Procesos
Threads

Contenido

Proceso Threads

Prologo

Bienvenidos al resumen adaptado del libro Fundamentos de Sistemas Operativos (Silberchtaz-Galvin-Gagne) que acompaña la cátedra Sistemas Operativos y Redes de la carrera de Informática. Este documento tiene como objetivo proporcionar un acceso conciso y claro a los conceptos fundamentales presentados en el libro de texto principal, adaptado específicamente para satisfacer las necesidades y los objetivos del curso.

Los sistemas operativos son la columna vertebral de la informática moderna, y comprender su funcionamiento es esencial para cualquier estudiante de ciencias de la computación. Este resumen condensa los conceptos clave, los principios fundamentales y las técnicas avanzadas discutidas en el libro de texto original, proporcionando una guía clara y accesible para el estudio y la comprensión de los sistemas operativos.

A lo largo de este resumen, encontrarás una serie de capítulos que abordan temas importantes como la gestión de procesos, la gestión de memoria, la planificación de la CPU, entre otros. Cada capítulo se ha diseñado cuidadosamente para ofrecer una visión completa y bien estructurada de los conceptos presentados en el libro de texto, acompañado de ejemplos prácticos y explicaciones claras para facilitar el aprendizaje.

Espero que este resumen sea una herramienta valiosa para complementar tu estudio de sistemas operativos y te ayude a alcanzar tus objetivos te deseamos mucho éxito en tu cursada.

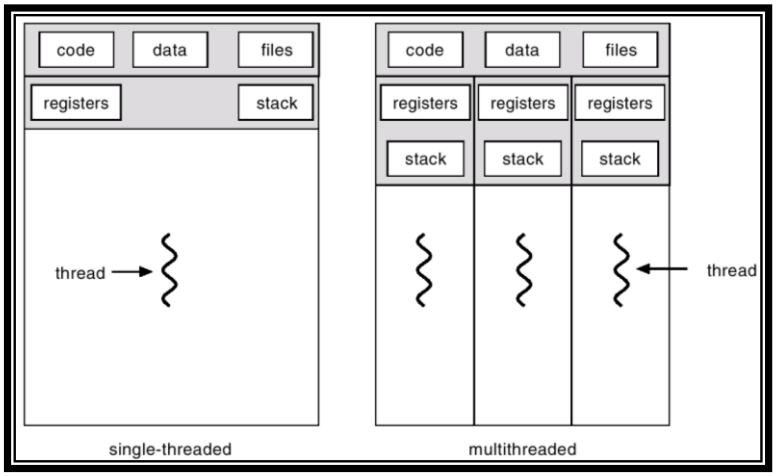
Lic. Mariano Vargas

Objetivos del resumen

- Presentar el concepto de hilo, una unidad fundamental de utilización de la CPU que conforma los fundamentos de los sistemas informáticos multihilos.
- Explicar las API de las bibliotecas de hilos de Pthreads, Win32 y Java.

Introducción

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



Diferencia entre un proceso tradicional proceso pesado y un proceso multihilo

Normalmente, una aplicación se implementa como un proceso propio con varios hilos. Por ejemplo, un explorador web puede tener un hilo para mostrar imágenes o texto mientras que otro hilo recupera datos de la red. Un procesador de textos puede tener un thread para mostrar gráficos, otro thread para responder al teclado del usuario y una tercer thread para el corrector ortográfico y gramatical que se ejecuta en segundo plano.

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 pesado de un solo hilo, sólo podría dar servicio a un cliente cada vez (dependiendo exclusivamente del grado de multiprogramación del Sistema Operativo del servidor) y la cantidad de tiempo que un cliente podría tener que esperar para que su solicitud fuera atendida podría ser enorme.

Hoy en día, la mayoría de los sistemas operativos implementan su kernel de son multithreads; hay varios hilos operando en el kernel y cada uno tarea específica, tal como gestionar dispositivos o tratar interrupciones.

Ventajas

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

- ✓ Capacidad de respuesta.
- ✓ Compartición de recursos.
- ✓ Economía.
- ✓ Utilización.

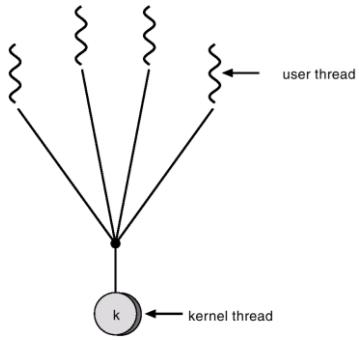
Modelos multihilos

Desde el punto de vista práctico, el soporte para threads puede proporcionarse en el nivel de usuario (para los hilos de usuario) o por parte del kernel (para los hilos del kernel). El soporte para los hilos de usuario se proporciona por encima del kernel y los hilos se gestionan sin soporte del mismo, mientras que el sistema operativo soporta y gestiona directamente las hilos del kernel. Casi todos los sistemas operativos actuales, incluyendo Windows, Linux. Mac OS x, Solaris y Tru64 UNIX (antes Digital UNIX) soportan las hilos de kernel.

En último término, debe existir una relación entras los hilos de usuario y los del kernel; vamos a ver tres formas de establecer esta relación.

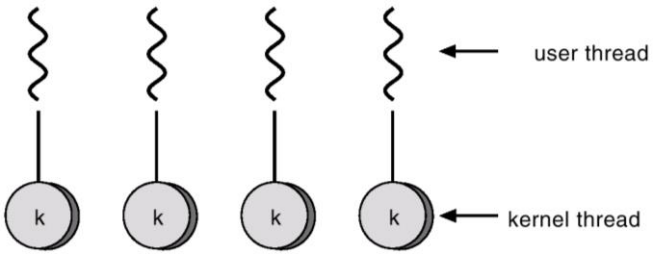
Modelo muchos-a-uno

El modelo muchos-a-uno se asigna múltiples hilos del nivel de usuario a un hilo del kernel. La gestión de hilos se hace mediante la biblioteca de hilos en el espacio de usuario, por lo que resulta eficiente, pero el proceso completo se bloquea si un hilo realiza una llamada bloqueante al sistema. También, dado que sólo un hilo puede acceder al kernel cada vez, no podrán ejecutarse varios hilos en paralelo sobre múltiples procesadores.



Modelo uno-a-uno

El modelo uno-a-uno asigna cada hilo de usuario a un hilo del kernel. Proporciona una mayor concurrencia que el modelo muchos-a-uno, permitiendo que se ejecute otro hilo mientras un hilo hace una llamada bloqueante al sistema; también permite que se ejecuten múltiples hilos en paralelo sobre varios procesadores. El único inconveniente de este modelo es que crear un hilo de usuario requiere crear el correspondiente hilo del kernel, Dado que la carga de trabajo administrativa para la creación de hilos del kernel puede repercutir en el rendimiento de una aplicación, la mayoría de las implementaciones de este modelo restringen el número de hilos soportados por el sistema. Linux, junto con la familia de sistemas operativos Windows (incluyendo Windows 10), implementan el modelo uno-a-uno.



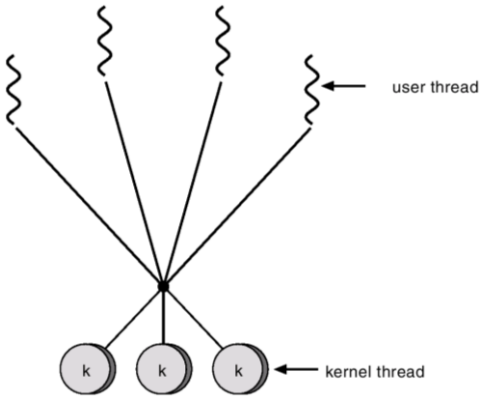
Modelo muchos-a-muchos

El modelo muchos-a-muchos multiplexa muchos hilos de usuario sobre un número menor o igual de hilos del kemel. El número de hilos del kemel puede ser específico de una determinada aplicación o de una determinada máquina (pueden asignarse más hilos 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 tantos hilos de usuario como desee, no se consigue una concurrencia real, ya que el kernel sólo puede planificar la ejecución de un hilo cada vez.

El modelo uno-a-uno permite una mayor concurrencia, pero el desarrollador debe tener cuidado de no crear demasiados hilos dentro de una aplicación (y, en algunos casos, el número de hilos que pueda crear estará limitado). El modelo muchos-a-muchos no sufre ninguno de estos inconvenientes. Los desarrolladores pueden crear tantos hilos de usuario como sean necesarios y los correspondientes hilos del kemel pueden ejecutarse en paralelo en un multiprocesador.

Asimismo, cuando un hilo realiza una llamada bloqueante al sistema, el kemel puede planificar otro hilo para su ejecución.



Una variación del modelo muchos-a-muchos multiplexa muchos hilos del nivel de usuario sobre un número menor o igual de hilos del kernel, pero también permite acoplar una hilo de usuario a un hilo del kernel. Algunos sistemas operativos como UNIX emplean esta variante, que algunas veces se denomina modelo de dos niveles.

Bibliotecas de hilos

Una biblioteca de hilos proporciona al programador una API para crear y gestionar hilos. Existen dos formas principales de implementar una biblioteca de hilos. 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 hilos actualmente en uso son: POSIX Pthreads, Win32 y Java.

Pthreads

Pthreads se basa en el estándar POSIX (IEEE 1003.1c) que define una API para la creación y sincronización de hilos. Se trata de una especificación para el comportamiento de los hilos, no de una implementación. Los diseñadores de sistemas operativos pueden implementar la especificación de la forma que deseen.

Pthread.lib: La biblioteca de hilos POSIX de Linux

Ejemplo 1: holaso.c

Para escribir programas multihilo en C podemos utilizar la biblioteca pthread que implementa el standard POSIX (Portable Operating System Interface). Para ello en nuestro programa debemos incluir algunas cabeceras y a la hora de compilar es necesario linkear el programa con la librería. Veamos con un ejemplo sencillo.

```
#include <stdio.h>
#include <pthread.h>

void* hola_so(void* arg) {
   printf("Soy estudiante de Sistemas Operativos!\n");
   return NULL;
}

int main() {
   pthread_t thread_id;
   pthread_create(&thread_id, NULL, hola_so, NULL);
   pthread_join(thread_id, NULL);
   return 0;
}
```

En este ejemplo, primero incluimos las librerías stdio.h y pthread.h para poder utilizar las funciones de la biblioteca threads. A continuación, definimos una función

llamada hola_so que será ejecutada por un hilo. Esta función simplemente imprime por pantalla "Soy estudiante de Sistemas Operativos!" y devuelve NULL.
En el main, declaramos una variable thread_id del tipo pthread_t que será utilizada para identificar el hilo que vamos a crear.

Luego, llamamos a la función pthread_create pasándole como argumentos la variable thread_id, NULL (para utilizar los atributos de hilo por defecto), la función hola_so que queremos que ejecute el hilo, y NULL como argumento para la función hola_so.

Por último, llamamos a pthread_join pasándole la variable thread_id y NULL como argumentos. Esto hace que el hilo principal espere a que el hilo creado por pthread_create termine de ejecutar la función hola_so antes de continuar con la ejecución del resto del código, se llama a pthread_join, que espera a que el thread identificado por thread_id termine su ejecución. En este caso, como la función hola_so no devuelve ningún valor, se utiliza NULL como segundo argumento.

Para compilar este último ejemplo, abrimos la consola de Linux:

```
gcc -pthread -o holaso holaso.c
```

Ejemplo 2: holamundo.c

Ahora hagamos un programa que escriba “Hola Mundo” utilizando para ello dos hilos distintos, uno para la palabra Hola y otro para la palabra Mundo. Además, hagamos que cada palabra se escriba muy lentamente para ver bien el efecto.

```
# include <stdio .h>
# include <stdlib .h>
# include <string .h>
# include <unistd .h>
# include <pthread .h>

void * hola(void *arg ) {
  char *msg = " Hola";
  int i;

  for ( i = 0 ; i < strlen ( msg ) ; i++ ) {
    printf (" %c", msg [i]);
    fflush ( stdout );
    usleep (1000000) ;
  }
return NULL;
}

void * mundo ( void *arg ) {
char *msg = " mundo ";
int i;
for ( i = 0 ; i < strlen ( msg ) ; i++ ) {
printf (" %c", msg [i]);
fflush ( stdout );
usleep (1000000) ;
}
return NULL;
```

```
}


int main(int argc , char *argv []) {
pthread_t h1;
pthread_t h2;
pthread_create (&h1 , NULL , hola , NULL );
pthread_create (&h2 , NULL , mundo , NULL);
printf ("Fin \n");


}
```

Como se puede ver en el listado, hemos escrito una función para cada tarea que queremos realizar en un hilo. El estándar POSIX impone que toda función que vaya a ejecutarse en un hilo debe tener un parámetro de entrada de tipo puntero a void (void *) y tiene que devolver también un puntero a void. Las funciones imprimen un mensaje cada una, pero lo hacen letra a letra. Además, tras cada impresión se bloquean durante un tiempo usando la función usleep(ms), la cual detiene el hilo que la ejecuta durante ms microsegundos.

En el programa principal tenemos dos variables de tipo **pthread_t** que van a almacenar el identificador de cada uno de los dos hilos que vamos a crear. El identificador de un hilo es necesario guardarlo ya que, una vez que un hilo comienza a funcionar, la única forma de controlarlo es a través de su identificador. El tipo **pthread_t** está definido en la cabecera **pthread.h**, por eso es necesario incluirla al principio del programa.

El lanzamiento o creación de los hilos se realiza con la función pthread create que también está definida en pthread.h. Esta función crea un hilo, inicia la ejecución de la función que se le pasa como tercer argumento dentro de dicho hilo y guarda el identificador del hilo en la variable que se le pasa como primer argumento. Por lo tanto, el primer argumento será la dirección (&) de la variable de tipo pthread_t que queramos que guarde el identificador del hilo creado. El tercer argumento será el nombre de la función que queramos que se ejecute dentro de dicho hilo.

El segundo parámetro se puede utilizar para especificar atributos del hilo, pero si usamos NULL
el hilo se crea con los parámetros normales.