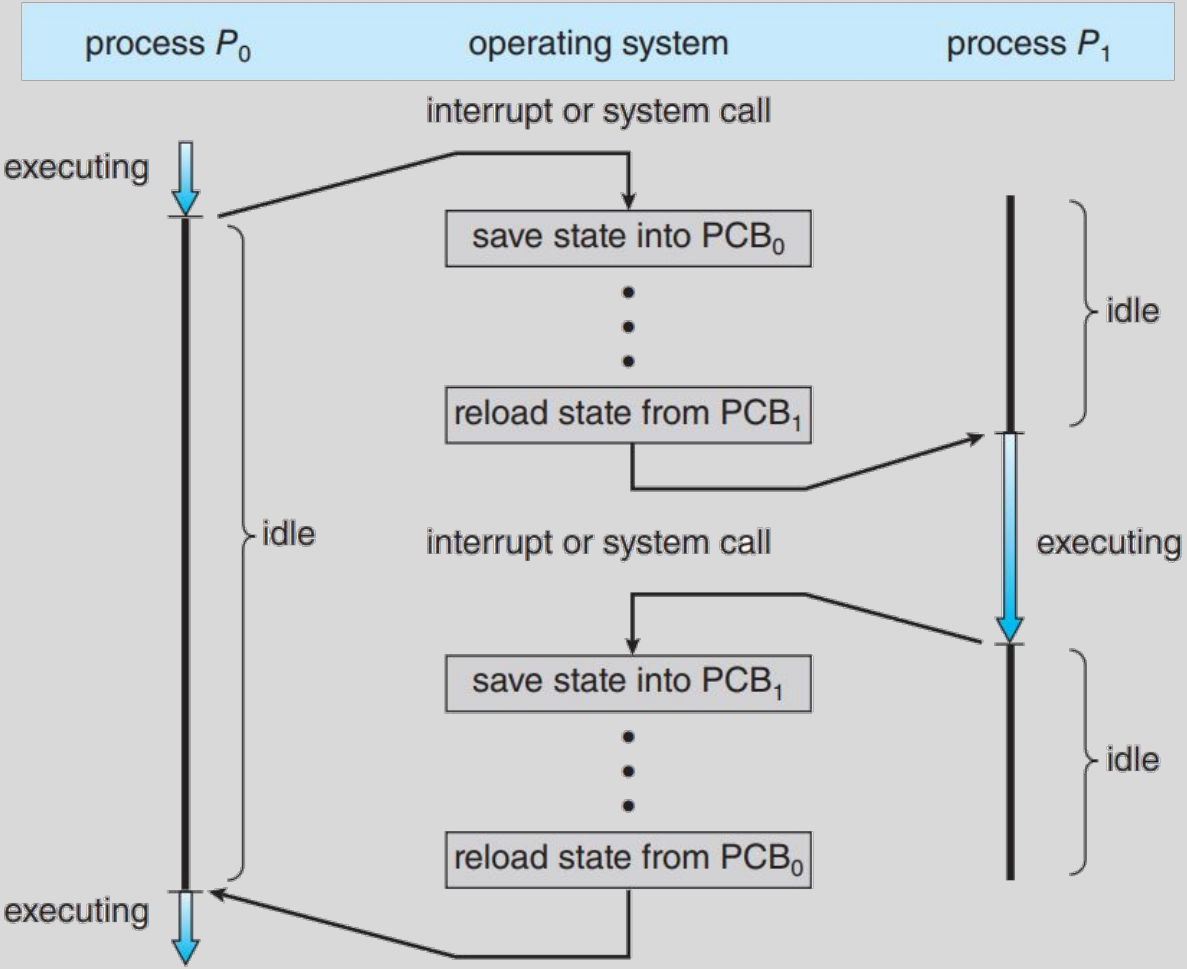
Cuando un proceso solicita una operación de E/S, tendrá que esperar a que la respuesta esté disponible. **Cola de espera** (*waiting*).



COLAS DE PLANIFICACIÓN

- El proceso puede emitir una solicitud de E/S y colocarse en una cola de espera de E/S.
- Puede crear un nuevo proceso hijo y luego colocarse en una cola de espera mientras espera su finalización.
- Puede ser eliminado forzosamente del núcleo, por resultado de una interrupción o por expiración del tiempo asignado, y volver a la cola de listos.
- O puede continuar este ciclo hasta que termina, momento en el que se elimina de todas las colas, y se liberan su PCB y sus recursos.





CAMBIO DE CONTEXTO

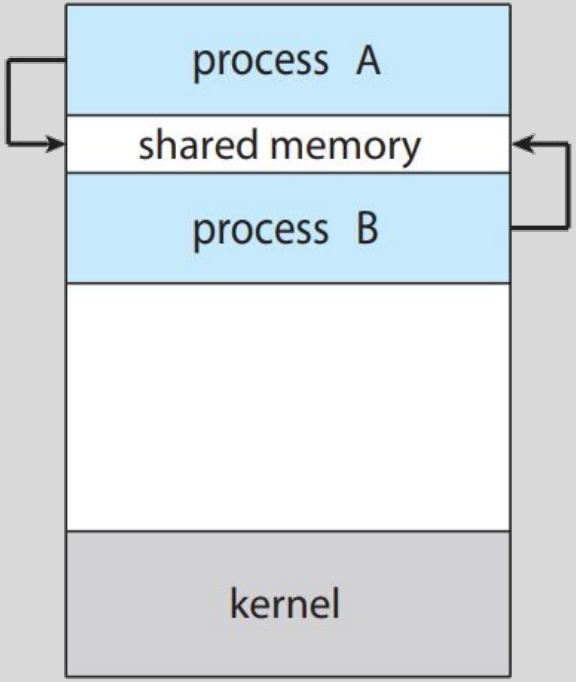
Cuando se produce una interrupción, el sistema necesita **guardar el contexto actual** del proceso que se ejecuta en un núcleo de la CPU, para poder restaurarlo una vez finalizado su procesamiento (o sea, **suspender** y luego **reanudar**).

El tiempo de cambio de contexto es pura **sobrecarga**, ya que el sistema no realiza ninguna **función útil** durante el cambio. Una velocidad típica es de varios microsegundos. Y dependen en gran medida de la compatibilidad del hardware.

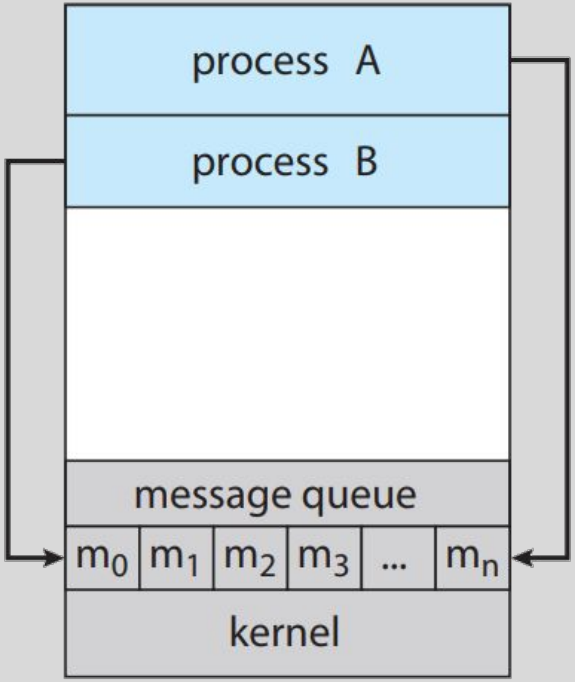
COMUNICACIÓN ENTRE PROCESOS

El **pasaje de mensajes** es útil para intercambiar pequeñas cantidades de datos. Además, es más fácil de implementar en sistemas distribuidos que la memoria compartida.

En los sistemas de **memoria compartida**, las *syscalls* solo se requieren para establecer las regiones de memoria compartida. Una vez establecida, todos los accesos se tratan como accesos rutinarios a la memoria y no se requiere la intervención del kernel.



Shared memory.



Message passing.

BIBLIOGRAFIA

- Operating System Concepts. By Abraham, Silberschatz.
  - Capítulo IV
- Modern Operating Systems. By Andrew S. Tanenbaum.
  - Capitulo II

TEMAS DE LA CLASE

- Descripción General
  - Motivación
  - Beneficios
- Programación Multinúcleo
  - Desafíos de Programación
  - Tipos de Paralelismo
- Modelos Multihilos
  - Modelo Muchos-a-Uno
  - Modelo Uno-a-Uno
  - Modelo Muchos-a-Muchos
- Librerías de Hilos
  - Pthreads
  - Windows Threads
  - Java Threads

TEMAS DE LA CLASE

- Problemas de Hilos
  - Las llamadas al sistema fork() y exec()
  - Manejo de señales
  - Cancelación de Hilos
  - Almacenamiento Local en Hilos
  - Programación de Activaciones
- Hilos implícitos
  - Pool de Hilos
  - Fork Join
  - OpenMP
  - Grand Central Dispatch

THREADS (HILOS) Y CONCURRENCIA

DESCRIPCIÓN GENERAL

Un *thread* (hilo) es la unidad de ejecución más pequeña dentro de un proceso. Mientras que un proceso es un programa independiente en ejecución con su propio espacio de memoria, un hilo representa **una única secuencia de instrucciones** dentro de ese proceso. Hoy en día, los sistemas operativos admiten el concepto de *multithread* (multitarea), que permite que varios hilos se ejecuten simultáneamente dentro de un mismo proceso.

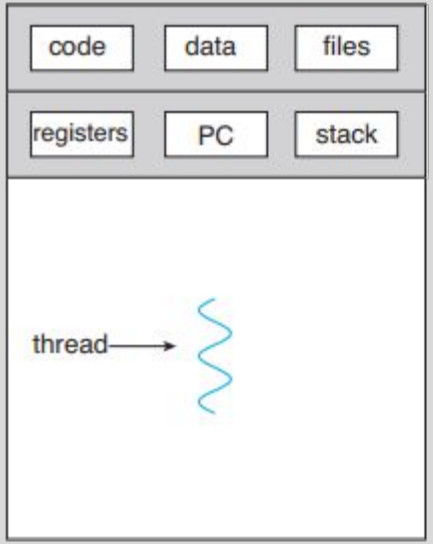
Un hilo es una unidad mínima de utilización del CPU; es un ID del *thread*, un PC, algunos registros y una *stack* propia. Si el proceso tiene más *threads*, todos comparten su sección *text* (código), *data* (variables globales) y otros recursos del sistema operativo, como archivos abiertos y señales.

Un proceso tradicional tiene un único *thread* de control. Si un proceso tiene varios *threads* de control, puede realizar más de una tarea a la vez.

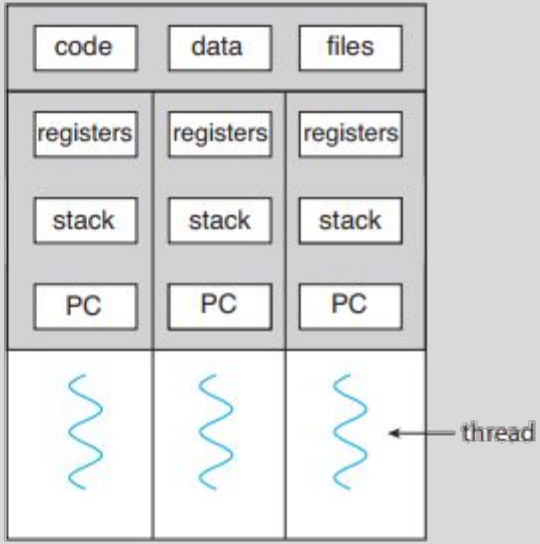
THREADS (HILOS)

La razón principal para tener hilos es que, en general las aplicaciones realizan múltiples actividades a la vez, y algunas de estas pueden bloquearse, esperando un dato por ejemplo. Al descomponer una aplicación de este tipo en múltiples hilos secuenciales que se ejecutan casi en paralelo, el modelo de programación se simplifica.

Con los hilos, añadimos un nuevo elemento: la capacidad de que las entidades puedan compartir un espacio de direcciones y todos sus datos. Esta capacidad es esencial para ciertas aplicaciones, donde tener múltiples procesos (cada uno con sus respectivos espacios de direcciones) no resulta funcional.



single-threaded process

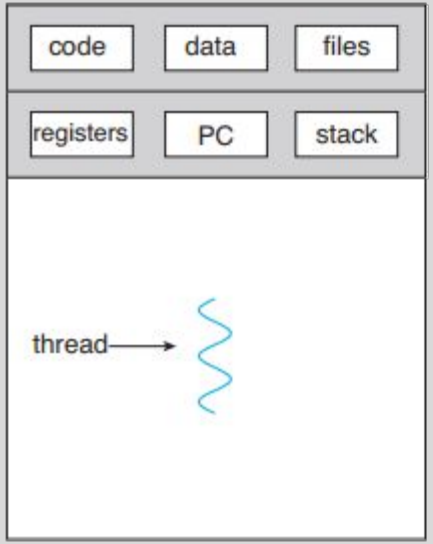


multithreaded process

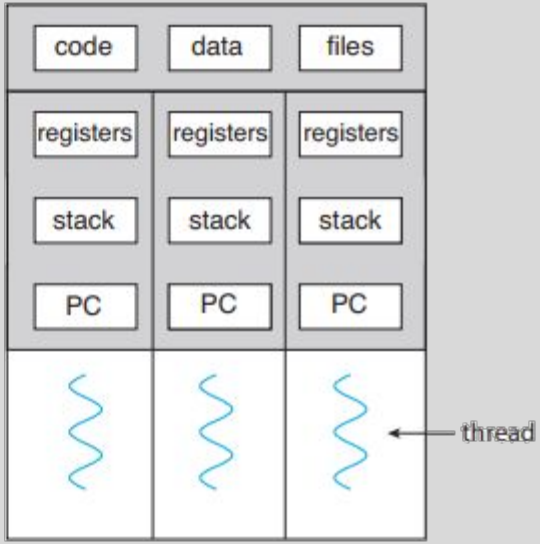
THREADS (HILOS)

Un segundo argumento para tener hilos es que, al ser más livianos que los procesos, son más fáciles (es decir, más rápidos) de crear y destruir. Para tener una referencia, en muchos sistemas crear un hilo es entre 10 y 100 veces más rápido que crear un proceso.

Una tercera razón para tener hilos también es un argumento de rendimiento. Los hilos no mejoran el rendimiento cuando todos están limitados por CPU, pero cuando hay un volumen de procesamiento y E/S considerable, tener hilos permite que estas actividades se superpongan, acelerando así la aplicación.



single-threaded process



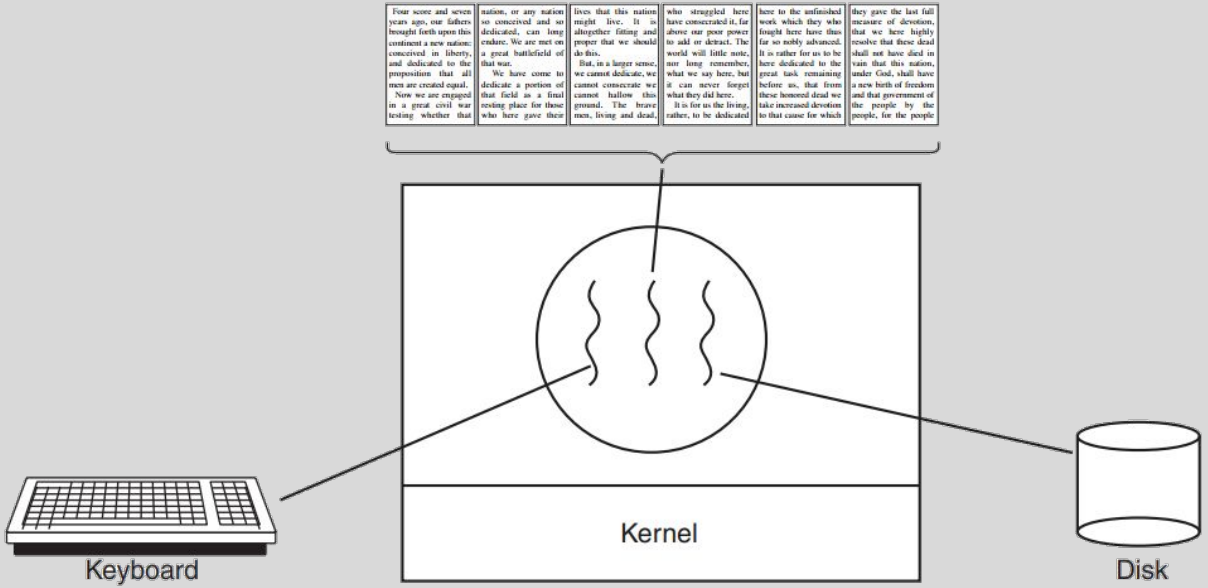
multithreaded process

HILOS - EJEMPLO

Consideren un procesador de texto simple que realiza 3 tareas en simultáneo:

1. Atender la E/S que ingresa el usuario por teclado.
2. Ejecutar la función de auto-corrector.
3. Realizar un *backup* del documento periódicamente.

Debe quedar claro que tener tres procesos separados no funcionaría, ya que las tres tareas deben operar en el mismo documento. Al tener tres hilos en lugar de tres procesos, estos comparten memoria y, por lo tanto, todos tienen acceso al documento que se está editando. Con tres procesos, esto sería imposible.

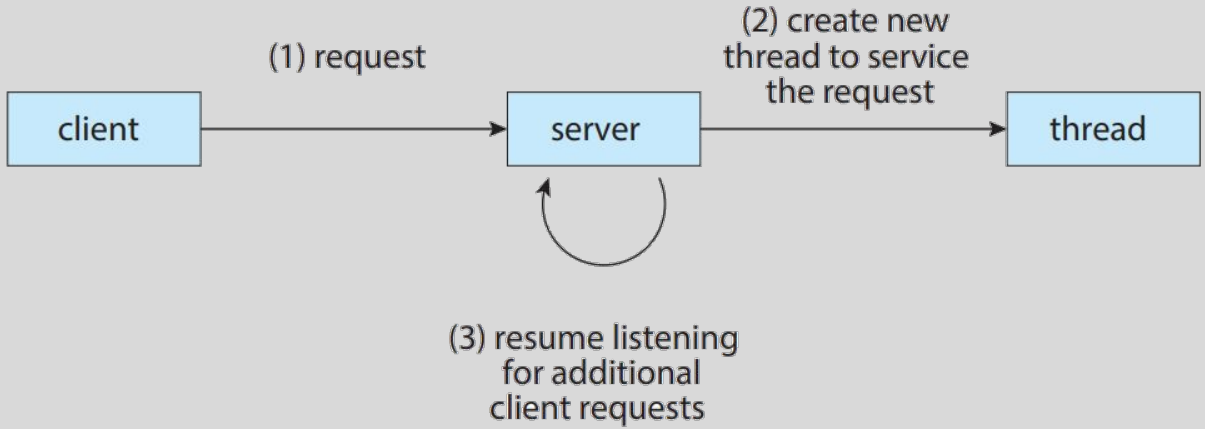


HILOS - EJEMPLO

Si un servidor web se ejecuta como un único proceso que acepta *requests*, cuando recibe una solicitud, crea un nuevo proceso para atenderla.

Pero como vimos, la creación de procesos es costosa. En cambio, si el nuevo proceso va a realizar las mismas tareas que el padre, generalmente es más eficiente usar un proceso que cree un hilo para atender la solicitud.

Si el proceso del servidor web es *multithread*, el servidor creará un nuevo hilo que escucha las solicitudes del cliente, así, cuando llega una una solicitud, en lugar de crear otro proceso, se crea un nuevo hilo para atenderla, y rápidamente reanuda la escucha de solicitudes siguientes.



BENEFICIOS

Capacidad de respuesta. El uso de múltiples hilos en una aplicación interactiva puede permitir que el programa continúe su ejecución incluso si una parte del mismo está bloqueada o está realizando una operación prolongada. Esto aumenta la capacidad de respuesta al usuario.

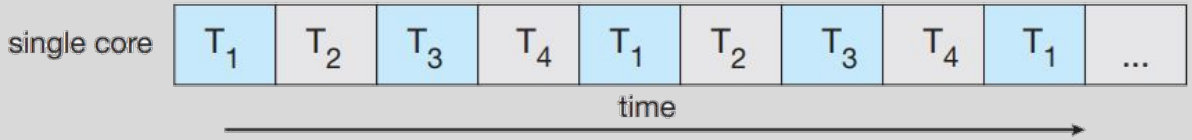
Uso compartido de recursos. Los procesos solo pueden compartir recursos mediante técnicas como la memoria compartida y pasaje de mensajes. Y deben ser configuradas explícitamente por el programador. Los hilos en cambio, comparten la memoria y los recursos del proceso al que pertenecen. Compartir código y datos permite que una aplicación tenga varios hilos de actividad diferentes dentro del mismo espacio de direcciones.

Economía. Asignar memoria y recursos para la creación de procesos es costoso. Como los hilos comparten los recursos del proceso al que pertenecen, es más económico su creación, eliminación y el cambio de contexto.

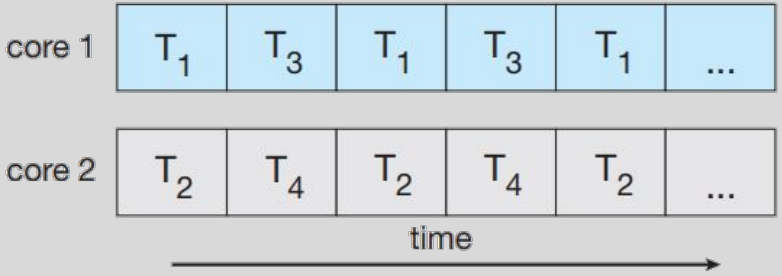
Escalabilidad. El uso de múltiples hilos en una arquitectura multiprocesador aumenta los beneficios, porque los hilos pueden ejecutarse en paralelo en diferentes núcleos. Un proceso de un solo hilo se ejecuta en un solo procesador, sin importar cuántos estén disponibles.

PROGRAMACIÓN EN SISTEMAS MULTINÚCLEO

MULTICORE



Concurrent execution on a single-core system.



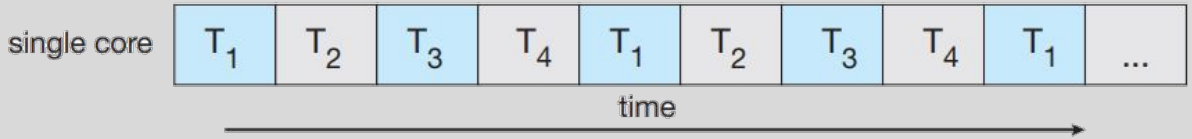
Parallel execution on a multicore system.

Por ejemplo: una aplicación con cuatro hilos.

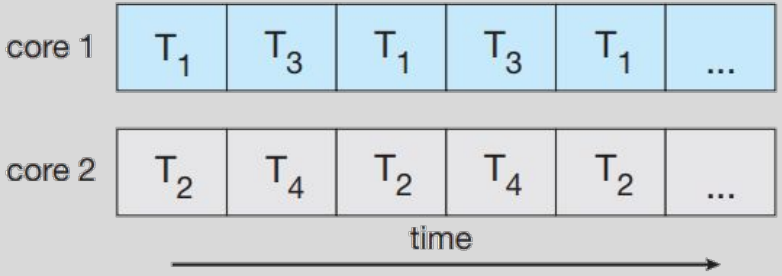
En un sistema *single-core*, la concurrencia simplemente significa que la ejecución de los hilos se intercalan en el tiempo, ya que el núcleo de procesamiento solo puede ejecutar un hilo a la vez.

Sistemas multinúcleo: arquitecturas con más de 1 núcleo de ejecución. La programación *multithread* brinda un mecanismo para un uso más eficiente de estas arquitecturas y una mejor concurrencia.

MULTICORE



Concurrent execution on a single-core system.



Parallel execution on a multicore system.

En un sistema *multicore*, la concurrencia significa que algunos hilos pueden ejecutarse en **paralelo**, ya que el sistema puede asignar un hilo independiente a cada núcleo.

Importante: notar la distinción entre concurrencia y paralelismo.

- Un sistema **concurrente** admite más de una tarea permitiendo que todas las tareas avancen.
- Un sistema **paralelo** puede realizar más de una tarea simultáneamente.

Es posible tener concurrencia sin paralelismo.

DESAFÍOS DE PROGRAMACIÓN

Para los diseñadores de sistemas operativos, el reto está en escribir algoritmos de programación que utilicen múltiples núcleos de procesamiento, permitiendo la ejecución paralela.

Para los programadores de aplicaciones, además de modificar los programas existentes, deben pensar cómo diseñar nuevos programas multihilo. En general se identifican 5 áreas:

1. **Identificación de tareas**. Examinar las aplicaciones para encontrar áreas que puedan dividirse en tareas independientes y concurrentes. Tareas independientes entre sí que, por lo tanto, pueden ejecutarse en paralelo en núcleos individuales.

2. **Equilibrio**. Asegurarse de que las tareas realicen el mismo trabajo y por el mismo costo. A veces, una tarea puede no aportar tanto valor al proceso general. Usar un núcleo de ejecución independiente para ejecutar esa tarea puede no justificar el costo.

3. **División de datos**. Al igual que las aplicaciones se dividen en tareas independientes, los datos a los que acceden y manipulan las tareas deben dividirse para ejecutarse en núcleos separados.

DESAFÍOS DE PROGRAMACIÓN

4. **Dependencia de datos**. Cuando una tarea depende de los datos de otra, los programadores deben asegurarse de que la ejecución de las tareas esté sincronizada para adaptarse a la dependencia de los datos.

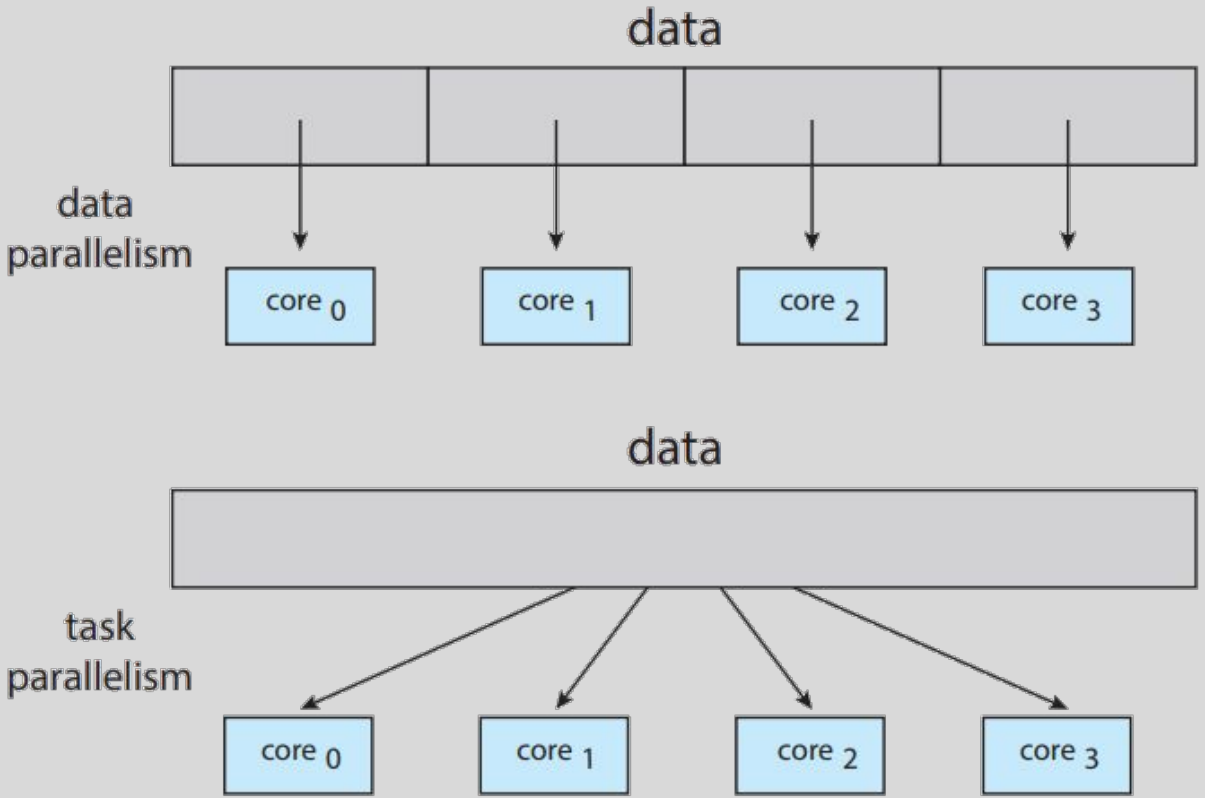
5. **Pruebas y depuración**. Cuando un programa se ejecuta en paralelo en varios núcleos, son posibles muchas rutas de ejecución diferentes. Probar y *debugguear* (permítanme el término) estos programas concurrentes es claramente más difícil que probar y depurar aplicaciones de un solo subproceso.

TIPOS DE PARALELISMO

Existen dos tipos de paralelismo: de datos y de tareas.

Paralelismo de datos: distribuir subconjuntos de datos entre múltiples núcleos y realizar la misma operación en cada uno.

Paralelismo de tareas: distribuir tareas (hilos) entre múltiples núcleos. Cada hilo realiza una operación única. Diferentes hilos pueden operar con los mismos datos o con datos diferentes.



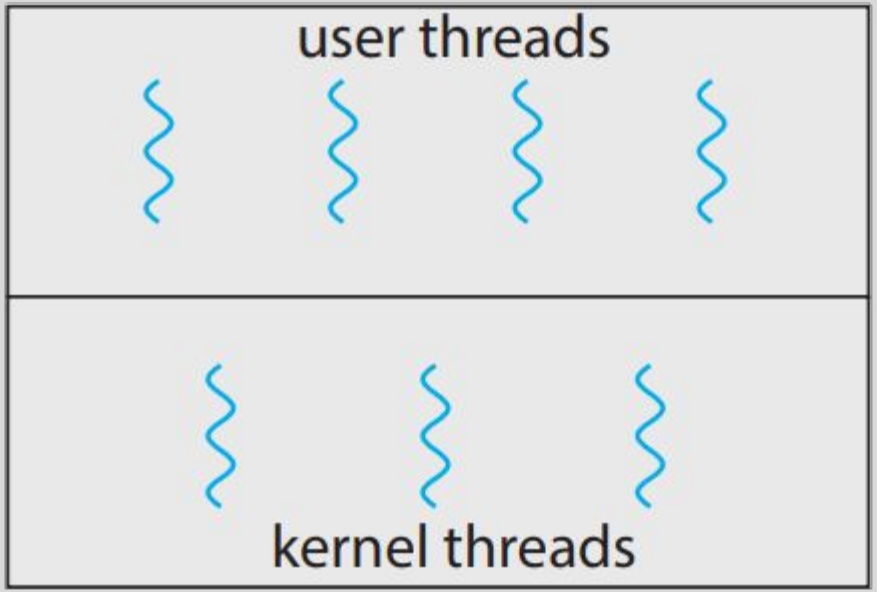
MODELOS MULTIHILOS

MODELOS MULTIHILOS

El soporte para hilos puede brindarse a nivel de usuario o por el kernel. Los hilos de usuario se gestionan sin soporte del kernel, mientras que los hilos de kernel son soportados y gestionados directamente por el sistema operativo.

En definitiva, debe existir una relación entre los hilos de usuario y los hilos de kernel. Comunmente, hay tres formas de establecer esta relación:

- muchos a uno
- uno a uno
- muchos a muchos



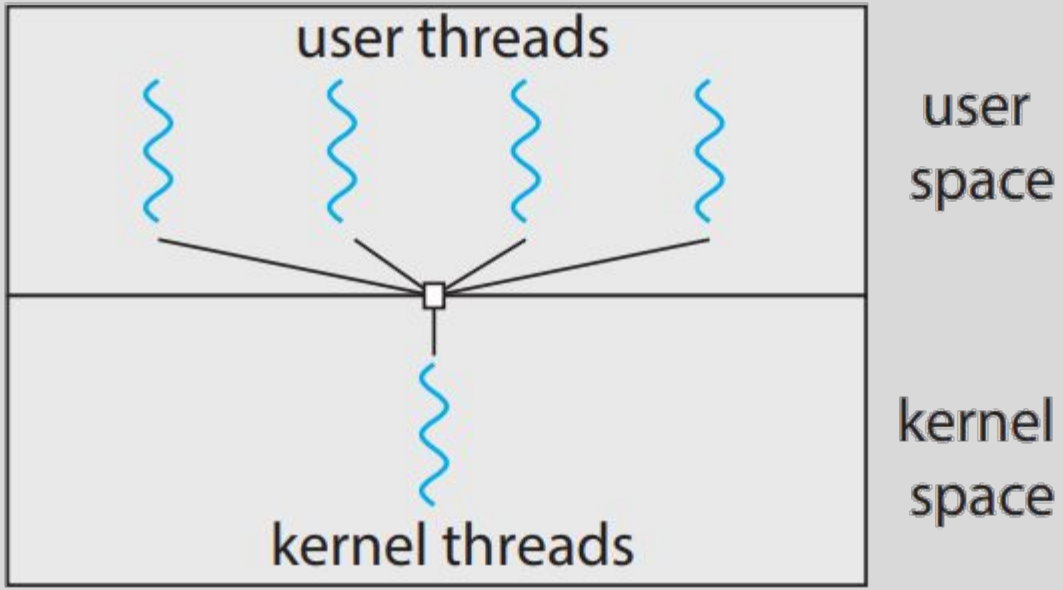
User and kernel threads.

MUCHOS A UNO

El modelo muchos a uno asigna varios hilos de nivel usuario a un hilo del *kernel*.

Es eficiente porque la gestión de hilos la realiza la librería de hilos en el espacio de usuario. Esto significa que invocar una función en la librería, es una llamada local en el espacio del usuario, y no una *syscall*. Sin embargo, todo el proceso se bloqueará si un hilo realiza una *syscall* que lo bloquee.

Además, dado que solo un hilo puede acceder al núcleo a la vez, varios hilos no pueden ejecutarse en paralelo en sistemas multinúcleo. Razón por la cual el modelo cayó en desuso.



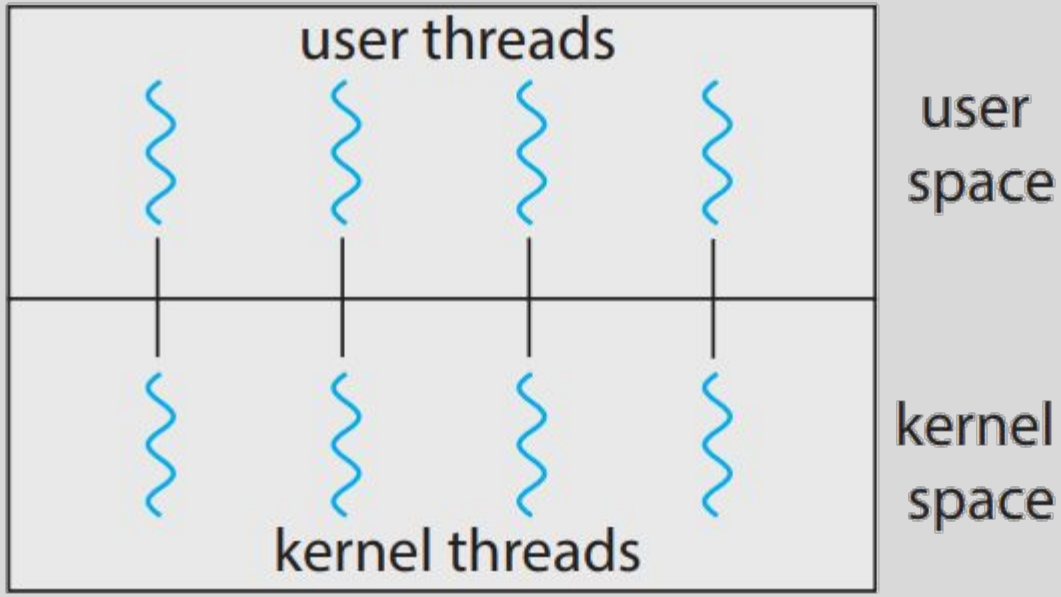
Many-to-one model.

UNO A UNO

El modelo uno a uno asigna cada hilo de usuario a un hilo del *kernel*. Tiene mayor concurrencia que el modelo muchos a uno, porque permite que otro hilo se ejecute cuando un hilo realiza una *syscall* bloqueante. Y permite que varios hilos se ejecuten en paralelo en sistemas multi núcleos.

La desventaja de este modelo es que la creación de un hilo de usuario, requiere la creación del hilo del *kernel* correspondiente, y un gran número de hilos del *kernel* puede afectar negativamente al rendimiento.

Linux, y Windows, implementan este modelo.

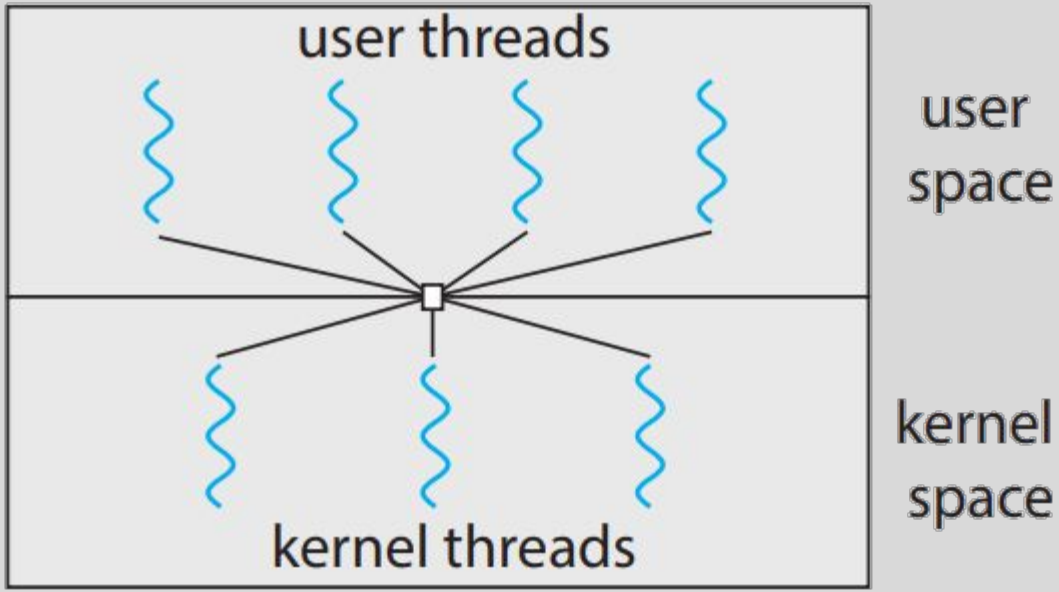


One-to-one model.

MUCHOS A MUCHOS

El modelo muchos a muchos multiplexa muchos hilos de usuario a un número menor o igual de hilos del kernel.

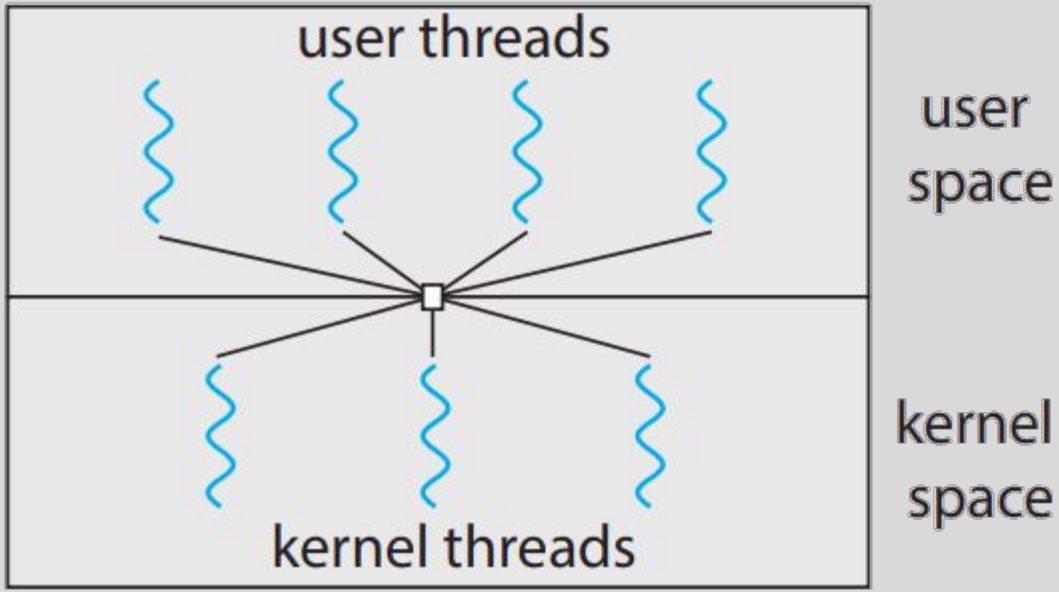
En telecomunicaciones, la multiplexación es la técnica de combinar dos o más señales, y transmitirlas por un solo medio de transmisión.



Many-to-many model.

MUCHOS A MUCHOS

Repasemos. El modelo muchos a uno le permite al desarrollador crear todos los hilos que quiera, pero no genera paralelismo, ya que el *kernel* solo puede programar un hilo a la vez. El modelo uno a uno permite una mayor concurrencia, pero el desarrollador debe tener cuidado de no crear demasiados hilos para no saturar al sistema.

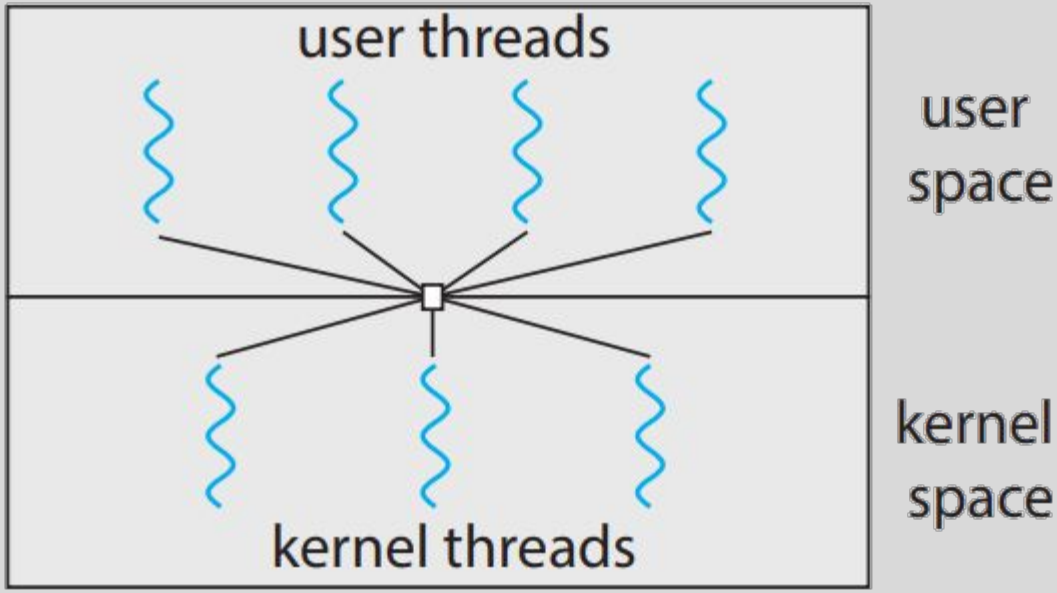


Many-to-many model.

MUCHOS A MUCHOS

El modelo muchos a muchos no presenta ninguna de estas desventajas, los desarrolladores pueden crear tantos hilos como quieran, y los hilos del *kernel* correspondientes pueden ejecutarse en paralelo. Además, cuando un hilo realiza una llamada al sistema de bloqueo, el núcleo puede programar otro hilo para su ejecución.

Una variación del modelo muchos a muchos también también permite enlazar uno a uno los hilo de usuario a un hilo del *kernel*. Modelo de 2 niveles.



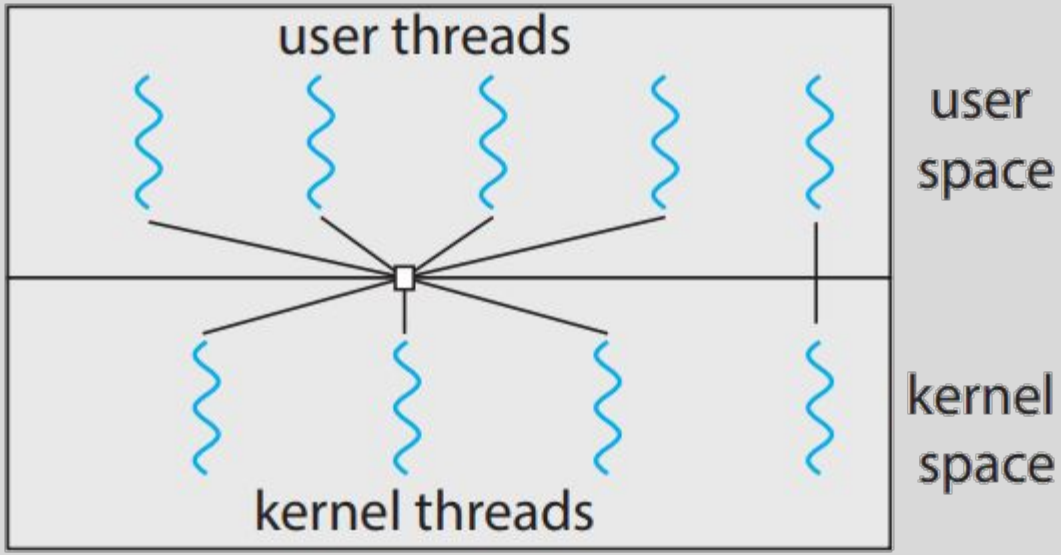
Many-to-many model.

MUCHOS A MUCHOS

Aunque el modelo muchos a muchos parece ser el más flexible, en la práctica es difícil de implementar. Además, con el aumento de núcleos de procesamiento en la mayoría de los sistemas, limitar el número de hilos del núcleo ha perdido importancia.

Hoy la mayoría de los sistemas operativos (Windows y Linux incluidos) utilizan el modelo uno a uno.

De todas formas, algunas librerías actuales de concurrencia permiten que las tareas se asignen a hilos mediante el modelo muchos a muchos.



Two-level model.

LIBRERÍAS DE HILOS

LIBRERÍAS DE HILOS

Proporcionan al programador una API para crear y gestionar hilos. (Las API son mecanismos que permiten que dos componentes de *software* se comuniquen entre sí mediante un conjunto de definiciones y protocolos).

Las librerías se pueden implementar: completamente en el espacio de usuario, sin soporte del *kernel*, donde todo el código y las estructuras de datos existen en el espacio del usuario. Esto significa que invocar una función en la librería implica una llamada a función local, y no en una llamada al sistema.

O implementar la librería a nivel de *kernel*, soportada directamente por el sistema operativo, donde el código y las estructuras de datos existen en el espacio del *kernel*. Invocar una función en la API implica una llamada al sistema.

Principalmente se utilizan tres librerías de hilos: POSIX Pthreads, Windows y Java.

LIBRERÍAS DE HILOS

Pthreads, la extensión de hilos del estándar POSIX, puede utilizarse como una librería a nivel de usuario o a nivel de *kernel*. La biblioteca de hilos de Windows es una librería a nivel de *kernel* disponible en sistemas Windows. La API de hilos de Java permite crear y gestionar hilos en programas Java. Pero como la JVM se ejecuta sobre un sistema operativo *host*, la API se implementa generalmente mediante la librería disponible en el *host*.

También existen dos estrategias para crear múltiples hilos:

- **Asincrónicos**: el hilo principal reanuda su ejecución luego de crear un hilo hijo y ambos se ejecutan de forma concurrente e independiente. Ejemplo del servidor web.
- **Síncronicos**: el hilo principal crea uno o más subprocesos secundarios y espera a que todos sus hilos hijos terminen antes de reanudarse. Cuando un hilo termina, se une a su padre, y solo después de que todos los hijos se hayan unido, el padre puede reanudar la ejecución. Por ejemplo, el hilo padre puede combinar los resultados calculados por sus diversos hijos.

PTHREADS

src > 03-threads > unix > pthread_example_sum_of_n.c > ...

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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 */
  /* set the default attributes of the thread */
  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 execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
  {
    sum += i;
    // usleep(1000); // Sleep for 1 millisecond
  }
  pthread_exit(0);
}
// Compile with: gcc -o pthread_example_sum_of_n pthread_example_sum_of_n.c -lpthread
// Run with: ./pthread_example_sum_of_n 1000000
```

WINDOWS THREADS

src > 03-threads > windows > windows_sum_of_n.c > ...

```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]);
  /* 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);
}
```

JAVA

src > 03-threads > java > Driver.java

```java
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 sum;
  }
}

public class Driver
{
  public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);
    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));
    try {
      System.out.println("sum = " + result.get());
    }
    catch (InterruptedException | ExecutionException ie) {
      System.out.println("Exception: " + ie.getMessage());
    }
    finally {
      pool.shutdown();
    }
  }
}
// The above code is a simple Java program that uses the Executor framework
```

PROBLEMAS DE HILOS

LAS *SYSCALLS FORK()* Y *EXEC()*

La semántica de las llamadas al sistema *fork()* y *exec()* cambia en un programa multihilo.

Si un hilo de un programa llama a *fork()*, ¿el nuevo proceso duplica todos los hilos del padre? Algunos sistemas UNIX eligen tener dos versiones de *fork()*: una que duplica todos los hilos y otra que duplica solo el hilo que invocó la llamada al sistema *fork()*.

La elección de la versión de *fork()* que se utilice depende de la aplicación.

Y si un hilo invoca la llamada al sistema *exec()*, el programa especificado por parámetro de *exec()* reemplazará todo el proceso, incluidos todos los hilos.

Si se llama a *exec()* inmediatamente después del *fork()*, no es necesario duplicar todos los hilos, ya que el programa especificado reemplazará el proceso. En este caso, es adecuado duplicar solo el hilo que realiza la llamada.

Sin embargo, si el proceso hijo no llama a *exec()* después del *fork()*, debería duplicar todos los hilos.

MANEJO DE SEÑALES

En los sistemas UNIX, una señal se utiliza para notificar a un proceso la ocurrencia de un evento específico. Una señal puede recibirse de forma sincrónica o asincrónica, según el origen y el motivo del evento. Todas las señales, ya sean sincrónicas o asincrónicas, siguen el mismo patrón:

1. Una señal se genera al ocurrir un evento específico.
2. La señal se entrega a un proceso.
3. Una vez entregada, la señal debe ser procesada.

Ejemplo de señales sincrónicas: 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 entregan al mismo proceso que realizó la operación que la generó (por eso se consideran sincrónicas).

Cuando una señal es generada por un evento externo a un proceso, la recibe de forma asincrónicas. Ejemplos: la finalización de un proceso con pulsaciones de teclas específicas (como <ctrl><C>) y la expiración de un temporizador. Normalmente, una señal asincrónicas se envía a otro proceso.

MANEJO DE SEÑALES

Una señal puede ser gestionada por:

1. Un controlador de señales predeterminado
2. Un controlador de señales definido por el usuario

Gestionar señales en programas monohilo es sencillo: las señales siempre se entregan a un proceso. Pero, en programas multihilo, donde un proceso puede tener varios hilos de ejecución. ¿Dónde se debe entregar una señal? En general, existen las siguientes opciones:

1. Entregar la señal al hilo al que se aplica.
2. Entregar la señal a todos los hilos del proceso.
3. Entregar la señal a determinados hilos del proceso.
4. Asignar un hilo específico para recibir todas las señales del proceso.

CANCELACIÓN DE HILOS

Cancelar un hilo implica terminarlo antes de que complete su tarea. Por ejemplo, cuando se hace click en un botón del navegador para detener la carga de la página. Generalmente, una página web se carga utilizando varios hilos; por ej: cada imagen se carga en un hilo independiente. Cuando se detiene la carga del navegador, se cancelan todos los hilos que cargan la página.

La cancelación de un hilo puede ocurrir en dos escenarios diferentes:

1. Cancelación asincrónica: Un hilo termina inmediatamente el hilo *target*.
2. Cancelación diferida. El hilo *target* verifica periódicamente si debe terminar, lo que le permite terminarse a sí mismo de forma controlada.

CANCELACIÓN DE HILOS

La dificultad surge cuando se han asignado recursos a un hilo cancelado o cuando un hilo se cancela mientras actualiza los datos que comparte con otros hilos.

Esto se vuelve especialmente problemático con la cancelación asincrónica. En general, el sistema operativo recupera recursos de un hilo cancelado, pero no todos los recursos. Esto puede generar que un recurso necesario para todo el sistema, no sea liberado.

En cambio, con la cancelación diferida, un hilo indica que se debe cancelar un hilo *target*, pero la cancelación sólo ocurre después de que este verifique si debe cancelarse. El hilo puede realizar esta verificación en un punto en el que pueda cancelarse de forma segura.

CANCELACIÓN DE HILOS

El tipo de cancelación predeterminada es la cancelación diferida. La mayoría de las llamadas al sistema bloqueantes en las librerías POSIX y C estándar, definen como puntos de cancelación, y estos se listan al invocar el comando `man pthreads` en un sistema Linux. Por ejemplo, la llamada al sistema *read()* es un punto de cancelación que permite cancelar un hilo bloqueado mientras espera la entrada de *read()*.

Debido a los problemas descritos, la cancelación asincrónica no se recomienda en la documentación de Pthreads.

ALMACENAMIENTO LOCAL DE HILOS

Los hilos que pertenecen a un proceso comparten los datos del proceso. Esto justamente proporciona una de las ventajas de la programación multihilo. Pero, en algunas circunstancias, cada hilo podría necesitar su propia copia de ciertos datos. Es decir, su almacenamiento local del hilo (o TLS). Tengan en cuenta que el desarrollador no tiene control sobre cómo se realiza la creación de hilos (por ejemplo, al utilizar una técnica implícita como un *pool* de hilos), y por esto se requiere un enfoque alternativo.

No confundir TLS con variables locales, las variables locales sólo son visibles durante una sola invocación de función, mientras que los datos TLS son visibles en todas las invocaciones de función.

ACTIVACIONES PROGRAMADAS

Un último aspecto a considerar con los programas multihilo se refiere a la comunicación entre el *kernel* y la librería de hilos. Esta coordinación permite ajustar dinámicamente el número de hilos del *kernel* para garantizar el mejor rendimiento. Notar que esto se da sólo en los modelos muchos a muchos y de 2 niveles.

Muchos sistemas colocan una estructura de datos intermedia entre los hilos de usuario y del *kernel*. Esta estructura se conoce como *lightweight process* o LWP. Para la librería de hilos del usuario, el LWP es un procesador virtual, donde la aplicación puede programar la ejecución de un hilo. Cada LWP está asociado a un hilo del *kernel*, y son los hilos del *kernel* los que el sistema operativo programa para que se ejecuten en los procesadores físicos. Si un hilo del *kernel* se bloquea (por ejemplo, espera a que se complete una operación de E/S), el LWP también se bloquea. Y así, el hilo de usuario asociado al LWP también se bloquea.

Por esto, una aplicación puede requerir cualquier número de LWP para ejecutarse eficientemente.

ACTIVACIONES PROGRAMADAS

Un esquema de comunicación entre la librería de hilos de usuario y el *kernel* se conoce como **activación programada**.

La idea es: el *kernel* proporciona a la aplicación un conjunto de procesadores virtuales (LWP), y la aplicación puede programar hilos de usuario en un procesador virtual disponible. Y luego el *kernel* debe informar a la aplicación la ocurrencia de ciertos eventos. Este procedimiento se conoce como *upcall* (llamada ascendente).

Por ejemplo, una *upcall* ocurre cuando un hilo está a punto de bloquearse. El *kernel* realiza una *upcall* a la aplicación para informarle de que un hilo está a punto de bloquearse, y luego asigna un nuevo LWP a la aplicación. La aplicación ejecuta un controlador de *upcall* en este nuevo LWP, que guarda el estado del hilo bloqueado y libera el LWP del hilo bloqueado. Cuando se produce el evento que esperaba el hilo bloqueado, el *kernel* realiza otra *upcall* a la librería de hilos para informarle que el hilo puede ejecutarse. Tras marcar el hilo desbloqueado como apto para ejecutarse, la aplicación programa un hilo apto para ejecutarse en un procesador virtual disponible.

IMPLICIT THREADING

IMPLICIT THREADING

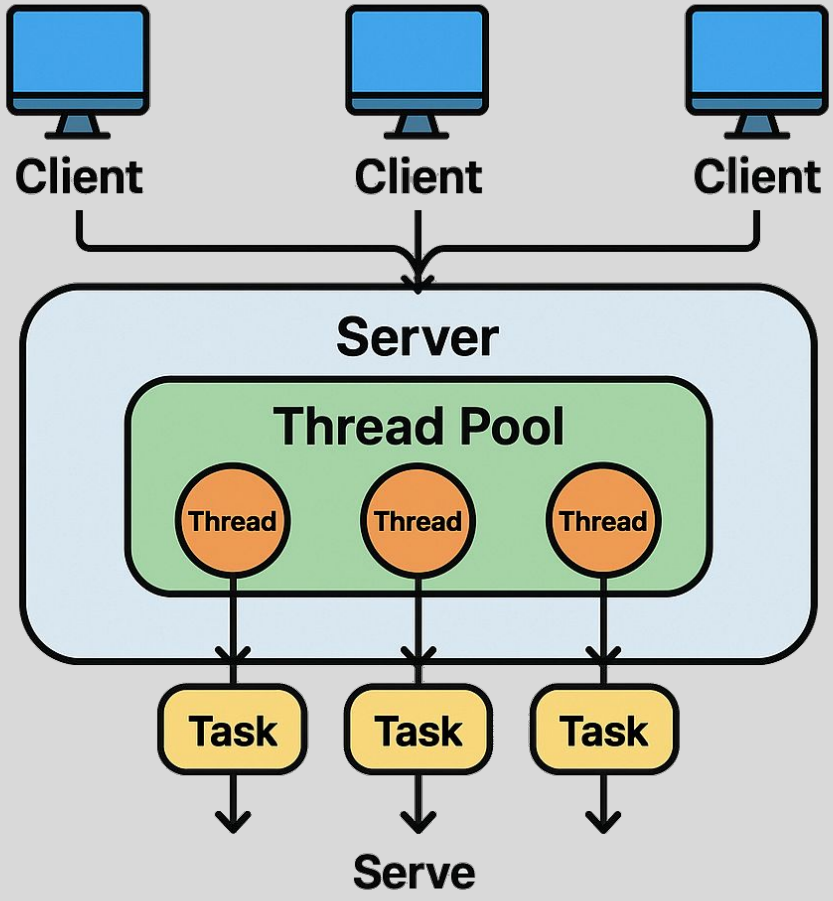
Recuerden los Desafíos de Programación mencionados anteriormente. Existe una forma de encarar estas dificultades y mejorar el diseño de aplicaciones concurrentes y paralelas. La idea es transferir la creación y gestión de hilos de los desarrolladores de aplicaciones a los compiladores y librerías de ejecución.

La ventaja es que los desarrolladores solo necesitan identificar las tareas paralelas, y las librerías determinan los detalles específicos de la creación y gestión de subprocesos. Esta estrategia se denomina **hilos implícitos**.

Generalmente requiere que los desarrolladores de aplicaciones identifiquen tareas (no subprocesos) que puedan ejecutarse en paralelo. Veremos cuatro enfoques diferentes para el diseño de aplicaciones que pueden aprovechar los procesadores multinúcleo mediante hilos implícitos.

THREAD POOLS

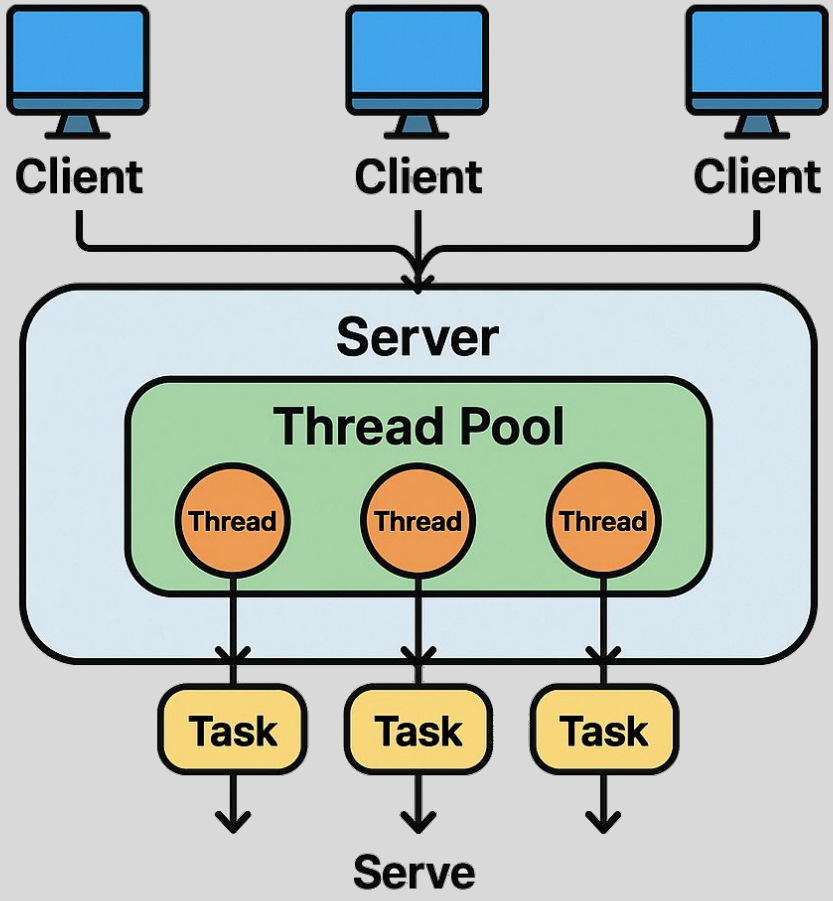
La idea general de un *pool de threads*, es crear varios hilos al inicio y colocarlos en un *pool*, donde permanecen a la espera de trabajo. Ejemplo: cuando un servidor recibe un *request*, en lugar de crear un hilo, envía el *request* al *pool* de hilos y reanuda la espera de solicitudes. Si hay un hilo disponible en el *pool*, se despierta y la solicitud se atiende inmediatamente. Sino, la tarea se pone en cola de espera hasta que se libera uno. Una vez que un hilo completa su servicio, regresa al *pool* y espera más trabajo. Los *pools* de hilos funcionan bien cuando las tareas enviadas al *pool* se pueden ejecutar de forma asincrónica.



THREAD POOLS

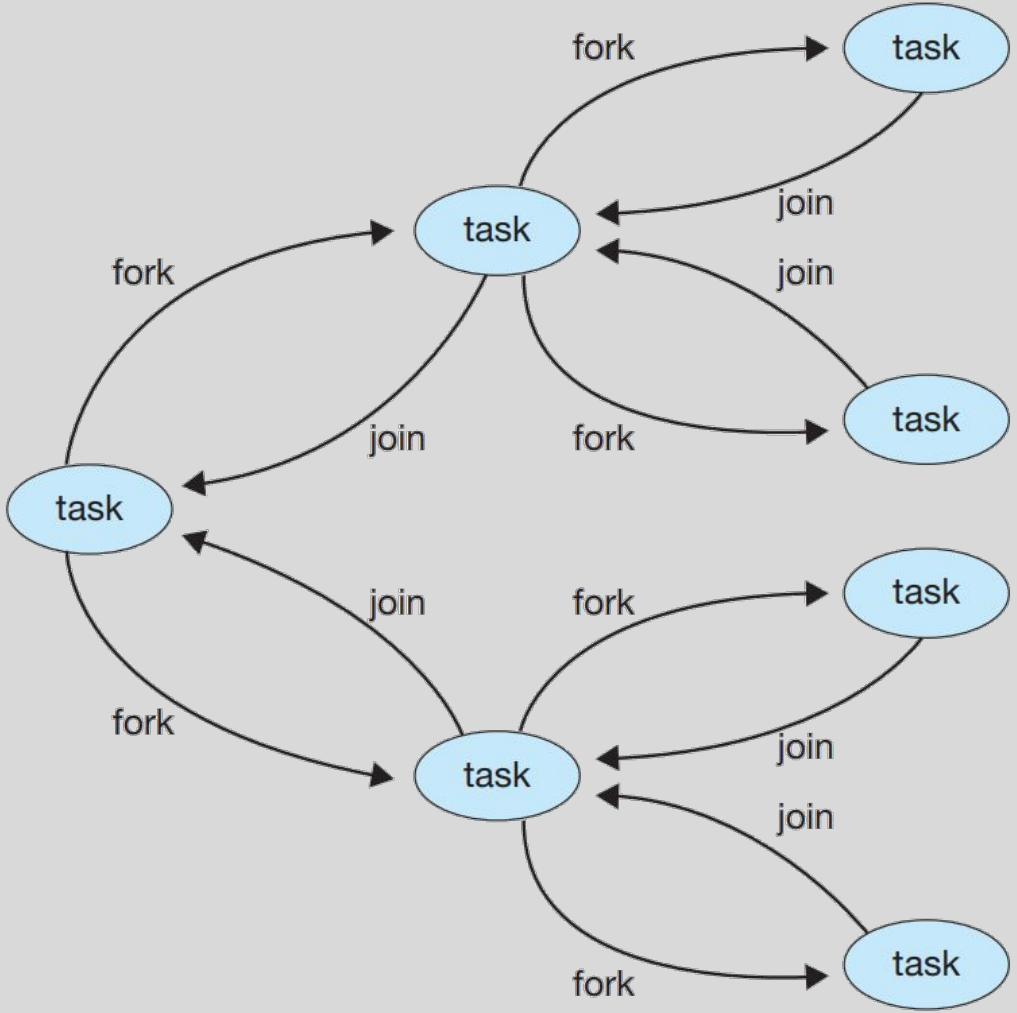
Ventajas:

1. Atender un *request* con un hilo existente suele ser más rápido que esperar la creación de uno.
2. Un *pool* de hilos limita el número de hilos existentes en un momento dado. Esto es importante en sistemas que no admiten un gran número de subprocesos simultáneos.
3. Separar la tarea a ejecutar de su mecánica de creación, permite utilizar diferentes estrategias para ejecutarla. Por ejemplo, se podría programar su ejecución para más adelante o de forma periódica.



FORK JOIN

El hilo principal crea (*fork*) uno o más hilos secundarios y espera a que estos terminen y se unan a él, momento en el que puede recuperar y combinar sus resultados. Este modelo sincrónico suele caracterizarse por la creación explícita de hilos, pero también es un excelente candidato para la creación implícita de hilos, porque los hilos no se construyen directamente durante la etapa de fork, sino que se designan tareas paralelas. Una librería gestiona el número de hilos creados y también se encarga de asignarles tareas. En cierto modo, este modelo de *fork-join* es una versión sincrónica de *pool* de hilos.



OPENMP

OpenMP es un conjunto de directivas de compilación y una API para programas escritos en C, C++ o FORTRAN. Proporciona soporte para programación paralela en entornos de memoria compartida. OpenMP identifica las regiones paralelas como bloques de código que pueden ejecutarse en paralelo. Los desarrolladores de aplicaciones insertan directivas de compilación en su código en las regiones paralelas, y estas directivas indican a la librería de ejecución de OpenMP que ejecute la región en paralelo.

src > 03-threads > unix > open_MP_example.c > ...

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

int main(int argc, char *argv[])
{
  /* sequential code */
  printf("I am a sequential region. INIT. \n");
  /* parallel region */
  #pragma omp parallel
  {
    printf("I am a parallel region. \n");
  }
  /* sequential code */
  printf("I am a sequential region. DONE. \n");
  return 0;
}

/*
  Compilation:
  gcc -fopenmp -o open_MP_example open_MP_example.c
  Execution:
  ./open_MP_example
  Output:
  I am a sequential region. INIT.
  I am a parallel region.
  I am a parallel region.
  I am a parallel region.
  I am a parallel region.
  I am a sequential region. DONE.
*/
```

OPENMP

Cuando OpenMP encuentra la directiva:

`#pragma omp parallel`

crea tantos hilos como núcleos de procesamiento haya en el sistema. Todos los hilos ejecutan simultáneamente la región paralela. Cada hilo finaliza al salir de la región paralela.

OpenMP también proporciona directivas para ejecutar regiones de código en paralelo, incluidos bucles:

`#pragma omp parallel for`

src > 03-threads > unix > open_MP_example.c > ...

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

int main(int argc, char *argv[])
{
  /* sequential code */
  printf("I am a sequential region. INIT. \n");
  /* parallel region */
  #pragma omp parallel
  {
    printf("I am a parallel region. \n");
  }
  /* sequential code */
  printf("I am a sequential region. DONE. \n");
  return 0;
}

/*
  Compilation:
  gcc -fopenmp -o open_MP_example open_MP_example.c
  Execution:
  ./open_MP_example
  Output:
  I am a sequential region. INIT.
  I am a parallel region.
  I am a parallel region.
  I am a parallel region.
  I am a parallel region.
  I am a sequential region. DONE.
*/
```

GRAND CENTRAL DISPATCH (GCD)

GCD es una tecnología desarrollada por Apple para sus sistemas operativos macOS e iOS. Combina una librería en tiempo de ejecución y una API que permiten a los desarrolladores identificar secciones de código (tareas) para ejecutarlas en paralelo.

GCD programa tareas para su ejecución en tiempo de ejecución colocándolas en una cola. Al quitar una tarea de una cola, la asigna a un hilo disponible de un *pool* de hilos que gestiona. GCD identifica dos tipos de colas de despacho: seriales y concurrentes.

- Las tareas colocadas en una cola serial se eliminan en orden FIFO. Una vez eliminada una tarea de la cola, debe completar su ejecución antes de que se elimine otra. Cada proceso tiene su propia cola serial, y los desarrolladores pueden crear colas seriales adicionales locales para un proceso en particular. Las colas seriales son útiles para garantizar la ejecución secuencial de varias tareas.
- Las tareas colocadas en una cola concurrente también se eliminan en orden FIFO, pero se pueden eliminar varias tareas a la vez, lo que permite que varias tareas se ejecuten en paralelo.

Muchas Gracias

Jeremías Fassi

Javier E. Kinter