4. Concurrency and Threads (concurrencia e hilos)

El uso de la palabra concurrencia se refiere a la realización de múltiples actividades al mismo tiempo. En la actualidad, por ejemplo, los servidores pueden tener docenas de procesadores y discos, otro ejemplo puede ser el de tener muchos dispositivos conectados al mismo tiempo.

El manejo correcto de la concurrencia es un desafío muy importante para el desarrollo de un sistema operativo, tanto para manejar el hardware como para ejecutar múltiples aplicaciones simultáneamente. Al mismo tiempo el SO tiene que mantener un trakeo de las aplicaciones que están corriendo en el momento.

La concurrencia es un tema principal en el desarrollo de las aplicaciones. Por ejemplo en redes es necesario que la aplicación pueda manejar múltiples peticiones al mismo tiempo. Otras aplicaciones necesitan dividir el trabajo para realizarlo concurrentemente en múltiples procesadores y así aprovechar su desempeño.

La idea clave atrás de la concurrencia es dividir el programa en múltiples hilos de ejecución llamados threads. Los threads nos permiten definir un conjunto de tareas a realizar, las cuales son secuenciales. Cada hilo de ejecución tiene su propio procesador. Muchas veces al usar threads se debe agregar código para poder compartir recursos y estructuras de datos entre los hilos.

La abstracción de los hilos permite que los programadores desarrollen aplicaciones sin preocuparse por la cantidad de procesadores disponibles, obviamente la cantidad es limitada, pero es tarea del sistema operativo proveer la ilusión de que se poseen infinitos procesadores.

4.1 Casos de uso de Threads

La intuición detrás de la abstracción de los hilos de ejecución es simple. Podemos representar cada tarea como un thread en sí. De hecho podemos pensar un programa común como un simple-thread (ya que usa uno solo).

Un multi-thread es una generalización del mismo concepto básico del modelo de programación, cada tarea sigue una secuencia de pasos que ejecuta estados o itera por bucles, etc. Sin embargo un programa puede ejecutar varios hilos a la vez.

4.1.1 Cuatro razones para usar Threads

El uso de threads para hacer manejo de la concurrencia tiene importantes ventajas.

- **Estructura del programa: expresar lógicamente tareas concurrentes.** Los programas interactúan con aplicaciones del mundo real que tiene actividades concurrentes, por ejemplo pueden escribir en diferentes tareas por threads separados. O pueden leer la posición del mouse e ir dibujando la pantalla al mismo tiempo
- **Respuesta al usuario: intercambiar trabajos para que ejecuten en background (segundo plano).** Para **mejorar** la respuesta al usuario y el desempeño, un patrón de diseño común es el de crear threads para que se realicen trabajos en segundo plano sin que el usuario espere los resultados. De esta manera la interfaz puede seguir respondiendo a los comandos ingresados.
- **Desempeño: explotar el uso de múltiples procesadores.** Los programas usan threads para poder realizar tareas en paralelo. De esta manera se puede realizar un conjunto de tareas en menor tiempo.
- **Desempeño: manejando los dispositivos de entrada y salida.** El sistema operativo interactuar con el mundo **exterior** por medio de los dispositivos de I/O. Ejecutando tareas en diferentes threads, cuando una está esperando por I/O, el procesador puede avanzar en otra tarea diferente.

El beneficio de la concurrencia entre el procesador y la I/O es doble: en primer lugar, los procesadores son mucho más rápido que los sistemas de I/O con los que interactúan, por lo que mantener la inactividad del procesador durante la I/O perdería gran parte de su capacidad. Por ejemplo, la latencia para leer desde el disco puede ser decenas de milisegundos, lo suficiente para ejecutar más de 10 millones de instrucciones en un procesador moderno. Después de solicitar un bloque de disco, el sistema operativo puede cambiar a otro programa, u otro hilo dentro del mismo programa, hasta que el disco haya finalizado y el hilo original está listo para reanudar.

En segundo lugar, I/O proporciona una forma para que la computadora pueda interactuar con entidades externas, como los usuarios pulsando una tecla en un teclado o el envío de un paquete desde un equipo remoto. La

llegada de este tipo de evento de I/O es impredecible, por lo que el procesador debe ser capaz de trabajar en otras tareas mientras y aun así poder responder rápidamente a estos eventos externos.

Threads vs procesos

Se describió un proceso como la ejecución de un programa con derechos restringidos. Un hilo es una secuencia de instrucciones que se ejecutan independientemente dentro de un programa. Tal vez la mejor manera de ver cómo estos conceptos están relacionados, es ver cómo los diferentes sistemas operativos los combinan de diferentes maneras:

1. **Un hilo por proceso:** Una aplicación de simple-thread tiene una sola secuencia de instrucciones, ejecutando de principio a fin. El núcleo del sistema operativo ejecuta aquellas instrucciones en modo usuario para restringir el acceso a las operaciones privilegiadas o la memoria del sistema. Los procesos realizan llamadas al sistema para pedir al kernel que autorice para realizar operaciones privilegiadas en su nombre.
2. **Muchos hilos por procesos:** un programa puede estar estructurado como varios procesos simultáneos, cada uno ejecutando dentro de los derechos restringidos del proceso. En un momento dado, un subconjunto de los hilos de proceso puede estar en ejecución, mientras que el resto se suspenden. Cualquier hilo que se ejecuta en un proceso puede hacer llamadas al sistema en el núcleo, bloqueando ese hilo hasta que se devuelve la llamada, pero permitiendo que otros hilos puedan continuar funcionando. Del mismo modo, cuando el procesador recibe una interrupción de I/O, se adelanta a uno de los subprocesos que se ejecutan, por lo que el núcleo puede ejecutar el manejador de interrupciones y cuando el manipulador termina, se reanuda el núcleo de ese hilo.
3. **Muchos procesos de hilo simple:** Tan recientemente como hace veinte años, muchos sistemas operativos soportaban varios procesos, pero sólo un hilo por proceso. Para el kernel, sin embargo, cada proceso se ve como un hilo: una secuencia de instrucciones que a veces está ejecutando en modo kernel y otras a nivel de usuario. Por ejemplo, en un multiprocesador, si múltiples procesos realizan llamadas al sistema al mismo tiempo, el kernel también tiene múltiples hilos de ejecución al mismo tiempo (concurrentemente) y en modo kernel.
4. **Muchoas hilos de kernel:** para manejar la complejidad, pasar tareas a segundo plano aprovecha el paralelismo y esconde las latencias de I/O. el sistema operativo por sí mismo puede beneficiarse del uso de múltiples hilos. En este caso cada hilo del kernel ejecuta con los privilegios del kernel, y puede realizar instrucciones privilegiadas, acceder a la memoria del sistema y ejecutar directamente comandos en los dispositivos de I/O. El sistema operativo del kernel implementa para sí mismo la abstracción de hilos.

Por la gran utilidad de los threads, la mayoría de los SO modernos permiten múltiples hilos por procesos y múltiples hilos de kernel

4.2 Abstracción de hilos

Un hilo es una secuencia simple (única) de ejecución que representa una tarea planificada separadamente

- **Secuencia de ejecución única.** Cada hilo ejecuta una secuencia de instrucciones (asignaciones, condicionales, bucles, procedimientos, etc.) al igual que el modelo de programación secuencial.
- **Tareas planificadas separadamente** el SO puede ejecutar suspender o reanudar un hilo en cualquier momento

4.2.1 Ejecución, suspensión, y reanudación de un hilo

Los hilos proporcionan la ilusión de un número infinito de procesadores. El sistema operativo debe ejecutar las instrucciones de cada hilo para que cada hilo haga progreso, pero el hardware real tiene un número limitado de procesadores, y tal vez sólo uno. Para manejar un conjunto de hilos dentro de pocos procesadores, el sistema operativo tiene un hilo planificador (threads scheduler) que puede cambiar entre los hilos que se están ejecutando y los que están listos pero no en funcionamiento.

El cambio entre hilos es transparente para el código que está siendo ejecutado dentro de cada hilo. La abstracción hace que cada hilo parezca ser un único flujo de ejecución; esto significa que el programador puede prestar atención a la secuencia de instrucciones dentro de un hilo y no si o cuando esa secuencia puede ser (temporalmente) suspendida para permitir que otro hilo se ejecute.

Los hilos proveen un modelo de ejecución en donde cada hilo ejecuta sobre un procesador virtual a una velocidad variable e impredecible. Desde el punto de vista del código del hilo, cada instrucción aparece para ser ejecutada inmediatamente después de la anterior. Sin embargo, el planificador puede suspender un hilo entre una

instrucción y la siguiente y volver a correrlo más tarde. Es como si el hilo se ejecuta en un procesador que a veces se vuelve muy lento.

4.2.2 ¿Por qué velocidad impredecible?

Puede parecer extraño pedir que los programadores asuman que el procesador virtual de un hilo ejecuta a velocidad impredecible y es posible que se intercale con la ejecución de otros hilos. En los programas multihilo deben hacerse suposiciones sobre el comportamiento del programador de hilos. A su vez, las decisiones de programador del kernel (cuando se asignan a un hilo a un procesador, y al adelantarse por un subproceso diferente) se pueden hacer sin tener que preocuparse si podrían afectar el correcto funcionamiento del programa.

Si los hilos son independientes, no comparten memoria ni otros recursos, entonces el orden de ejecución no importa. Cualquier coordinador producirá el mismo resultado. Sin embargo, la mayoría de los programas multihilo comparten estructuras de datos. El programador debe utilizar la sincronización explícita para garantizar el correcto funcionamiento del programa, independientemente de la posible intercalación de instrucciones entre diferentes hilos.

Incluso si podríamos ignorar el problema de la programación de hilos (por ejemplo si tenemos más porcesadores que hilos, donde asignamos un hilo a cada procesador físico) la realidad física es que la velocidad de ejecución relativa de los diferentes hilos puede verse afectada significativamente por factores fuera de control. Un ejemplo extremo es que el programador puede estar depurando un hilo paso a paso, mientras que otros hilos corren a toda velocidad en otros procesadores. Si el programador ha de tener alguna esperanza de entender el comportamiento concurrente programa, la corrección del programa no puede depender del hilo que se está observando.

La variabilidad en la velocidad de ejecución también se produce durante el funcionamiento normal. El acceso a memoria puede estancar un procesador para cientos o miles de ciclos si se produce un fallo de caché. Otros factores incluyen la frecuencia con la que el programador tiene prioridad sobre el hilo, el número de procesadores físicos que están presentes en una máquina, el tamaño de los cachés, qué tan rápida es la memoria, cómo el firmware de ahorro de energía ajusta la velocidad de reloj del procesador, los mensajes de que llegan por la red, o lo que se reciben de entrada del usuario. Las velocidades de ejecución de los diferentes hilos de un programa son difíciles de predecir, puede variar en un hardware diferente, e incluso puede variar de una ejecución a otra en el mismo hardware. Como resultado, hay que coordinar las acciones de hilo a través de la sincronización explícita en lugar de tratar de razonar acerca de su velocidad relativa.

Un coordinador de interrupciones no es un hilo. El manejador de interrupción del núcleo comparte cierto parecido con un hilo: se trata de una única secuencia de instrucciones que ejecuta de principio a fin. Sin embargo, un manejador de interrupciones no es planificable independiente: se desencadena por un evento I / O de hardware, en lugar de una decisión por el planificador de hilos en el núcleo. Una vez iniciado, el manejador de interrupciones ejecuta hasta el final, a no ser que sea interrumpido por in interrupción de mayor prioridad.

4.3 Simple Thread API

Simple Threads API	
Void thread_create (thread, func, arg)	Crear un nuevo hilo, almacenando información sobre él en hilo. Simultáneamente con la llamada al hilo, se ejecuta la función func con el argumento arg.
Void thread_yield ()	La llamada al hilo deja voluntariamente que el procesador se encargue de ejecutar otro hilo/s. El coordinador puede restablecer la llamada cuando quiera
int thread_join (thread)	Espera a que el hilo termine si aún no lo hizo, luego devuelve el valor pasado a pthread_exit por ese hilo. Tenga en cuenta que pthread_join puede ser llamado una sola vez para cada hilo.
void thread_exit (ret)	Termina el hilo actual. Almacena el valor RET en la estructura de datos del hilo actual. Si otro hilo ya está a la espera de una llamada a pthread_join, lo reanuda.

Esta API simplificada se basa en el estándar API POSIX pthreads, pero omite algunas opciones POSIX y manejo de errores por simplicidad. La mayoría de los otros paquetes de hilos son bastante similares; si usted entiende cómo programar con esta API, le resultará fácil escribir código con la mayoría de las API de hilos estándar.

Una buena manera de entender la API de hilos simples es que proporciona una manera de invocar una llamada a procedimiento asincrónico. Una llamada de procedimiento normal pasa a un conjunto de parámetros a la función, la función se ejecuta inmediatamente en la pila de la función que llama, y cuando se termina la función, el control vuelve de nuevo a la función que llamó con el resultado. Una llamada a procedimiento asincrónico separa la llamada de la devolución: con thread_create, la función que llama inicia la función, pero a diferencia de una llamada a procedimiento normal, la función que llama continúa la ejecución concurrentemente con la función llamada. Más tarde, la función invocadora puede esperar a la finalización de función invocada (con thread_join). thread_create es análogo al proceso de UNIX fork y exec, mientras thread_join es análogo al proceso de UNIX wait. UNIX fork crea un nuevo proceso que se ejecuta simultáneamente con el proceso llamando fork; UNIX exec hace que el proceso ejecute un programa específico. UNIX wait permite que el proceso de llamada pueda suspender la ejecución hasta la finalización del nuevo proceso.

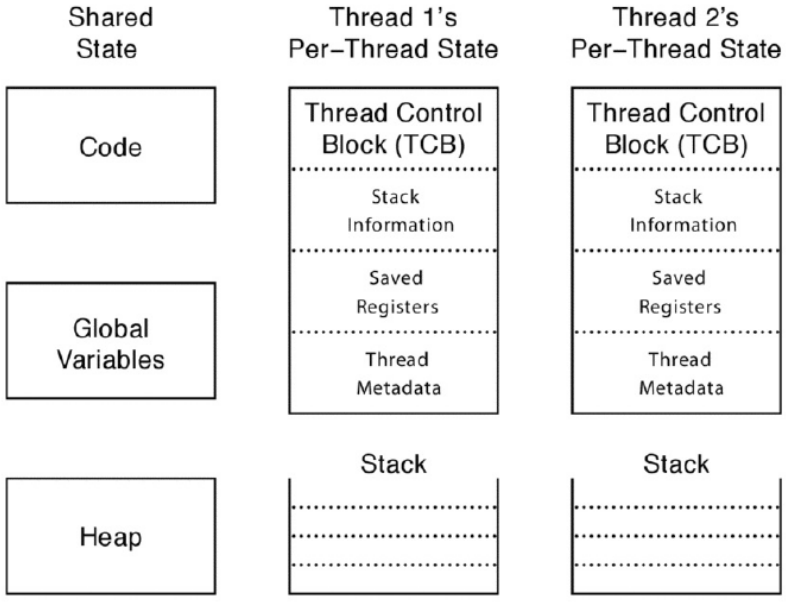
4.4 Estructura de datos y ciclo de vida de un hilo

Como hemos visto, cada hilo representa un flujo de ejecución secuencia. El sistema operativo proporciona la ilusión de que cada hilo se ejecuta en su propio procesador virtual mediante la suspensión y reanudación de los hilos de forma transparente. Para que la ilusión funcione, el sistema operativo debe guardar y restaurar el estado de un hilo. Sin embargo, debido a los hilos funcionan tanto en un proceso o en el núcleo, también hay un estado compartido que no se guarda o ni se restaura cuando se cambia el procesador entre los hilos.

Por lo tanto, para entender cómo el sistema operativo implementa la abstracción hilo, hay que definir tanto el estado de cada hilo como el estado que se comparte entre los hilos. Entonces podemos describir el ciclo de vida de un hilo.

Figura 4.4

Un proceso multi hilo o el kernel de un sistema operativo tiene un estado por hilo y un estado compartido entre los hilos. El bloque de control de hilos almacena el estado para cada uno (registros guardados del procesador y un puntero a la pila) y metadatos necesarios para gestionar el hilo (ID, prioridad de la programación, propietario y el consumo de recursos) El estado compartido incluye el código del programa, las variables estáticas globales y el heap.



4.4.1 Per-Thread State and Thread Control Block (TCB)

El sistema operativo necesita una estructura de datos para representar el estado de un hilo; un hilo es como cualquier otro objeto en este sentido. Esta estructura de datos se llama el bloque de control de hilo (TCB). Para cada hilo creado el sistema operativo crea un TCB. El bloque de control del hilo lleva a cabo dos tipos de información para cada subproceso:

1. El estado de la computación que se realiza por el hilo.
2. Los metadatos sobre el hilo que se utiliza para gestionar el hilo.

Estado de Cálculo para cada hilo. Para crear múltiples hilos y ser capaz de iniciar y detener cada uno de ellos cuando sea necesario, el sistema operativo debe asignar espacio en el TCB para el estado actual de la computación de cada hilo: un puntero a la pila de hilos y una copia de sus registros del procesador.

- **PILA**. La pila de un hilo es la misma que la pila para un único hilo de cálculo, es necesario almacenar información para los procedimientos anidados del hilo que se está ejecutando actualmente. Por ejemplo, si el hilo llama foo (), foo () llama a bar (), y el bar () llama bas(), entonces la pila contendría un marco de pila para cada uno de estos tres procedimientos; cada marco de pila contiene las variables locales utilizadas por el procedimiento, los parámetros del procedimiento fueron llamados con la dirección de retorno y saltan cuando finaliza el procedimiento. (VER)
  Debido a que en un momento dado diferentes hilos pueden estar en diferentes estados en sus cómputos secuenciales, cada uno puede estar en un lugar diferente, en un procedimiento distinto llamado con diferentes argumentos de un agrupamiento diferente de los procedimientos que los encierra. Cada hilo necesita su propia pila. Cuando se crea un nuevo hilo, el sistema operativo asigna una nueva pila y almacena un puntero a la pila de hilos en el TCB.
- **Copia de los registros del procesador**. registros del procesador incluyen no sólo sus registros de propósito general para el almacenamiento de valores intermedios para los cálculos actuales, sino que también incluyen registros para fines especiales, tales como el puntero de instrucción y puntero de pila.
  Para poder suspender a un hilo, ejecutar otro hilo, y luego reanudar el hilo original, el sistema operativo **necesita** un lugar para almacenar los registros de un hilo cuando el hilo no se está ejecutando de forma activa. En algunos sistemas, los registros de propósito general, para un hilo detenido, se almacenan en la parte superior de la pila, y la TCB incluye solamente un puntero a la pila. En otros sistemas, el TCB contiene espacio para una copia de todos los registros del procesador.

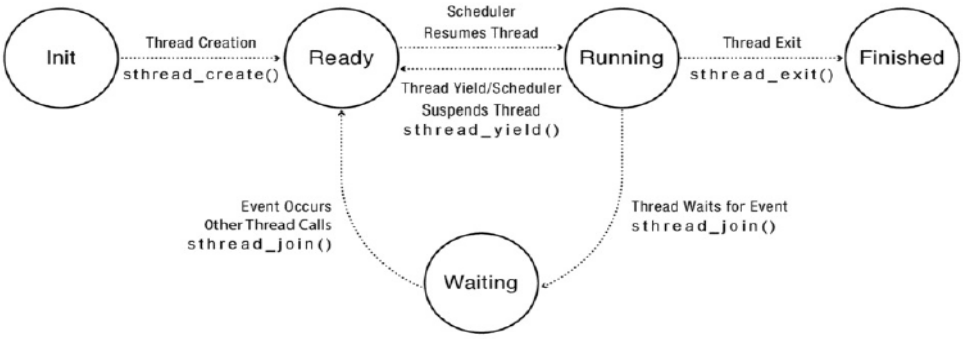
Los metadatos para cada hilo. El TCB también incluye metadatos para cada hilo, información para la gestión del hilo. Por ejemplo, cada hilo podría tener un ID, la prioridad de programación, y el estado (por ejemplo, si el hilo está esperando un evento o está listo para ser colocado en un procesador).

4.4.2 Shared State

A diferencia del estado que se asigna para cada hilo, algunos estados se comparten entre los hilos que se ejecuta en el mismo proceso o en el núcleo del sistema operativo. En particular, el código del programa es compartido por todos los hilos de un proceso, aunque cada hilo puede estar ejecutando en un lugar diferente dentro de ese código. Además, las variables globales asignadas estáticamente y las variables dinámicamente asignadas del heap pueden almacenar información que es accesible a todos los hilos.

4.5 Ciclo de vida del hilo

Es útil tener en cuenta la progresión de estados como un hilo pasa de ser creado, de ser programado y desprogramado en entradas y salidas de un procesador, y luego a la salida.



Ciclo de vida de un hilo

INIT: la creación del hilo pone un hilo en su estado INIT y asigna e inicializa las estructuras de datos por hilo. Una vez hecho esto, el código de creación del hilo pone el mismo en estado READY mediante la adición del hilo a la *ready list*. La lista de listas es el conjunto de hilos ejecutables que están esperando su turno para utilizar un procesador. En la práctica, la *ready list* no es en realidad una "lista"; el sistema operativo utiliza típicamente una estructura de datos más sofisticada para realizar un seguimiento de hilos ejecutables, tales como una cola de prioridad. Sin embargo, para seguir una convención, vamos a seguir refiriéndonos a ella como *ready list*.

READY: un hilo en estado de *ready* está disponible para ser ejecutado pero no lo está haciendo actualmente. El TCB está en la *ready list* y los valores de sus registros son almacenados en su TCB. En cualquier momento, el programador puede causar que un hilo pase de *ready* a *running* copiando sus valores de registro de su TCB a los registros de un procesador.

RUNNING: Un hilo en estado de ejecución se ejecuta en un procesador. En este momento, sus valores de registro se almacenan en el procesador en vez de en el TCB. Un hilo en ejecución puede pasar al estado *ready* de dos maneras:

- El programador (scheduler) puede adelantarse un hilo en ejecución y moverlo al estado READY por: ahorro de los registros de los hilos a su TCB o por el cambio del procesador para ejecutar el siguiente hilo de la *ready list*.
- Un hilo en ejecución puede renunciar voluntariamente al procesador y pasar de *running* a *ready*.

Observe que un hilo puede pasar de *ready* a *running* y volver muchas veces. Dado que el sistema operativo guarda y restaura los registros de los hilos, sólo la velocidad de ejecución del hilo se ve afectada por estas transiciones.

(Por convención en este libro, un hilo que se está ejecutando no está en la *ready list*. Solo los hilos en están en esa lista. En Linux se usa otra convención: el hilo en ejecución esta primero en la *ready list*. Cualquier convención es válida mientras se mantenga para todos los hilos)

WAITING: Un hilo en el estado de *waiting* está a la espera de algún acontecimiento. Mientras que el programador puede mover un hilo en el estado *ready* al estado *running*, un hilo en el estado *waiting* no se puede ejecutar hasta que por alguna acción de otro hilo se lo mueva de *waiting* a *ready*.

El programa threadHello en la figura anterior es un ejemplo de un hilo en *waiting*. Después de crear sus hilos hijos, el hilo principal debe esperar a que se completen, llamando thread_join vez por cada hijo. Si un hilo hijo específico no terminó para el tiempo de la unión, el hilo principal va desde *running* a *waiting* hasta que el hilo hijo termina.

Mientras que un hilo espera un evento, no puede avanzar; Por lo tanto, no es útil para ejecutarlo. En lugar de seguir ejecutando el hilo o guardar la TCB en la programación de la *ready list*, la TCB se almacena en la *waiting list* de alguna variable de sincronización asociado con el evento. Cuando se produce el evento es necesario que el sistema

operativo mueva el TCB de la lista de espera de la variable de sincronización a la programación de la *ready list*, pasando desde *waiting* a *ready*.

FINISHED: Un hilo en estado *finished* nunca se ejecuta de nuevo. El sistema puede liberar parte o la totalidad de su estado para otros usos, aunque puede mantener algunos remanentes del hilo terminado por un tiempo, poniendo la TCB en la *finished list*. Por ejemplo, la llamada thread_exit permite pasar un hilo su valor de salida a su padre a través de thread_join. Finalmente, cuando el estado de un hilo ya no es necesario (por ejemplo, después de que su valor de salida ha sido leído por thread_join), el sistema puede eliminar y recuperar el estado del hilo.

State of Thread	Location of Thread Control Block (TCB)	Location of Registers
INIT	Being Created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List then Deleted	TCB or Deleted

Figure 4.10: Location of thread's per-thread state for different life cycle stages.