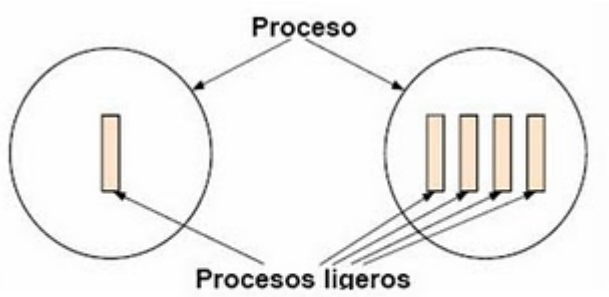

HEBRAS
JULIO FRESNEDA

DEFINICIÓN

Un proceso ligero, hebra o hilo, es un programa en ejecución (flujo de ejecución) que comparte la imagen de memoria y otras informaciones con otros procesos ligeros. Un proceso puede contener un solo flujo de ejecución, como ocurre en los procesos clásicos, o mas de un flujo de ejecución (procesos ligeros).

Desde el punto de vista de la programación, un proceso ligero se define como una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario, o proceso ligero primario, corresponde a la función main.



Cada proceso ligero tiene informaciones que le son propias y que no comparte con otros procesos ligeros. Las informaciones propias se refieren fundamentalmente al contexto de ejecución, pudiéndose destacar las siguientes:

- Contador de programa.
- Pila.
- Registros.
- Estado del proceso ligero (ejecutando, listo o bloqueado).

Todos los procesos ligeros de un mismo proceso comparten la información del mismo. En concreto, comparten:

- Espacio de memoria.
- Variables globales.
- Archivos abiertos.
- Procesos hijos.
- Temporizadores.
- Señales y semáforos.
- Contabilidad.

Es importante destacar que todas las hebras de un mismo proceso comparten el mismo espacio de direcciones de memoria, que incluye el código, los datos y las pilas de los diferentes procesos ligeros. Esto hace que no exista protección de memoria entre las hebras de un mismo proceso, algo que si ocurre con los procesos convencionales.

La utilidad de múltiples hilos paralelos de un proceso, sirven para que ese mismo proceso

pueda hacer varias cosas a la vez. Cada hebra trata una tarea separada. Este enfoque puede tener varios beneficios.

- Podemos simplificar el código que se ocupa de los eventos asíncronos asignando un hilo para manejar cada tipo de evento. Cada hilo puede entonces manejar su evento utilizando un modelo de programación síncrono. Un modelo de programación síncronoes mucho más simple que un asíncrono.
- Múltiples procesos tienen que utilizar mecanismos complejos proporcionados por el sistema operativo para compartir memoria y descriptores de archivo, cosa que se ahorra con multihilos.

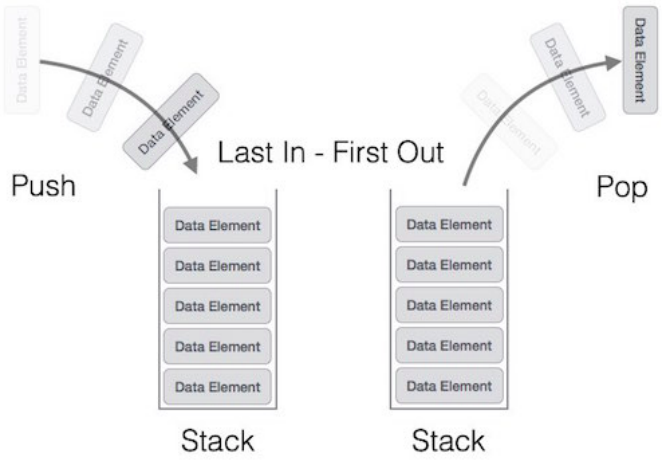
Cada hilo tiene su propio contador de programa y almacena su información en una pila. Aquí se describe qué son.

CONTADOR DE PROGRAMA

El contador de programa (en inglés Program Counter o PC) es un registro interno del computador en el que se almacena la dirección de la última instrucción leída. De esta manera el computador puede saber cuál es la siguiente instrucción que debe ejecutar. Más detalladamente, El contador de programa, también llamado Puntero de instrucciones (Instruction Pointer), parte del secuenciador de instrucciones en algunas computadoras, es un registro del procesador de un computador que indica la posición donde está el procesador en su secuencia de instrucciones. Dependiendo de los detalles de la máquina particular, contiene o la dirección de la instrucción que es ejecutada, o la dirección de la próxima instrucción a ser ejecutada. El contador de programa es incrementado automáticamente en cada ciclo de instrucción de tal manera que las instrucciones son leídas en secuencia desde la memoria. Ciertas instrucciones, tales como las bifurcaciones y las llamadas y retornos de subrutinas, interrumpen la secuencia al colocar un nuevo valor en el contador de programa.
En la inmensa mayoría de los procesadores, el puntero de instrucciones es incrementado inmediatamente después de leer (fetch) una instrucción de programa; esto significa que la dirección a la que apunta una instrucción de bifurcación es obtenida agregando el operando de la instrucción de bifurcación a la dirección de la instrucción siguiente (byte o word) dependiendo del tipo de la computadora) después de la instrucción de bifurcación. La dirección de la siguiente instrucción a ser ejecutada siempre se encuentra en el contador de instrucción.

PILA

Cada hebra consta de un espacio propio en la pila del kernel. Para entender esto, vamos a describir la definición y funcionamiento de una pila.
Una pila típica es un área de la memoria de los computadores con un origen fijo y un tamaño variable. Al principio, el tamaño de la pila es cero. Un puntero de pila, por lo general en forma de un registro de hardware, apunta a la más reciente localización en la pila; cuando la pila tiene un tamaño de cero, el puntero de pila de puntos en el origen de la pila.



Las dos operaciones aplicables a todas las pilas son:

- Una operación apilar, en el que un elemento de datos se coloca en el lugar apuntado por el puntero de pila, y la dirección en el puntero de pila se ajusta por el tamaño de los datos de partida.
- Una operación desapilar: un elemento de datos en la ubicación actual apuntado por el puntero de pila es eliminado, y el puntero de pila se ajusta por el tamaño de los datos de partida.

Hay muchas variaciones en el principio básico de las operaciones de pila. Cada pila tiene un lugar fijo en la memoria en la que comienza. Como los datos se añadirán a la pila, el puntero de pila es desplazado para indicar el estado actual de la pila, que se expande lejos del origen (ya sea hacia arriba o hacia abajo, dependiendo de la aplicación concreta).

Por ejemplo, una pila puede comenzar en una posición de la memoria de mil, y ampliar por debajo de las direcciones, en cuyo caso, los nuevos datos se almacenan en lugares que van por debajo de 1000, y el puntero de pila se decrementa cada vez que un nuevo elemento se agrega. Cuando un tema es eliminado de la pila, el puntero de pila se incrementa.

Los punteros de pila pueden apuntar al origen de una pila o de un número limitado de direcciones, ya sea por encima o por debajo del origen (dependiendo de la dirección en que crece la pila), sin embargo el puntero de pila no puede cruzar el origen de la pila. En otras palabras, si el origen de la pila está en la dirección 1000 y la pila crece hacia abajo (hacia las direcciones 999, 998, y así sucesivamente), el puntero de pila nunca debe ser incrementado más allá de 1000 (para 1001, 1002, etc.) Si un desapilar operación en la pila hace que el puntero de pila se deje atrás el origen de la pila, una pila se produce desbordamiento. Si una operación de apilar hace que el puntero de pila incremente o decremente más allá del máximo de la pila, en una pila se produce desbordamiento.

La pila es visualizada ya sea creciente de abajo hacia arriba (como pilas del mundo real), o, con el máximo elemento de la pila en una posición fija, o creciente, de izquierda a derecha, por lo que el máximo elemento se convierte en el máximo a "la derecha". Esta visualización puede ser independiente de la estructura real de la pila en la memoria. Esto significa que rotar a la derecha es mover el primer elemento a la tercera posición, la segunda a la primera y la tercera a la segunda. Aquí hay dos equivalentes visualizaciones de este proceso:

Una pila es normalmente representada en los ordenadores por un bloque de celdas de memoria, con los "de abajo" en una ubicación fija, y el puntero de pila de la dirección actual de la "cima" de células de la pila. En la parte superior e inferior se utiliza la terminología con independencia de que la pila crece realmente a la baja de direcciones de memoria o direcciones de memoria hacia mayores.

Apilando un elemento en la pila,se ajusta el puntero de pila por el tamaño de elementos (ya sea decrementar o incrementar, en función de la dirección en que crece la pila en la memoria), que apunta a la próxima celda, y copia el nuevo elemento de la cima en área de la pila. Dependiendo de nuevo sobre la aplicación exacta, al final de una operación de apilar, el puntero de pila puede señalar a la siguiente ubicación no utilizado en la pila, o tal vez apunte al máximo elemento de la pila. Si la pila apunta al máximo elemento de la pila, el puntero de pila se actualizará antes de que un nuevo elemento se apile, si el puntero que apunta a la próxima ubicación disponible en la pila, que se actualizará después de que el máximo elemento se apile en la pila.

Desapilando es simplemente la inversa de apilar. El primer elemento de la pila es eliminado y el puntero de pila se actualiza, en el orden opuesto de la utilizada en la operación de apilar.

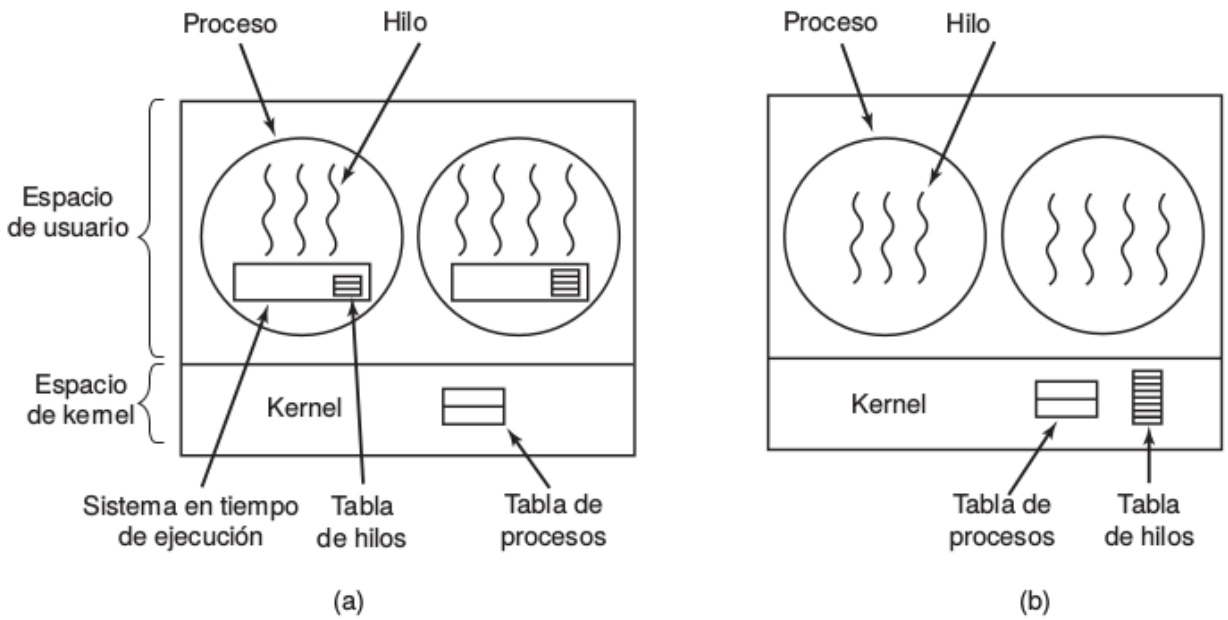
¿POR QUÉ HILOS?

¿Por qué alguien querría tener un tipo de proceso dentro de otro proceso? Resulta ser que hay varias razones de tener estos miniprocesos, conocidos como hilos. Ahora analizaremos algunos de ellos. La principal razón de tener hilos es que en muchas aplicaciones se desarrollan varias actividades a la vez. Algunas de ésas se pueden bloquear de vez en cuando. Al descomponer una aplicación en varios hilos secuenciales que se ejecutan en cuasi-paralelo, el modelo de programación se simplifica.
Ya hemos visto este argumento antes: es precisamente la justificación de tener procesos. En vez de pensar en interrupciones, temporizadores y conmutaciones de contexto, podemos pensar en procesos paralelos. Sólo que ahora con los hilos agregamos un nuevo elemento: la habilidad de las entidades en paralelo de compartir un espacio de direcciones y todos sus datos entre ellas. Esta habilidad es esencial para ciertas aplicaciones, razón por la cual no funcionará el tener varios procesos (con sus espacios de direcciones separados).

Un segundo argumento para tener hilos es que, como son más ligeros que los procesos, son mas fáciles de crear (es decir, rápidos) y destruir. En muchos sistemas, la creación de un hilo es de 10 a 100 veces más rápida que la de un proceso. Cuando el número de hilos necesarios cambia de manera dinámica y rápida, es útil tener esta propiedad. Una tercera razón de tener hilos es también un argumento relacionado con el rendimiento. Los hilos no producen un aumento en el rendimiento cuando todos ellos están ligados a la CPU, pero cuando hay una cantidad considerable de cálculos y operaciones de E/S, al tener hilos estas actividades se pueden traslapar, con lo cual se agiliza la velocidad de la aplicación. Por último, los hilos son útiles en los sistemas con varias CPUs, en donde es posible el verdadero paralelismo.

TIPOS DE HEBRAS

Hay dos formas principales de implementar un paquete de hilos: en espacio de usuario y en el kernel. La elección es un poco controversial y también es posible una implementación híbrida. Ahora describiremos estos métodos, junto con sus ventajas y desventajas.



Implementación de hilos a nivel usuario (a) y kernel (b)

Implementación de hilos a nivel usuario

El primer método es colocar el paquete de hilos completamente en espacio de usuario. El kernel no sabe nada acerca de ellos. En lo que al kernel concierne, está administrando

procesos ordinarios con un solo hilo. La primera ventaja, la más obvia, es que un paquete de hilos de nivel usuario

puede implementarse en un sistema operativo que no acepte hilos. Todos los sistemas operativos solían entrar en esta categoría e incluso hoy en día algunos todavía lo están. Con este método, los hilos se implementan mediante una biblioteca.

Todas estas implementaciones tienen la misma estructura general, que se ilustra en la figura (a). Los hilos se ejecutan encima de un sistema en tiempo de ejecución, el cual es una colección de procedimientos que administran hilos.

Cuando los hilos se administran en espacio de usuario, cada proceso necesita su propia tabla de hilos privada para llevar la cuenta de los hilos en ese proceso. Esta tabla es similar a la tabla de procesos del kernel, excepto porque sólo lleva la cuenta de las propiedades por cada hilo, como el contador de programa, apuntador de pila, registros, estado, etc. La tabla de hilos es administrada por el sistema en tiempo de ejecución. Cuando un hilo pasa al estado listo o bloqueado, la información necesaria para reiniciarlo se almacena en la tabla de hilos, en la misma forma exacta que el kernel almacena la información acerca de los procesos en la tabla de procesos.

Cuando un hilo hace algo que puede ponerlo en estado bloqueado en forma local —por ejemplo, esperar a que otro hilo dentro de su proceso complete cierto trabajo— llama a un procedimiento del sistema en tiempo de ejecución. Este procedimiento comprueba si el hilo debe ponerse en estado bloqueado. De ser así, almacena los registros del hilo (es decir, sus propios registros) en la tabla de hilos, busca en la tabla un hilo listo para ejecutarse y vuelve a cargar los registros de la máquina con los valores guardados del nuevo hilo. Tan pronto como se conmutan el apuntador de pila y el contador de programa, el nuevo hilo vuelve otra vez a la vida de manera automática. Si la máquina tiene una instrucción para guardar todos los registros y otra para cargarlos de una sola vez, toda la conmutación de hilos se puede realizar con sólo unas cuantas instrucciones. Realizar una conmutación de hilos como éste es por lo menos una orden de magnitud (o tal vez más) más veloz que hacer el trap al kernel y es un sólido argumento a favor de los paquetes de hilos de nivel usuario.

Sin embargo, hay una diferencia clave con los procesos. Cuando un hilo termina de ejecutarse por el momento, por ejemplo, cuando llama a thread_yield, el código de thread_yield puede guardar la información del hilo en la tabla de hilos. Lo que es más, así puede entonces llamar al planificador de hilos para elegir otro hilo y ejecutarlo. El procedimiento que guarda el estado del hilo y el planificador son sólo procedimientos locales, por lo que es mucho más eficiente invocarlos que realizar una llamada al kernel. Entre otras cuestiones, no se necesita un trap ni una conmutación de contexto, la memoria caché no necesita vaciarse, etc. Esto hace que la planificación de hilos sea muy rápida.

Los hilos de nivel usuario también tienen otras ventajas. Permiten que cada proceso tenga su propio algoritmo de planificación personalizado. Por ejemplo, para algunas aplicaciones, las que tienen un hilo recolector de basura, es una ventaja no tener que preocuparse porque

un hilo se detenga en un momento inconveniente. También se escalan mejor, ya que los hilos del kernel requieren sin duda algo de espacio en la tabla y en la pila del kernel, lo cual puede ser un problema si hay una gran cantidad de hilos.

A pesar de su mejor rendimiento, los paquetes de hilos de nivel usuario tienen algunos problemas importantes. El primero de todos es la manera en que se implementan las llamadas al sistema de bloqueo. Suponga que un hilo lee del teclado antes de que se haya oprimido una sola tecla. Es inaceptable permitir que el hilo realice la llamada al sistema, ya que esto detendrá a todos los hilos. Uno de los principales objetivos de tener hilos en primer lugar era permitir que cada uno utilizara llamadas de bloqueo, pero para evitar que un hilo bloqueado afectara a los demás. Con las llamadas al sistema de bloqueo, es difícil ver cómo se puede lograr este objetivo sin problemas.

Todas las llamadas al sistema se podrían cambiar para que quedaran sin bloqueo (por ejemplo, un read en el teclado sólo devolvería 0 bytes si no hubiera caracteres en el búfer), pero es inconveniente requerir cambios en el sistema operativo. Además, uno de los argumentos para los hilos de nivel usuario era precisamente que se podían ejecutar con los sistemas operativos e xistentes. Además, si se cambia la semántica de read se requerirán cambios en muchos programas de usuario.

Es posible otra alternativa si se puede saber de antemano si una llamada va a bloquear. En algunas versiones de UNIX existe una llamada al sistema (select), la cual permite al procedimiento que hace la llamada saber si una posible llamada a read realizará un bloqueo. Cuando esta llamada está presente, el procedimiento de biblioteca r ead se puede reemplazar con uno nuevo que primero realice una llamada a select y después sólo realice la llamada a read si es seguro (es decir, si no va a realizar un bloqueo). Si la llamada a read va a bloquear, no se hace; en vez de ello, se ejecuta otro hilo. La próxima vez que el sistema en tiempo de ejecución obtenga el control, puede comprobar de nuevo para ver si la llamada a read es ahora segura. Este método requiere que se vuelvan a escribir partes de la biblioteca de llamadas al sistema, es ineficiente y nada elegante, pero hay muy poca opción. El código colocado alrededor de la llamada al sistema que se encarga de la comprobación se conoce como envoltura.

Algo similar al problema de las llamadas al sistema de bloqueo es el problema de los fallos de página. Por ahora, basta con decir que las computadoras se pueden configurar de forma que no todo el programa se encuentre en memoria a la vez. Si el programa llama o salta a una instrucción que no esté en memoria, ocurre un fallo de página y el sistema operativo obtiene la instrucción faltante (y las instrucciones aledañas) del disco. A esto se le conoce como fallo de página. El proceso se bloquea mientras la instrucción necesaria se localiza y se lee. Si un hilo produce un fallo de página, el kernel (que ni siquiera sabe de la existencia de los hilos) bloquea naturalmente todo el proceso hasta que se complete la operación de E/S, incluso si otros hilos pudieran ser ejecutados.

Otro problema con los paquetes de hilos de nivel usuario es que, si un hilo empieza a ejecutarse, ningún otro hilo en ese proceso se ejecutará a menos que el primero renuncie

de manera voluntaria a la CPU. Dentro de un solo proceso no hay interrupciones de reloj, lo cual hace que sea imposible planificar procesos en el formato r ound robin (tomando turnos). A menos que un hilo entre al sistema en tiempo de ejecución por su propia voluntad, el planificador nunca tendrá una oportunidad.

Una posible solución al problema de los hilos que se ejecutan en forma indefinida es hacer que el sistema en tiempo de ejecución solicite una señal de reloj (interrupción) una vez por segundo para dar el control, pero esto también es crudo y complicado para un programa. No siempre son posibles las interrupciones periódicas de reloj a una frecuencia más alta e incluso si lo son, la sobrecarga total podría ser considerable. Lo que es peor: un hilo podría requerir también una interrupción de reloj, interfiriendo con el uso que el sistema en tiempo de ejecución da al reloj.

Otro argumento (que en realidad es el más devastador) contra los hilos de nivel usuario es que, por lo general, los programadores desean hilos precisamente en aplicaciones donde éstos se bloquean con frecuencia, como, por ejemplo, un servidor Web con multihilado. Estos hilos están realizando llamadas al sistema en forma constante. Una vez que ocurre un trap al kernel, de manera que lleve a cabo la llamada al sistema, no le cuesta mucho al kernel conmutar hilos si el anterior está bloqueado y, al hacer esto el kernel, se elimina la necesidad de realizar llamadas al sistema select en forma constante que comprueben si las llamadas al sistema read son seguras. Para las aplicaciones que en esencia están completamente limitadas a la CPU y raras veces se bloquean,

¿cuál es el objetivo de tener hilos? Nadie propondría con seriedad calcular los primeros n númerosprimos o jugar ajedrez utilizando hilos, debido a que no se obtiene ninguna ventaja al hacerlo de esta forma.

Implementación de hilos en el kernel

Ahora vamos a considerar el caso en que el kernel sabe acerca de los hilos y los administra. No se necesita un sistema en tiempo de ejecución para ninguna de las dos acciones, como se muestra en la figura anterior (b). Además, no hay tabla de hilos en cada proceso. En vez de ello, el kernel tiene una tabla de hilos que lleva la cuenta de todos los hilos en el sistema. Cuando un hilo desea crear un nuevo hilo o destruir uno existente, realiza una llamada al kernel, la cual se encarga de la creación o destrucción mediante una actualización en la tabla de hilos del kernel. La tabla de hilos del kernel contiene los registros, el estado y demás información de cada hilo. Esta información es la misma que con los hilos de nivel usuario, pero ahora se mantiene en el kernel, en vez de hacerlo en espacio de usuario (dentro del sistema en tiempo de ejecución). Esta información es un subconjunto de la información que mantienen tradicionalmente los kernels acerca desus procesos con un solo hilo; es decir, el estado del proceso. Además, el kernel también mantiene la tabla de procesos tradicional para llevar la cuenta de los procesos.

Todas las llamadas que podrían bloquear un hilo se implementan como llamadas al sistema, a un costo considerablemente mayor que una llamada a un procedimiento del sistema en tiempo de ejecución. Cuando un hilo se bloquea, el kernel, según lo que decida, puede ejecutar otro hilo del mismo proceso (si hay uno listo) o un hilo de un proceso distinto. Con los hilos de nivel usuario, el sistema en tiempo de ejecución ejecuta hilos de su propio proceso hasta que el kernel le quita la CPU (o cuando ya no hay hilos para ejecutar).

Debido al costo considerablemente mayor de crear y destruir hilos en el kernel, algunos sistemas optan por un método ambientalmente correcto, reciclando sus hilos. Cuando se destruye un hilo, se marca como no ejecutable pero las estructuras de datos de su kernel no se ven afectadas de ninguna otra forma. Más adelante, cuando debe crearse un hilo, se reactiva uno anterior, lo que ahorra cierta sobrecarga. El reciclaje de hilos también es posible para los hilos de nivel usuario, pero como la sobrecarga de la administración de los hilos es mucho menor, hay menos incentivos para hacer esto.

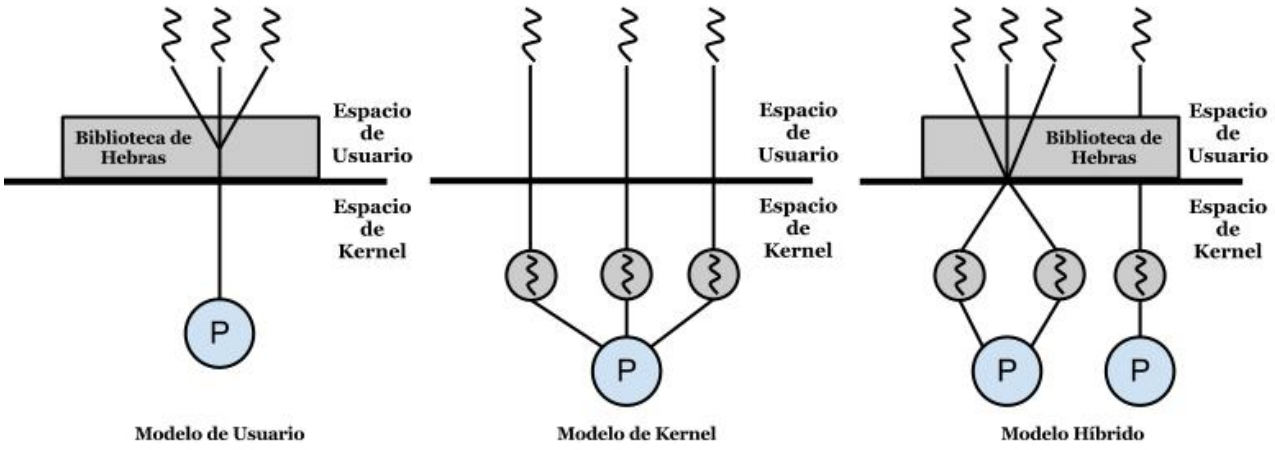
Los hilos de kernel no requieren de nuevas llamadas al sistema sin bloqueo. Además, si un hilo en un proceso produce un fallo de página, el kernel puede comprobar con facilidad si el proceso tiene otros hilos que puedan ejecutarse y de ser así, ejecuta uno de ellos mientras espera a que se traiga la página requerida desde el disco. Su principal desventaja es que el costo de una llamada al sistema es considerable, por lo que si las operaciones de hilos (de creación o terminación, por ejemplo) son comunes, se incurrirá en una mayor sobrecarga.

Los hilos de kernel resuelven sólo algunos problemas, no todos. Por ejemplo, ¿qué ocurre cuando un proceso con multihilamiento utiliza la llamada a fork para crear otro proceso? ¿El nuevo proceso tiene los mismos hilos que el anterior o tiene sólo uno? En muchos casos, la mejor elección depende de lo que el proceso planee realizar a continuación. Si va a llamar a exec para iniciar un nuevo programa, probablemente la elección correcta sea un hilo, pero si continúa en ejecución, tal vez lo mejor sea reproducir todos.

Otra cuestión es la relacionada con las señales. Recuerde que las señales se envían a los procesos, no a los hilos, por lo menos en el modelo clásico. Cuando entra una señal, ¿qué hilo debe hacerse cargo de ella? Es posible que los hilos puedan registrar su interés en ciertas señales, de manera que cuando llegue una señal se envíe al hilo que la está esperando. Pero ¿que ocurre si dos o más hilos se registran para la misma señal? Éstos son sólo dos de los problemas que introducen los hilos, pero hay más.

Implementaciones híbridas

Se han investigado varias formas de tratar de combinar las ventajas de los hilos de nivel usuario con los hilos de nivel kernel. Una de esas formas es utilizar hilos de nivel kernel y después multiplexar los hilos de nivel usuario con alguno o con todos los hilos de nivel kernel. Cuando se utiliza este método, el programador puede determinar cuántos hilos de kernel va a utilizar y cuántos hilos de nivel usuario va a multiplexar en cada uno. Este modelo proporciona lo último en flexibilidad.



Con este método, el kernel está consciente sólo de los hilos de nivel kernel y los planifica. Algunos de esos hilos pueden tener varios hilos de nivel usuario multiplexados encima de ellos; los hilos de nivel de usuario se crean, destruyen y planifican de igual forma que los hilos de nivel usuario en un proceso que se ejecuta en un sistema operativo sin capacidad de multihilamiento. En este modelo, cada hilo de nivel kernel tiene algún conjunto de hilos de nivel usuario que toman turnos para utilizarlo.

HEBRAS EN DISTINTOS S.O.

POSIX (IEEE 1003.1c) es el acrónimo de Portable Operating System Interface y la X viene de UNIX, un estándar de creación y sincronización de hebras. La API especifica el comportamiento de la biblioteca de hebras, común en el sistema operativo UNIX. Persiguen generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas. Estos estándares surgieron de un proyecto de normalización de las API y describen un conjunto de interfaces de aplicación adaptables a una gran variedad de implementaciones de sistemas operativos.

- **Hebras en Windows:** Implementados con el modelo uno-a-uno, cada hilo contiene un id de la hebra, el conjunto de registros, stacks de usuario y de kernel por separado y un área privada de almacenamiento de datos. Windows no soporta nativamente *pthreads*, sin embargo existe *Pthreads-w32* que busca proveer una implementación de forma portable.

- **Hebras en Linux:** Linux se refiere a las hebras como tareas. La creación de tareas se realiza a través de la llamada al sistema *clone()* y permite que una tarea hijo comparta el espacio de memoria de la tarea padre. Implementaciones de esta API estan disponibles en sistemas operativos tales como FreeBSD, NetBSD, OpenBSD, GNU/Linux, Mac OS X y Solaris.

PROCESOS VS. HEBRAS

Para algunos programas que se benefician de la concurrencia, la decisión de elegir procesos o hebras puede ser difícil. Vamos a describir algunas pautas para ayudar a elegir que modelo de concurrencia puede resultar más útil.

Todas las hebras de un programa deben ejecutar el mismo ejecutable. Un proceso hijo, por otro lado, puede ejecutar un archivo ejecutable diferente llamando a una función Exec.

Un hilo errante puede dañar otros hilos en el mismo proceso porque los hilos comparten el mismo espacio de memoria virtual y otros recursos. Por ejemplo, una escritura de memoria a través de un puntero no inicializado en una hebra puede dañar memoria visible para otro hilo.
Un proceso errante, por otro lado, no puede hacerlo porque cada proceso tiene un espacio de memoria propio.

Copiar memoria para un nuevo proceso añade una sobrecarga de rendimiento adicional en relación con la creación de un nuevo hilo. Sin embargo, la copia se realiza sólo cuando ésta memoria cambia, por lo que la penalidad es mínima si el proceso hijo sólo lee memoria.

Los hilos deben usarse para programas que necesitan un paralelismo más fino o preciso. Por ejemplo, si un problema se puede dividir en múltiples, casi idénticas tareas, las hebras pueden ser una buena opción. Los procesos deben ser usados para programas que necesitan un paralelismo más rudimentario o no lo necesitan.

Compartir datos entre las hebras es trivial porque los hilos comparten la misma memoria. La compartición de datos entre procesos requiere el uso de mecanismos IPC. Esto puede ser más engorroso, pero con muchos procesos disminuye la probabilidad de sufrir errores de concurrencia.

IDENTIFICACIÓN, CREACIÓN, TERMINACIÓN Y SINCRONIZACIÓN DE UNA HEBRA O HILO

Identificación de hebras

Todos y cada uno de los procesos tienen un único ID de proceso, y cada hebra un ID de hebra. A diferencia de los ID de procesos, que son únicos en el sistema, los ID de hebra tienen importancia sólo dentro del proceso al que pertenecen.
Recuerde que un ID de proceso, representado por el tipo de datos pid_t, es un entero no negativo. Un ID de hebra está representado por el tipo de datos pthread_t. Las implementaciones permiten usar una estructura para representar el tipo de datos pthread_t, ya que es un tipo de dato que no se puede tratar como entero. Por lo tanto, para comparar dos ID de hilo debe usarse una función:

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Devuelve: distinto de 0 si son iguales, 0 si no lo son.

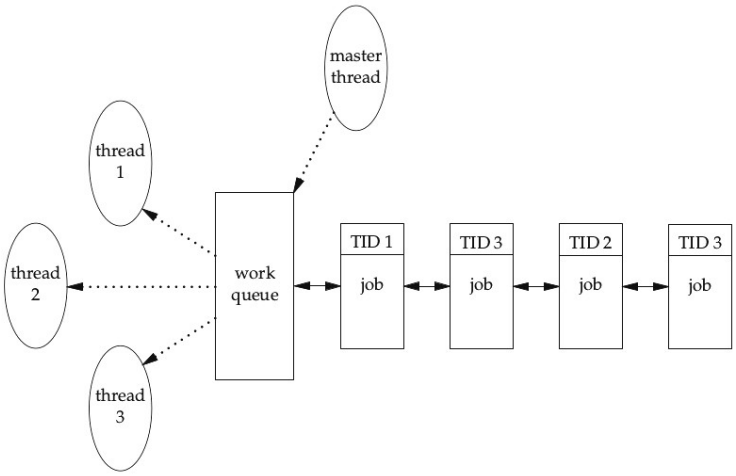
Creación de hebras

Linux 3.2.0 un tipo de dato unsigned long int pthread_t. Solaris 10 representa el tipo de dato pthread_t como un unsigned int. FreeBSD 8.0 and Mac OS X 10.6.8 usan un puntero a la estructura pthread que contiene el pthread_t.
La consecuencia de permitir al tipo de dato pthread_t ser una estructura es que no es un tipo de dato fácil para imprimir su valor. A veces, es útil imprimir ID de la hebra durante el programa de depuración, pero por lo general no hay necesidad de hacerlo.
Una hebra puede obtener su propio ID de hebra llamando a la función pthread_self.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Devuelve el ID de la hebra que lo llama.

Esta función se puede utilizar con pthread_equal cuando un hilo necesita identificar estructura de datos que están etiquetadas con su ID de hilo. Por ejemplo, un hilo maestro podría colocar asignaciones de trabajo en una cola y utilizar el ID de hilo para controlar qué trabajos van a cada hilo de trabajo. Esta situación se ilustra en esta figura.



Una sola hebra maestra coloca nuevos trabajos en la cola de trabajo. Un grupo de tres hilos de trabajo elimina trabajos de la cola. En lugar de permitir que cada hilo procese cualquier trabajo que esté al frente de la cola, el hilo maestro controla la asignación de trabajo colocando el ID del hilo que debe procesar el trabajo en cada estructura de trabajo. Cada hilo de trabajo entonces elimina sólo los trabajos que están etiquetados con su propio ID de hilo.

El modelo de proceso UNIX tradicional soporta sólo un hilo de control por proceso. Conceptualmente, esto es lo mismo que un modelo basado en hilos. Con pthreads, cuando se ejecuta un programa, también empieza un solo proceso con un sólo hilo de control. A medida que el programa se ejecuta . A medida que el programa se ejecuta, su comportamiento debe ser Indistinguible del proceso tradicional, hasta que crea más hilos de control.
Se pueden crear hilos adicionales llamando a la función pthread_create:

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr,
void*(*start_rtn)(void *), void *restrict arg);
```

Devuelve 0 si ha tenido éxito, distinto de 0 si ha ocurrido un error.

La ubicación de memoria señalada por tidp se establece en el ID de hilo de la nueva creación, siempre y cuando pthread_create tenga éxito. El argumento attr se usa para Personalizar varios atributos de hilo. Lo estableceremos en NULL para crear un hilo con los atributos predeterminados.

El subproceso recién creado comienza a ejecutarse en la dirección de la función start_rtn. Esta función toma un único argumento, arg, que es un puntero sin tipo. Si necesitas pasar más de un argumento a la función start_rtn, entonces necesitas almacenarlos en un estructura y pasar la dirección de la estructura en arg.

Cuando se crea una hebra, no hay ninguna garantía que se ejecute primero: la hebra creada o la hebra creadora. El hilo creado recientemente tiene acceso al
espacio de direcciones del proceso y hereda el entorno de punto flotante del hilo que llama y la máscara de señal. Sin embargo, el conjunto de señales pendientes para el hilo se borra.

Hay que tener en cuenta que las funciones pthread generalmente devuelven un código de error cuando fallan. Ellos no establecen errores del estilo errno como las otras funciones POSIX. Con hilos, al devolver el código de error de la función, restringe el alcance del error en la función que lo causó, en lugar de devolver algún estado global.

Ejemplo

Aunque no hay ninguna forma portátil de imprimir el ID del hilo, podemos escribir un pequeño programa de prueba que lo hace, para obtener alguna idea de cómo funcionan los hilos. El programa de la figura siguiente crea un hilo e imprime los ID de proceso e hilo del nuevo hilo y el hilo inicial.

```c
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
      (unsigned long)tid, (unsigned long)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Este ejemplo tiene dos rarezas, que son necesarias para manejar carreras entre el hilo principal y el nuevo hilo. La primera es la necesidad de usar sleep en el hilo principal. Si no duerme, el hilo principal podría salir, por lo tanto, terminar todo el proceso antes de que el nuevo hilo tenga la oportunidad de ejecutarse. Este comportamiento depende de la implementación y algoritmos de programación de hilos del sistema operativo.

La segunda rareza es que el nuevo hilo obtiene su ID de hilo llamando a pthread_self en lugar de leerlo de memoria compartida o recibirlo como argumento de su rutina de inicio de hilos. Hay que recordar que pthread_create devuelve el ID de hilo del hilo recién creado a través del primer parámetro (tidp). En nuestro caso, por ejemplo, el hilo principal almacena este ID en ntid, pero el nuevo hilo no puede utilizarlo de forma segura.
Si el nuevo hilo se ejecuta antes de que el hilo principal regrese de llamar a pthread_create, entonces el nuevo hilo verá el contenido no inicializado de ntid en lugar del ID de hilo.

Ejecutando el programa de la figura en Solaris OS

```
$ ./a.out
main thread: pid 20075 tid 1 (0x1)
new thread: pid 20075 tid 2 (0x2)
```

Como esperábamos, cada hilo tiene el mismo ID de proceso, pero diferentes IDs de hilo.

Ejecutando el programa en FreeBSD se obtiene:

```
$ ./a.out
main thread: pid 37396 tid 673190208 (0x28201140)
new thread: pid 37396 tid 673280320 (0x28217140)
```

Como esperábamos, ambos hilos tienen el mismo ID de proceso. Si miramos los ID de hilo como enteros decimales, los valores parecen extraños, pero si los miramos en formato hexadecimal, tienen más sentido. Como señalamos anteriormente, FreeBSD usa un puntero a una estructura de datos del hilo para su ID de hilo.

Se esperaría que Mac OS X sea similar a FreeBSD. Sin embargo, el identificador del hilo del hilo principal es de un rango de dirección diferente que el ID de hilo para hilos creados Con pthread_create:

```
$ ./a.out
main thread: pid 31807 tid 140735073889440 (0x7fff70162ca0)
new thread: pid 31807 tid 4295716864 (0x1000b7000)
```

Ejecutando el programa en linux se obtiene:

```
$ ./a.out
main thread: pid 17874 tid 140693894424320 (0x7ff5d9996700)
new thread: pid 17874 tid 140693886129920 (0x7ff5d91ad700)
```

Los IDs de hilo en Linux son como punteros, cada uno de ellos representados como unsigned-long int. La implementación de los hilos cambió entre Linux 2.4 y Linux 2.6. En el Linux 2.4, LinuxThreads implementó cada hilo con un proceso independiente. Esto hizo difícil conocer el comportamiento de los hilos POSIX. En Linux 2.6, el kernel de Linux y la librería de hilos fueron revisados para usar una nueva implementación de hilos denominada Native POSIX Thread Library (NPTL). Esto apoyaba un modelo de múltiples hilos dentro de un solo proceso y facilitaba soportar la semántica de los hilos POSIX.

Terminación de hebras

Si algun hilo de un proceso llama a exit, _Exit, o _exit, entonces termina el proceso entero. Similarmente, cuando la acción por defecto es terminar el proceso, una señal se envía al hilo que terminará el proceso entero.

Un solo hilo puede salir de tres maneras, deteniendo así su flujo de control, sin terminar todo el proceso.
1. El hilo puede simplemente volver a la rutina de inicio. El valor devuelto es el código de salida del hilo.
2. El hilo puede ser cancelado por otro hilo en el mismo proceso.
3. El hilo puede llamar a pthread_exit.

pthread_exit se define así:

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

El argumento rval_ptr es un puntero sin tipo, similar al único argumento pasado al iniciar la rutina. Este puntero está disponible para otros hilos en el proceso llamando a la función pthread_join.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

Devuelve 0 si tiene éxito, distinto de 0 en caso contrario.

El hilo que llama a la función se bloqueará hasta que el hilo especificado llame a pthread_exit, devuelva desde su rutina de inicio, o se cancele. Si el hilo simplemente regresó de su rutina de inicio, rval_ptr contendrá el código de retorno. Si se canceló el hilo, la ubicación de memoria especificado por rval_ptr se establece en PTHREAD_CANCELED.

Al llamar a pthread_join, colocamos automáticamente el hilo con el que estamos uniendo en el estado separado para que sus recursos puedan ser recuperados.
Si el hilo ya estaba en el estado desvinculado, pthread_join puede fallar, devolviendo EINVAL, aunque este comportamiento es específico de la implementación.
Si no estamos interesados en el valor de retorno de un hilo, podemos establecer rval_ptr en NULL. En este caso, llamar a pthread_join nos permite esperar el hilo especificado, pero no recuperar el estado de terminación del hilo.

Ejemplo

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

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{

     printf("thread 2 exiting\n");
     pthread_exit((void *)2);
 }

 int
 main(void)
 {
     int         err;
     pthread_t   tid1, tid2;
     void        *tret;

     err = pthread_create(&tid1, NULL, thr_fn1, NULL);
     if (err != 0)
         err_exit(err, "can't create thread 1");
     err = pthread_create(&tid2, NULL, thr_fn2, NULL);
     if (err != 0)
         err_exit(err, "can't create thread 2");
     err = pthread_join(tid1, &tret);
     if (err != 0)
         err_exit(err, "can't join with thread 1");
     printf("thread 1 exit code %ld\n", (long)tret);
     err = pthread_join(tid2, &tret);
     if (err != 0)
         err_exit(err, "can't join with thread 2");
     printf("thread 2 exit code %ld\n", (long)tret);
     exit(0);
 }
```

La figura anterior muestra cómo obtener el código de salida de un hilo que se ha terminado. La figura obtiene este estado de salida:

```
$ ./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

Como podemos ver, cuando sale un hilo llamando a pthread_exit o simplemente regresando desde la rutina de inicio, el estado de salida puede obtenerse mediante otro hilo llamando a pthread_join.
El puntero void que se pasó a pthread_create y pthread_exit se puede utilizar para pasar más de un solo valor. El puntero se puede utilizar para pasar la dirección de una

estructura que contenga información más compleja. Hay que tener cuidado cuidado de que la memoria utilizada por la estructura siga siendo válida cuando el llamador ha terminado. Si la estructura fue asignada en la pila de la persona que llama, por ejemplo, el contenido de la memoria podría haber cambiado en el tiempo en el que se usaba la estructura. Si un hilo asigna una estructura a su pila y pasa un puntero de esta estructura a pthread_exit, entonces la pila puede ser destruida y su memoria reutilizada para otra cosa en el momento en que el llamador de pthread_join intenta usarlo.

Ejemplo
El programa de la figura siguiente enseña el problema usando una variable automática (localizada en la pila) como argumento de pthread_exit.

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

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf("%s", s);
    printf("  structure at 0x%lx\n", (unsigned long)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo  foo = {1, 2, 3, 4};

    printfoo("thread 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2: ID is %lu\n", (unsigned long)pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    struct foo  *fp;

    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    sleep(1);
    printfoo("parent:\n", fp);
    exit(0);
}
```

Hacemos un uso incorrecto del argumento de pthread_exit.
Cuando ejecutamos el programa en linux, obtenemos:

```
$ ./a.out
thread 1:
structure at 0x7f2c83682ed0
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
parent starting second thread
thread 2: ID is 139829159933696
parent:
structure at 0x7f2c83682ed0
foo.a = -2090321472
foo.b = 32556
foo.c = 1
foo.d = 0
```

Por supuesto, los resultados varían, dependiendo de la arquitectura de memoria, el compilador, y la implementación de las librerías de hebras. El resultado en Solaris es similar:

```
$ ./a.out
thread 1:
structure at 0xffffffff7f0fbf30
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
parent starting second thread
thread 2: ID is 3
parent:
structure at 0xffffffff7f0fbf30
foo.a = -1
foo.b = 2136969048
foo.c = -1
foo.d = 2138049024
```

Como podemos ver, el contenido de la estructura (localizada en la pila de thread tid1) ha cambiado en el tiempo en el que la hebra principal accedía a la estructura. La pila de la segunda hebra (tid2) ha sobreescrito la pila de la primera hebra. Para arreglar el problema, podemos usar una estructura global para contener la estructura usando malloc.

En Mac OS X, obtenemos resultados diferentes:

```
$ ./a.out
thread 1:
structure at 0x1000b6f00
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
parent starting second thread
thread 2: ID is 4295716864
parent:
structure at 0x1000b6f00
Segmentation fault (core dumped)
```

En este caso, la memoria ya no es válida cuando el padre intenta acceder a la estructura pasada a ella por el primer hilo que salió, y el padre envía la señal SIGSEGV.
En FreeBSD, la memoria no se ha sobrescrito en el momento en que el padre accede a él, y conseguimos:

```
thread 1:
structure at 0xbf9fef88
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
parent starting second thread
thread 2: ID is 673279680
parent:
structure at 0xbf9fef88
foo.a = 1
foo.b = 2
foo.c = 3
foo.d = 4
```

Aunque la memoria todavía está intacta después de que el hilo salga, no podemos depender de que siempre sea este el caso. Ciertamente no es lo que observamos en las otras plataformas. Un hilo puede solicitar que otro en el mismo proceso se cancele llamando a la función pthread_cancel.

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

Devuelve 0 si tiene éxito, distinto de 0 si no lo tiene.

En las circunstancias predeterminadas, pthread_cancel hará que la hebra especificada por tid se comporte como si hubiera llamado pthread_exit con un argumento de PTHREAD_CANCELED. Sin embargo, un hilo puede elegir ignorar o controlar de otra manera cómo se cancela. Tenga en cuenta que pthread_cancel no espera a la hebra para terminar, simplemente hace la solicitud.

Un hilo puede organizar que se llamen las funciones cuando sale, similar a la forma en que la función atexit puede ser utilizada por un proceso para organizar esas funciones. Las funciones se conocen como controladores de limpieza de hebras.
Se puede establecer más de un controlador de limpieza para una hebra.

Los manejadores son grabados en una pila, lo que significa que se ejecutan en el orden inverso a los que se registraron.

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

La función pthread_cleanup_push programa la función de limpieza, rtn, para ser llamada con un argumento único, arg, cuando el hilo realiza uno de los siguientes comportamiento:
- Realiza una llamada a pthread_exit
- Responde a una solicitud de cancelación
- Realiza una llamada a pthread_cleanup_pop con un argumento de ejecución diferente de cero

Si el argumento de ejecución se pone a cero, no se llama a la función de limpieza. En cualquier caso, pthread_cleanup_pop elimina el controlador de limpieza establecido por la última llamada a pthread_cleanup_push.

Una restricción con estas funciones es que, porque pueden ser implementadas como macros, deben ser utilizadas en parejas emparejadas dentro del mismo ámbito en un hilo. La

definición de macro de pthread_cleanup_push puede incluir un carácter, en cuyo caso el carácter correspondiente está en la definición de pthread_cleanup_pop.

La siguiente figura muestra cómo usar los controladores de limpieza de hebras. Aunque el ejemplo es algo artificial, ilustra la mecánica implicada. Hay que tener en cuenta que aunque nunca se intente pasar cero como argumento a las rutinas de inicio del hilo, todavía se necesita emparejar las llamadas a pthread_cleanup_pop con las llamadas a pthread_cleanup_push;
De lo contrario, el programa podría no compilar.

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

void
cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)

{
    printf("thread 1 start\n");
    pthread_cleanup_push(cleanup, "thread 1 first handler");
    pthread_cleanup_push(cleanup, "thread 1 second handler");
    printf("thread 1 push complete\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 start\n");
    pthread_cleanup_push(cleanup, "thread 2 first handler");
    pthread_cleanup_push(cleanup, "thread 2 second handler");
    printf("thread 2 push complete\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

Ejecutando el programa de la figura en Linux o Solaris obtenemos:

```
$ ./a.out
thread 1 start
thread 1 push complete
thread 2 start
thread 2 push complete
cleanup: thread 2 second handler
cleanup: thread 2 first handler
thread 1 exit code 1
thread 2 exit code 2
```

A partir de la salida, se puede ver que ambos hilos comienzan correctamente y salen, pero que sólo se llama a los controladores de limpieza del segundo hilo. Por lo tanto, si el hilo termina por devolver desde su rutina de inicio, sus manejadores de limpieza no son llamados, aunque este comportamiento varía entre implementaciones.

Si ejecutamos el mismo programa en FreeBSD o Mac OS X, vemos que el programa incurre en una violación de segmentación. Esto sucede porque en estos sistemas, pthread_cleanup_push se implementa como una macro que almacena algún contexto sobre la pila. Cuando el hilo 1 vuelve entre la llamada a pthread_cleanup_push y la llamada a pthread_cleanup_pop, la pila se sobrescribe y estas plataformas intente utilizar este contexto (ahora dañado) cuando invoca a los controladores de limpieza. En el Single UNIX Specification, mientras está entre un par de llamadas igualadas a pthread_cleanup_push y pthread_cleanup_pop se devuelve un comportamiento indefinido. La única forma portátil de regresar entre estas dos funciones es llamar a pthread_exit.

Por ahora, se debería empezar a ver similitudes entre las funciones de hilo y funciones de proceso. La figura siguiente resume las funciones similares.

Process primitive	Thread primitive	Description
fork	pthread_create	create a new flow of control
exit	pthread_exit	exit from an existing flow of control
waitpid	pthread_join	get exit status from flow of control
atexit	pthread_cleanup_push	register function to be called at exit from flow of control
getpid	pthread_self	get ID for flow of control
abort	pthread_cancel	request abnormal termination of flow of control

De forma predeterminada, el estado de terminación de un hilo se conserva hasta que llamamos pthread_join para ese hilo. Después de separar un hilo, no podemos usar el pthread_join para esperar su estado de terminación, ya que llamar pthread_join para un hilo separado da como resultado un comportamiento indefinido. Podemos separar un hilo llamando a pthread_detach.

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

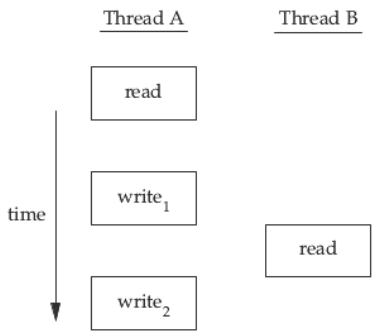
Devuelve 0 si tiene éxito, distinto de 0 si no lo tiene

Sincronización de hebras

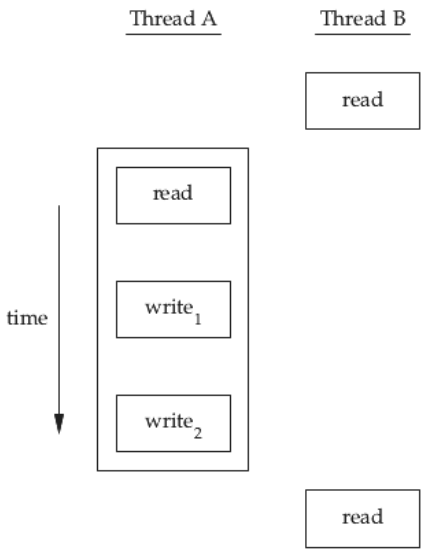
Cuando varios hilos de control comparten la misma memoria, necesitamos asegurarnos de que cada hilo ve una vista coherente de sus datos. Si cada hilo utiliza variables que otros hilos no leen ni modifican, no existen problemas de coherencia. De manera similar, si una variable es de sólo lectura, no hay problema de coherencia con más de un hilo al leer su valor al mismo tiempo. Sin embargo, cuando un hilo puede modificar una variable que el resto de hilos pueden leer o modificar, necesitamos sincronizar los hilos para asegurar que no utilice un valor no válido al acceder al contenido de la memoria de la variable.

Cuando un hilo modifica una variable, otros hilos pueden ver potencialmente inconsistencias al leer el valor de esa variable. En arquitecturas de procesador en el que la modificación lleva más de un ciclo de memoria, esto puede suceder cuando la lectura de memoria se entrelaza entre los ciclos de escritura de memoria. Por supuesto, este comportamiento es dependiente de la arquitectura, pero los programas portátiles no pueden hacer suposiciones acerca de qué tipo de arquitectura de procesador se está utilizando.

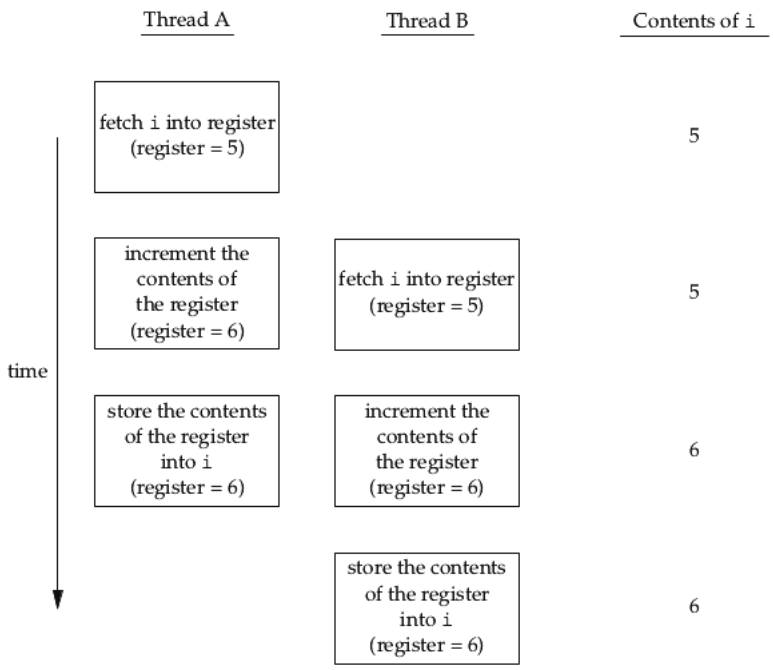
La figura muestra un ejemplo hipotético de dos hilos que leen y escriben en la misma variable. En este ejemplo, el hilo A lee la variable y luego escribe un nuevo valor, pero la operación de escritura toma dos ciclos de memoria. Si el hilo B lee la misma variable entre los dos ciclos de escritura, verá un valor inconsistente.



Para resolver este problema, los hilos tienen que utilizar un bloqueo que permitirá que sólo un hilo pueda acceder a la variable a la vez. La figura siguiente muestra esta sincronización. Si quiere leer la variable, el hilo B adquiere un bloqueo. Del mismo modo, cuando el hilo A actualiza la variable, adquiere el mismo bloqueo. Así, el hilo B no podrá leer la variable hasta que el hilo A libere el bloqueo.



También necesitamos sincronizar dos o más hilos que podrían intentar modificar la misma variable al mismo tiempo. Consideremos el caso en el que incrementamos una variable (Figura siguiente). La operación de incremento se descompone normalmente en tres pasos.
1. Leer la ubicación de la memoria en un registro.
2. Incrementar el valor en el registro.
3. Escribir el nuevo valor en la ubicación de memoria.



Si dos hilos tratan de incrementar la misma variable casi al mismo tiempo sin sincronización entre sí, los resultados podrían ser inconsistentes. Se puede terminar con un valor que es uno o dos mayor que antes, dependiendo del valor observado cuando el el segundo hilo inicia su funcionamiento. Si el segundo hilo realiza el paso 1 antes de que la primera hebra realice el paso 3, el segundo hilo leerá el mismo valor inicial que el primer hilo, incrementándolo y volviéndolo a escribirlo, sin efecto neto.

Si la modificación es atómica, entonces no hay problema. En el ejemplo anterior, si el incremento tomase sólo un ciclo de memoria, entonces no existiría problemas. Si nuestros datos siempre aparecen en una secuencia consistente, entonces no necesitamos ninguna sincronización adicional. Las operaciones son secuencialmente consistentes cuando múltiples subprocesos no pueden observar inconsistencias en nuestros datos.

BIBLIOGRAFIA

Libro Advanced Programming in the UNIX Environment, Third Edition de W. Richard Stevens
y Stephen A. Rago

Libro Sistemas operativos modernos, 3ed, de Tanenbaum

http://advancedlinuxprogramming.com/alp-folder/alp-ch04-threads.pdf

http://sistemasoperativos.angelfire.com/html/2.3.html

http://www.diclib.com/contador%20de%20programa/show/es/es_wiki_10/15654#.WGp77GfavQo

https://es.wikipedia.org/wiki/Contador_de_programa

https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)

https://sistemaoperativo.wikispaces.com/Hilos

http://wiki.inf.utfsm.cl/index.php?title=Motivacion_y_ventajas_de_las_hebras#Manejo_de_hebras_en_espacio_de_usuario_.28Modelo_Muchos_a_Uno.29