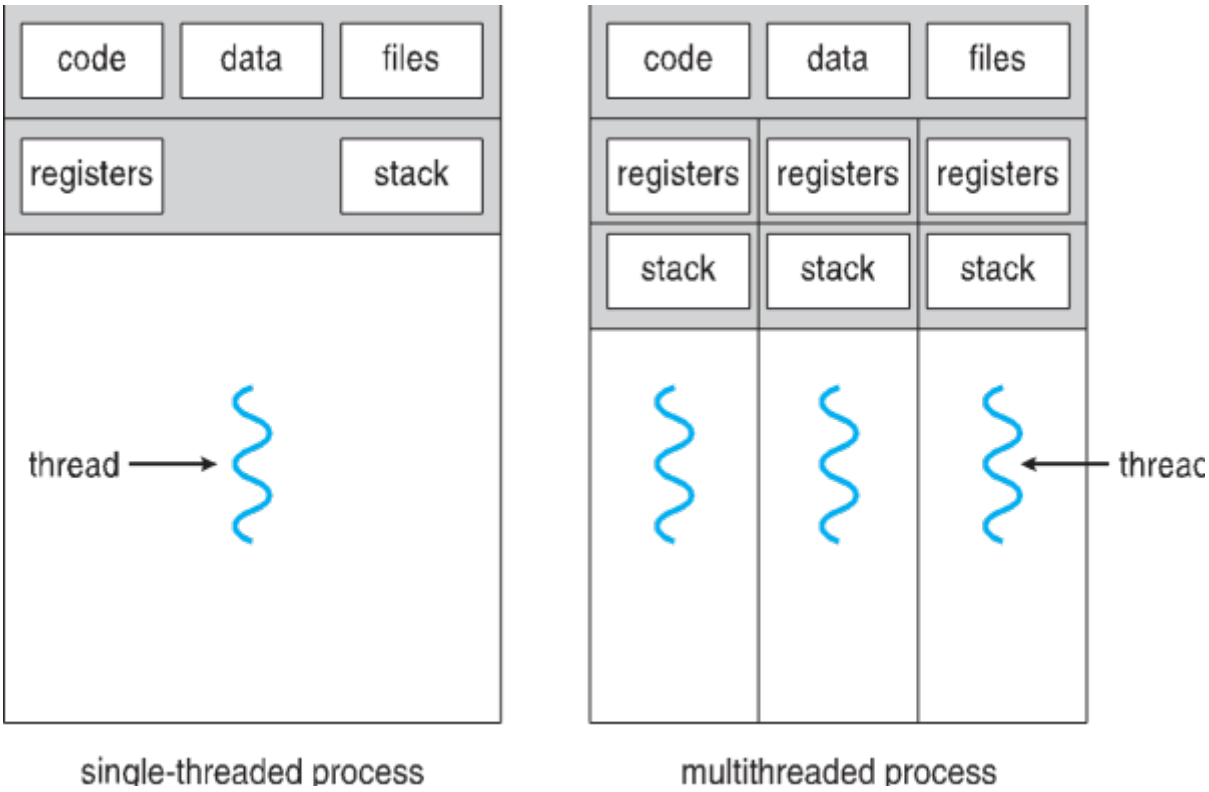
Threads

Una hebra es una unidad básica de utilización de la CPU; comprende un ID de hebra, un contador de programa, un conjunto de registros y una pila. Comparte con otras hebras que pertenecen al mismo proceso la sección de código, la sección de datos y otros recursos del sistema operativo, como los archivos abiertos y las señales. Un proceso tradicional (o proceso pesado) tiene una sola hebra de control. Si un proceso tiene, por el contrario, múltiples hebras de control, puede realizar más de una tarea a la vez.



Diferencia entre un proceso tradicional monohebra y un proceso multihebra

Muchas aplicaciones que se ejecutan en las computadoras modernas de escritorio son multihebra. Normalmente, una aplicación se implementa como un proceso propio con varias hebras de control. Por ejemplo, un explorador web puede tener una hebra para mostrar imágenes o texto mientras que otra hebra recupera datos de la red. Un procesador de textos puede tener una hebra para mostrar gráficos, otra hebra para responder a las pulsaciones de teclado del usuario y una tercera hebra para el corrector ortográfico y gramatical que se ejecuta en segundo plano. En determinadas situaciones, una misma aplicación puede tener que realizar varias tareas similares. Por ejemplo, un servidor web acepta solicitudes de los clientes que piden páginas web, imágenes, sonido, etc. Un servidor web sometido a una gran carga puede tener varios (quizá, miles) de clientes accediendo de forma concurrente a él. Si el servidor web funcionara corno un proceso tradicional de una sola hebra, sólo podría dar servicio a un cliente cada vez y la cantidad de tiempo que un cliente podría tener que esperar para que su solicitud fuera servida podría ser enorme.

Una solución es que el servidor funcione como un solo proceso de aceptación de solicitudes. Cuando el servidor recibe una solicitud, crea otro proceso para dar servicio a dicha solicitud. De hecho, este método de creación de procesos era habitual antes de que las hebras se popularizaran. Lla creación

de procesos lleva tiempo y hace un uso intensivo de los recursos. Si el nuevo proceso va a realizar las mismas tareas que los procesos existentes, ¿por qué realizar todo ese trabajo adicional? Generalmente, es más eficiente usar un proceso que contenga múltiples hebras. Según este método, lo que se hace es dividir en múltiples hebras el proceso servidor web. El servidor crea una hebra específica para escuchar las solicitudes de cliente y cuando llega una solicitud, en lugar de crear otro proceso, el servidor crea otra hebra para dar servicio a la solicitud.

Las hebras también juegan un papel importante en los sistemas de llamada a procedimientos remotos (RPC). Las RPC permiten la comunicación entre procesos proporcionando un mecanismo de comunicación similar a las llamadas a funciones o procedimientos ordinarias. Normalmente, los servidores RPC son multihebra. Cuando un servidor recibe un mensaje, sirve el mensaje usando una hebra específica. Esto permite al servidor dar servicio a varias solicitudes concurrentes. Los sistemas RMI de Java trabajan de forma similar. Por último, ahora muchos kernel de sistemas operativos son multihebra; hay varias hebras operando en el kernel y cada hebra-realiza una tarea específica, tal como gestionar dispositivos o tratar interrupciones. Por ejemplo, Solaris crea un conjunto de hebras en el kernel específicamente para el tratamiento de interrupciones; Linux utiliza una hebra del kernel para gestionar la cantidad de memoria libre en el sistema.

Ventajas

Las ventajas de la programación multihebra pueden dividirse en cuatro categorías principales:

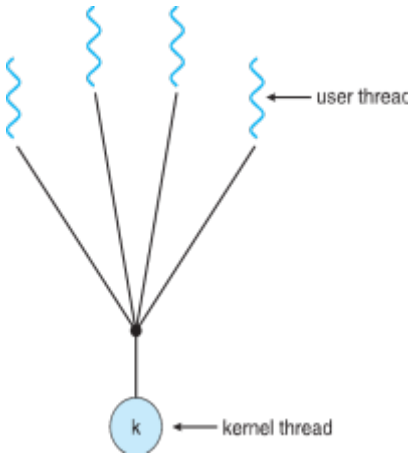
- **Capacidad de respuesta.** El uso de múltiples hebras en una aplicación interactiva permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado o realizando una operación muy larga, lo que incrementa la capacidad de respuesta al usuario. Por ejemplo, un explorador web multihebra permite la interacción del usuario a través de una hebra mientras que en otra hebra se está cargando una imagen.
- **Compartición de recursos.** Por omisión, las hebras comparten la memoria y los recursos del proceso al que pertenecen. La ventaja de compartir el código y los datos es que permite que una aplicación tenga varias hebras de actividad diferentes dentro del mismo espacio de direcciones.
- **Economía.** La asignación de memoria y recursos para la creación de procesos es costosa. Dado que las hebras comparten recursos del proceso al que pertenecen, es más económico crear y realizar cambios de contexto entre unas y otras hebras. Puede ser difícil determinar empíricamente la diferencia en la carga de adicional de trabajo administrativo pero, en general, se consume mucho más tiempo en crear y gestionar los procesos que las hebras. Por ejemplo, en Solaris, crear un proceso es treinta veces más lento que crear una hebra, y el cambio de contexto es aproximadamente cinco veces más lento.
- **Utilización sobre arquitecturas multiprocesador.** Las ventajas de, usar configuraciones multihebra pueden verse incrementadas significativamente en una arquitectura multiprocesador, donde las hebras pueden ejecutarse en paralelo en los diferentes procesadores. Un proceso monohebra sólo se puede ejecutar en una CPU, independientemente de cuántas haya disponibles. Los mecanismos multihebra en una máquina con varias CPU incrementan el grado de concurrencia.

Modelos multihebra

Desde el punto de vista práctico, el soporte para hebras puede proporcionarse en el nivel de usuario (para las hebras de usuario) o por parte del kernel (para las hebras del kernel). El soporte para las hebras de usuario se proporciona por encima del kernel y las hebras se gestionan sin soporte del mismo, mientras que el sistema operativo soporta y gestiona directamente las hebras del kernel. Casi todos los sistemas operativos actuales, incluyendo Windows 7, Linux. Mac OS x, Solaris y Tru64 UNIX (antes Digital UNIX) soportan las hebras de kernel. En último término, debe existir una relación entras las hebras de usuario y las del kernel; vamos a ver tres formas de establecer esta relación.

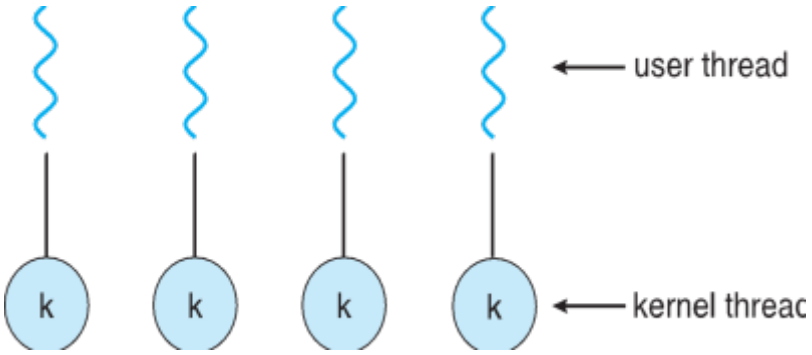
Modelo muchos-a-uno

El modelo muchos-a-uno asigna múltiples hebras del nivel de usuario a una hebra del kernel. La gestión de hebras se hace mediante la biblioteca de hebras en el espacio de usuario, por lo que resulta eficiente, pero el proceso completo se bloquea si una hebra realiza URa llamada bloqueante al sistema. También, dado que sólo una hebra puede acceder al kernel cada vez, no podrán ejecutarse varias hebras en paralelo sobre múltiples procesadores. El sistema de hebras Green, una biblioteca de hebras disponibles en Solaris, usa este modelo, asi como GNU Portable Threads.



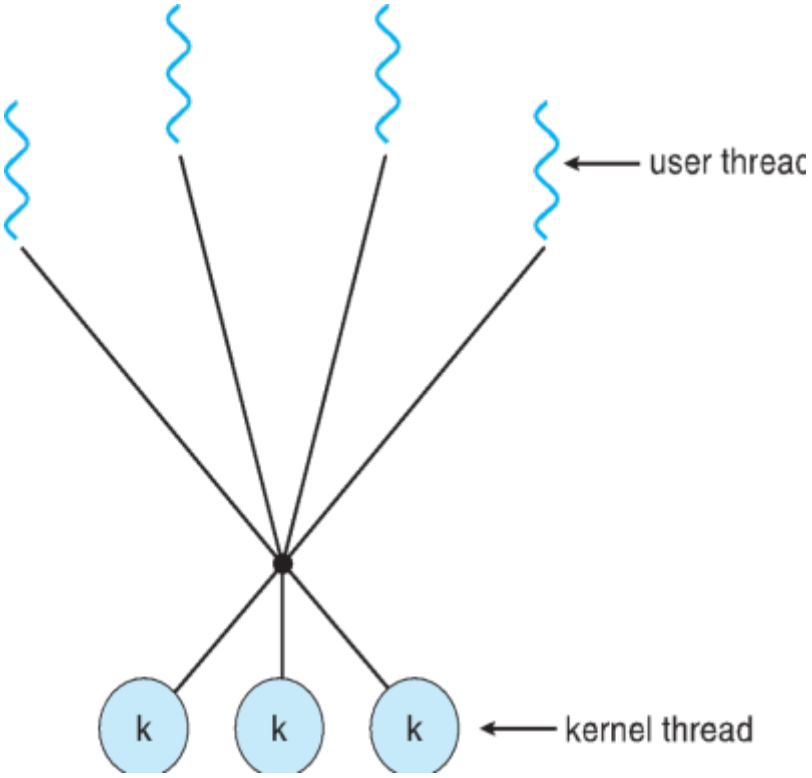
Modelo uno-a-uno

El modelo uno-a-uno asigna cada hebra de usuario a una hebra del kernel. Proporciona una mayor concurrencia que el modelo muchos-a-uno, permitiendo que se ejecute otra hebra mientras una hebra hace una llamada bloqueante al sistema; también permite que se ejecuten múltiples hebras en paralelo sobre varios procesadores. El único inconveniente de este modelo es que crear una hebra de usuario requiere crear la correspondiente hebra del kernel, Dado que la carga de trabajo administrativa para la creación de hebras del kernel puede repercutir en el rendimiento de una aplicación, la mayoría de las implementaciones de este modelo restringen el número de hebras soportadas por el sistema. Linux, junto con la familia de sistemas operativos Windows (incluyendo Windows 95, 98, NT, 2000, XP y 7), implementan el modelo uno-a-uno.



Modelo muchos-a-muchos

El modelo muchos-a-muchos multiplexa muchas hebras de usuario sobre un número menor o igual de hebras del kemeL El número de hebras del kemel puede ser específico de una determinada aplicación o de una determinada máquina (pueden asignarse más hebras del kemel a una aplicación en un sistema multiprocesador que en uno de un solo procesador). Mientras que el modelo muchos-a-uno permite al desarrollador crear tantas hebras de usuario como desee, no se consigue una concurrencia real, ya que el kernel sólo puede planificar la ejecución de una hebra cada vez. El modelo uno-a-uno permite una mayor concurrencia, pero el desarrollador debe tener cuidado de no crear demasiadas hebras dentro de una aplicación (y, en algunos casos, el número de hebras que pueda crear estará limitado). El modelo muchos-a-muchos no sufre ninguno de estos inconvenientes. Los desarrolladores pueden crear tantas hebras de usuario como sean necesarias y las correspondientes hebras del kemel pueden ejecutarse en paralelo en un multiprocesador. Asimismo, cuando una hebra realiza una llamada bloqueante al sistema, el kemel puede planificar otra hebra para su ejecución.



Una popular variación del modelo muchos-a-muchos multiplexa muchas hebras del nivel de usuario sobre un número menor o igual de hebras del kernel, pero también permite acoplar una hebra de usuario a una hebra del kernel. Algunos sistemas operativos como IRIX, HP-UX y Tru64 UNIX emplean esta variante, que algunas veces se denomina modelo de dos niveles. El sistema operativo Solaris

permitía el uso del modelo de dos niveles en las versiones anteriores a Solaris 9. Sin embargo, a partir de Solaris 9, este sistema emplea el modelo uno-a-uno.

Bibliotecas de hebras

Una biblioteca de hebras proporciona al programador una API para crear y gestionar hebras. Existen dos formas principales de implementar una biblioteca de hebras. El primer método consiste en proporcionar una biblioteca enteramente en el espacio de usuario, sin ningún soporte del kernel. Todas las estructuras de datos y el código de la biblioteca se encuentran en el espacio de usuario. Esto significa que invocar a una función de la biblioteca es como realizar una llamada a una función local en el espacio de usuario y no una llamada. al sistema.

El segundo método consiste en implementar una biblioteca en el nivel del kernel, soportada directamente por el sistema operativo. En este caso, el código y las estructuras de datos de la biblioteca se encuentran en el espacio del kernel. Invocar una función en la API de la biblioteca normalmente da lugar a que se produzca una llamada al sistema dirigida al kernel.

Las tres principales bibliotecas de hebras actualmente en uso son: (1) POSIX Pthreads, (2) Win32 y (3) Java. Pthreads. la extensión de hebras del estándar POSIX, puede proporcionarse como biblioteca del nivel de usuario o del nivel de kernel. La biblioteca de hebras de Win32 es una biblioteca del nivel de kernel disponible en los sistemas Windows .. La API de hebras Java permite crear y gestionar directamente hebras en los programas Java. Sin embargo, puesto que en la mayoría de los casos la]VM se ejecuta por encima del sistema operativo del host, la API de hebras Java se implementa habitualmente usando una biblioteca de hebras disponible en el sistema host. Esto significa que, normalmente, en los sistemas Windows, las hebras Java se implementan usando la API de Win32, mientras que en los sistemas Linux se suelen implementar empleando Pthreads.

Diseñaremos un programa multihebra que calcule la sumatoria de un número entero no negativo en una hebra específica empleando la muy conocida función:

Sum=∑_(i=0)^N▒i

Por ejemplo, si N fuera igual a 5, esta función representaría la sumatoria desde 0 hasta 5, que es 15. Cada uno de los tres programas se ejecutará especificando el límite superior de la sumatoria a través de la línea de comandos; por tanto, si el usuario escribe 8, a la salida se obtendrá la suma de los valores enteros comprendidos entre 0 y 8.

Pthreads

Pthreads se basa en el estándar POSlX (IEEE 1003.1c) que define una API para la creación y sincronización de hebras. Se trata de una especificación para el comportamiento de las hebras, no de una implementación. Los diseñadores de sistemas operativos pueden implementar la especificación de la forma que deseen. Hay muchos sistemas que implementan la especificación Pthreads, incluyendo Solaris, Linux; Mac OS X v Tru64 UNIX. También hay disponibles implementaciones de libre distribución para diversos sistemas operativos Windows. El programa C mostrado ilustra la API básica de Pthreads mediante un ejemplo de creación de un programa multihebra que calcula la sumatoria de un entero no negativo en una hebra específica. En un programa Pthreads, las diferentes hebras inician su ejecución en una función específica. En la Figura de abajo, se trata de la función runner(). Cuando este programa se inicia, da comienzo una sola hebra de control en main (). Después de

algunas inicializaciones, main () crea una segunda hebra que comienza en la función runner(). Ambas hebras comparten la variable global sum Analicemos más despacio este programa. Todos los programas Pthreads deben incluir el archivo de cabecera pthread.h. La instrucción pthread_t tid declara el identificador de la hebra que se va a crear, Cada hebra tiene un conjunto de atributos, que incluye el tamaño de la pila y la información de planifícación. La declaración pthread_attr_t attr representa los atributos de la hebra; establecemos los atributos en la llamada a la función pthread_attr_init (&attr). Dado que no definimos explícitamente ningún atributo, se usan los atributos predeterminados. Para crear otra hebra se usa la llamada a la función pthread_create( ). Además de pasar el identificador de la hebra y-los atributos de la misma, también pasamos el nombre de la función en la que la nueva hebra comenzará la ejecución, que en este caso es la función runner( ) .Por último, pasamos el parámetro entero que se proporcionó en la línea de comandos, argv (1].

En este punto, el programa tiene dos hebras: la hebra inicial (o padre) en main () y la hebra de la sumatoria (o hijo) que realiza la operación de suma en la función runner( ). Después de crear la hebra sumatoria, la hebra padre esperará a que se complete llamando a la función pthread_join. La hebra sumatoria se completará cuando llame a la función pthread_exit( ) Una vez que la hebra sumatoria ha vuelto, la hebra padre presenta a la salida el valor de la variable compartida sum,

From:
http://wiki.educabit.ar/ - **Wiki Sistemas**

Permanent link:
http://wiki.educabit.ar/doku.php?id=so_procthreads

Last update: **2025/09/11 22:48**