



UNIVERSIDAD NACIONAL

AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ARQUITECTURAS CLIENTE/SERVIDOR

GRUPO: 01

GARCÍA CHÁVEZ LUIS EDGAR

GARCÍA LOERA FEDERICO

MORALES VILCHIS DIANA

SANDOVAL CASTELLANOS JAIR

**HILOS EN LENGUAJE C**

FECHA: 08/09/2009

**Introducción**

En los primeros días de la computación, la programación manejaba procesos en un solo hilo de ejecución. Los programabas se creaban en base a tarjetas perforadas o cintas, por lo cual su lectura era totalmente secuencial. A este tipo de procesamiento se le llamaba por lotes, basado en el hecho de “el primero que llega es el primero en ser atendido”, y cuando era ejecutado el programa, la computadora tenía uso exclusivo para él.

Actualmente esto ha cambiado. El concepto de múltiples hilos de ejecución aparece con los sistemas de tiempo compartido, donde más de una persona podía conectarse a una computadora central a la vez. Con ello nacen los conceptos de proceso(process) e hilo (thread).

Los primeros sistemas operativos funcionaban con un único hilo de ejecución, por ejemplo DOS. Con la creciente demanda actual de aplicaciones gráficas y en red, el uso de los sistemas operativos multiproceso y multihilo se volvieron algo común. Con ello se ha obtenido un mayor uso y rendimiento.

**Definiciones**

* Proceso

El proceso se puede definir como un programa ejecutándose en la computadora, co-existiendo y compartiendo el microprocesador, la memoria y otros recursos del sistema, con otros programas. El proceso es invocado por un código ejecutable, así además, posee una única existencia. Es decir, cada proceso se ejecuta de manera aislada a los otros. Los procesos ejecutan módulos de código, lo cuales pueden ser independientes o no. Por ejemplo, los módulos de código que competen a Word son distintos a los de Power Point, mas no por ello dejan de compartir otros módulos como los DLL’s.

* Hilos de ejecución

Los hilos de ejecución o threads surgieron de la necesidad de que existieran aplicaciones que realizaran varias acciones a la vez, así como una mayor libertad en cuanto a seguir una sola línea de ejecución y realizar varias tareas simultáneas. En situaciones donde algunas acciones podrían causar una demora considerable a un hilo de ejecución (como la espera de que un usuario haga algo), era más deseable que el programa siguiera funcionando, ejecutando otras acciones simultáneas.

**Programación con threads**

Actualmente existen librerías o paquetes de threads que permiten escribir programas con varias líneas de ejecución, sincronizadas a través de memoria compartida. Sin embargo, la programación de hilos supone nuevas dificultades a comparación de la programación secuencial.

Un thread en su concepto más básico, puede ser visto como *un simple flujo de control secuencial*. Con un único punto de ejecución, el programador no necesita aprender nada nuevo para programar un único thread. Sin embargo, cuando se tienen múltipleas hilos significa que el programa tiene distintos puntos de ejecución. Es aquí donde el porgramador deberá decidir dónde y cuándo crear esos threads.

En un lenguaje de alto nivel, las variables globales son compartidas por los diferentes hilos de la aplicación, es decir, los hilos leen y escriben en las mismas localidades de memoria. El programador por tanto, será el responsable de que la sincronización de hilos sea adecuada, y la memoria sea asignada de forma correcta a estos.

Las facilidades proveídas por la librería de hilos, son conocidas como primitiva ligeras, y existen varios tipos:

* Creación
* Mantenimiento
* Sincronización
* Destrucción

**Características de un hilo**

Los hilos pueden ser empleados en situaciones donde se necesitan realizar largos procesamientos, procesamientos en segundo plano o tareas de E/S.

Todos estos ejemplos de uso, tienen una cosa en común: el programa. Esto puede llevar a una sobrecarga de los recursos, que en algunas situaciones sería inaceptable. Sin embargo, hay grandes beneficios, tales como: hacer uso de sistemas multiprocesador y compartir el tiempo con eficiencia. En resumidas cuentas, el uso de hilos convierte a las aplicaciones en una respuesta eficiente y veloz a varios problemas de usanza.

Los hilos comparten ciertas características, entre ellas:

* El espacio de direcciones
* El ID del proceso
* El ID del proceso padre
* El ID del proceso líder del grupo
* Los identificadores de usuario y grupo
* El directorio de trabajo raíz y actual
* La tabla de descriptores de archivos
* El timer de proceso

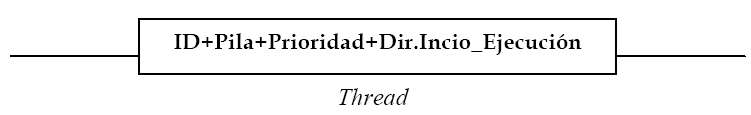
Por otro lado, para identificar a los hilos, poseen ciertas características propias, como:

* Un identificador de hilo único
* La política de planificación y prioridad
* Una variable errno por hilo
* Datos específicos por hilo
* Gestores de cancelación por hilo
* Máscara de señales por hilo

Finalmente, las operaciones que pueden ser llevadas a cabo en un hilo, son:

* Creación y destrucción
* Sincronización entre hilos
* Posibilidad de disponer para cada thread memoria propia
* Gestión de prioridades entre hilos y de señales

Con base en lo anterior, un hilo tiene un identificador, una pila, una prioridad de ejecución y una dirección de inicio de ejecución.



Como ya se ha mencionado, un proceso puede tener sus propios hilos, es decir, los threads viven dentro de un proceso pesado, por tanto, si un thread cambia alguna de las entidades del proceso, el cambio será visto por todos los threads. En la programación, la función main() de algunos lenguajes de programación es un hilo de ejecución. Cualquier otro hilo de ejecución que queramos se deberá crear explícitamente.

**Usos de hilos**

* **Usos en sistemas operativos**

**Trabajo interactivo y en segundo plano**

Por ejemplo, en un programa de hoja de cálculo un hilo puede estar visualizando los menús y leer la entrada del usuario mientras que otro hilo ejecuta las órdenes y actualiza la hoja de cálculo. Esta medida suele aumentar la velocidad que se percibe en la aplicación, permitiendo que el programa pida la orden siguiente antes de terminar la anterior.

**Procesamiento asíncrono**

Los elementos asíncronos de un programa se pueden implementar como hilos. Un ejemplo es como los softwares de procesamiento de texto guardan archivos temporales cuando se está trabajando en dicho programa. Se crea un hilo que tiene como función guardar una copia de respaldo mientras se continúa con la operación de escritura por el usuario sin interferir en la misma.

**Aceleración de la ejecución**

Se pueden ejecutar, por ejemplo, un lote mientras otro hilo lee el lote siguiente de un dispositivo.

**Estructuración modular de los programas**

Puede ser un mecanismo eficiente para un programa que ejecuta una gran variedad de actividades, teniendo las mismas bien separadas mediante a hilos que realizan cada una de ellas.

**Hilos a nivel de usuario (ULT)**

En una aplicación ULT pura, todo el trabajo de gestión de hilos lo realiza la aplicación y el núcleo o kernel no es consciente de la existencia de hilos. Es posible programar una aplicación como multihilo mediante una biblioteca de hilos. La misma contiene el código para crear y destruir hilos, intercambiar mensajes y datos entre hilos, para planificar la ejecución de hilos y para salvar y restaurar el contexto de los hilos.

Todas las operaciones descritas se llevan a cabo en el espacio de usuario de un mismo proceso. El kernel continua planificando el proceso como una unidad y asignándole un único estado (Listo, bloqueado, etc.).

**Hilos a nivel de núcleo (KLT)**

En una aplicación KLT pura, todo el trabajo de gestión de hilos lo realiza el kernel. En el área de la aplicación no hay código de gestión de hilos, únicamente un API (interfaz de programas de aplicación) para la gestión de hilos en el núcleo. Windows 2000, Linux y OS/2 utilizan este método. Linux utiliza un método muy particular en que no hace diferencia entre procesos e hilos, para linux si varios procesos creados con la llamada al sistema "clone" comparten el mismo espacio de direcciones virtuales el sistema operativo los trata como hilos y lógicamente son manejados por el kernel.

* **Usos en general**

Un servidor que debe bloquearse en forma ocasional en espera de acceso al disco, si el servidor tiene varios hilos de control, se podría ejecutar un segundo hilo mientras el primero duerme. El resultado neto sería mejor rendimiento y desempeño.

Cada hilo se ejecuta en forma secuencial y tiene su contar de programa y una pila para llevar un registro de su posición.

Los hilos comparten el CPU de la misma forma que lo hacen los procesos, primero se ejecuta un hilo y después otro (tiempo compartido). Solo en un multiprocesador se pueden ejecutar en realidad en paralelo.

Los distintos hilos de un proceso no son tan independientes como los procesos distintos. Todos los hilos tienen el mismo espacio de direcciones lo que quiere decir que comparten también las mismas variables globales. Puesto que cada hilo puede tener acceso a cada dirección virtual, un hilo puede leer, escribir o limpiar de manera completa la pila de otro hilo.

No existe protección entre los hilos debido a que es imposible y no debe ser necesaria.

Además de compartir un espacio de direcciones todos los hilos comparte el mismo conjunto de archivos abiertos, procesos hijo, cronómetros, señales, etc.

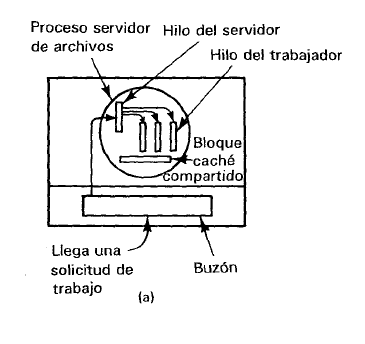
Los hilos pueden tener uno de los siguientes estados: en ejecución, bloqueado, listo o terminado. Un hilo en ejecución posee CPU y esta activo, un hilo bloqueado espera que otro elimine el bloqueo, un hilo listo está programado para su ejecución la cual se llevara a cabo tan pronto le llegue su turno, un hilo terminado es aquel que ha hecho su salida pero que todavía no es recogido por su padre.

**Tres organizaciones de hilos en un proceso**

Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamadas al sistema.

Por ejemplo un servidor de archivos.

**Modelo del servidor/trabajador**



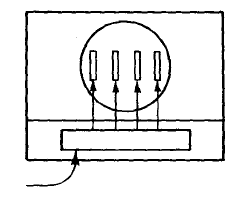
El servidor lee las solicitudes de trabajo del buzón del sistema. Después de examinar la solicitud elije un hilo inactivo (bloqueado) y le envía la solicitud.

Cuando el hilo se activa verifica si puede satisfacer la solicitud por medio del bloque cache compartido, al que tienen acceso todos los hilos. Si no puede, envía un mensaje al disco para obtener el bloque necesario

Los hilos mantienen la idea de procesos secuenciales que hacen llamadas al sistema con bloqueo y aun así logran un paralelismo. Este tipo de llamadas al sistema facilita la programación y el paralelismo mejora el desempeño. El servidor de un hilo mantiene el uso de estas llamadas al sistema pero no aumenta el desempeño.

**Modelo de equipo**

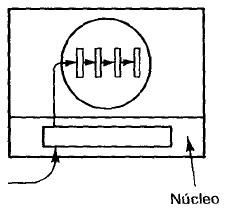
Otra forma de organizar un proceso de muchos hilos



Aquí todos los hilos son iguales y cada uno obtiene y procesa sus propias solicitudes. No hay servidor. A veces llega trabajo que un hilo no puede manejar en especial si cada hilo se especializa en manejar cierto tipo de trabajo. En este caso se puede realizar una cola de trabajo, la cual contiene todos los trabajos pendientes. Con este tipo de organización un hilo debe verificar primero la cola de trabajo antes de buscar en el buzón del sistema

**Entubamiento**

Los hilos también se pueden organizar mediante en modelo de entubamiento.



En este modelo el primer hilo genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan de hilo en hilo y en cada etapa se lleva a cabo cierto procesamiento. Esta puede ser una buena opción en ciertos problemas, como el de los productores y consumidores, aunque no es adecuado para servidores de archivos. Los entubamientos se utilizan con amplitud en muchas aéreas de los sistemas de cómputo, desde la estructura interna de los CPU RISC hasta las líneas de comandos UNIX

**Otros usos**

Con frecuencia los hilos también son útiles para los clientes. Por ejemplo si un cliente desea copiar un archivo en varios servidores, puede hacer que un hilo se comunique con cada servidor.

Otro uso de los hilos cliente es el manejo de señales, como las interrupciones desde el teclado (DEL o BREAK). En vez de dejar que la señal interrumpa al proceso, se dedica un hilo de tiempo completo para la espera de señales.

Por lo general este se bloquea, pero cuando llega una señal, despierta y la procesa. Así el uso de hilos puede eliminar las interrupciones a nivel usuario.

En un sistema multiprocesador es posible que los hilos de un espacio de direcciones se ejecuten en realidad en paralelo, en varios CPU. De hecho, esta es una de las formas principales para compartir los objetos en tales sistemas.

**POSIX.1c**

Existen 8 tipos de datos en POSIX.1c:

|  |  |
| --- | --- |
| **Tipo de dato** | **Descripción** |
| pthread\_attr\_t | Atributo de hilo |
| pthread\_mutexattr\_t | Atributo de mutex |
| pthread\_condattr\_t | Atributo de variable de condición |
| pthread\_mutex\_t | Mutex (bloqueo con exclusión mutua) |
| pthread\_cond\_t | Variable de condición |
| pthread\_t | Hilo (identificador de hilo o ID) |
| pthread\_once\_t | Ejecución una sola vez |
| pthread\_key\_t | Clave sobre datos específicos de hilo |

**Gestión de hilos**

Las funciones básicas para la gestión de hilos son las siguientes:

|  |  |
| --- | --- |
| **Función** | **Descripción** |
| pthread\_create | Crea un hilo |
| pthread\_equal | Verifica la igualdad de dos identificadores de hilo |
| pthread\_exit | Termina el hilo que realiza la llamada |
| pthread\_join | Espera por el termino de un hilo especifico |
| pthread\_self | Regresa el ID del hilo que realiza la llamada |
| pthread\_detach | Configura la liberación de recursos cuando termina (hilo independiente) |
| pthread\_getschedparam | Obtiene la política de planificación y parámetros de un hilo específico |
| pthread\_setschedparam | Establece la política de planificación y parámetros de un hilo específico |
| pthread\_kill | Envía una señal de determinada a un hilo específico |
| pthread\_cancel | Permite a un hilo cancelar otro hilo del mismo proceso |

**Creando un hilo (pthread\_create)**

Esta función automáticamente pone en ejecución el hilo que crea. La sintaxis es la siguiente:

int pthread\_create( pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void \*arg);

Donde:

* El parámetro **thread** apunta al ID del hilo recientemente creado.
* El parámetro **attr** representa un objeto atributo que encapsula los atributos de un hilo. Si attr es NULL, el hilo nuevo tendrá los atributos asignados por defecto.
* El tercer parámetro, **start\_routine**, es el nombre de la función que es invocada por el hilo cuando comienza su ejecución.
* El parámetro **arg** especifica el parámetro que recibe la función start\_routine

Si pthread\_create se ejecuta satisfactoriamente retorna 0. Si la rutina no se ejecuta satisfactoriamente retorna un código de error diferente de cero. A continuación se muestra una tabla con los errores que puede generar una invocación a pthread.

|  |  |
| --- | --- |
| **Error** | **Causa** |
| EAGAIN | El sistema no posee los recursos necesarios para crear el nuevo hilo, o se excede el límite total del número de hilos permitidos por el sistema. |
| EINVAL | El parámetro attr es inválido |
| EPERM | No se tienen los permisos suficientes para cambiar la política de planificación o cualquier otro parámetro especificado en attr |

* **Sobre el objeto atributo**

Los atributos se almacenan en un objeto atributo de tipo pthread\_attr\_t.

Para la creación y el inicio de un objeto atributo (utilizado en la creación de un proceso ligero) se utiliza la siguiente función:

int pthread\_attr\_init(pthread\_attr\_t \*attr);

Para la destrucción del objeto de tipo atributo se utiliza:

int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

El servicio para el establecimiento (set) del atributo correspondiente al estado de terminación es:

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

El valor del argumento detachstate puede ser:

* PTHREAD\_CREATE\_DETACHED: El proceso ligero que se cree con este estado de terminación se considerará independiente y liberará sus recursos cuando finalice su ejecución.
* PTHREAD\_CREATE\_JOINABLE: El proceso ligero que se cree con este estado de terminación se considerará como no independiente y no liberará sus recursos cuando finalice su ejecución. En este caso, es necesario que otro proceso espere por su finalización utilizando pthread\_join.

El servicio para la obtención (get) del atributo correspondiente al estado de terminación es el siguiente:

int pthread\_attr\_getdetachstate(pthread\_attr\_t \*attr, int \*detachstate);

Cada hilo tiene una pila cuyo tamaño se puede establecer en el momento de la creación, dándole al atributo que se va a utilizar en la creación el valor adecuado. Esto se hace mediante el servicio:

int pthread\_attr\_setstacksize (pthread\_attr\_t \*attr, int stacksize);

El servicio para obtener el tamaño de la pila es el siguiente:

int pthread\_attr\_getstacksize (pthread\_attr\_t \*attr, int \*stacksize);

**Comparando IDs (pthread\_equal)**

El ID de un hilo es  dependiente del SO y puede ser una estructura, por esta razón se deberá utilizar una función para comparar la igualdad de los IDs de los hilos

int pthread\_equal(pthread\_t t1, pthread\_t t2);

Donde t1 y t2 son IDs de los hilos que van a ser comparados.

Si t1 es igual a t2, pthread\_equal retorna un valor diferente de cero. Si los IDs de los hilos son diferentes, pthread\_equal retorna 0.

**Salir (pthread\_exit)**

Un hilo puede terminar de tres maneras sin terminar el proceso: retornando de su rutina de inicio, cancelado por otro hilo del mismo proceso, o llamando pthread\_exit.

void pthread\_exit(void \*value\_ptr);

El valor de value\_ptr debe apuntar a datos que existan inclusive después que el hilo termine.

**Unir (pthread\_join)**

La función pthread\_join suspende la ejecución del hilo que la invoca hasta que el hilo objetivo, especificado por el primer parámetro, termine.

int pthread\_join(pthread\_t, void \*\*value\_ptr);

El parámetro **valor\_ptr** es un apuntador al valor de retorno que el hilo objetivo pasa a la función pthread\_exit o a return. Si value\_ptr es NULL, el hilo que invoca join no recibe el valor de retorno del hilo objetivo.

Si la función termina exitosamente retorna 0. Si no termina exitosamente retorna un valor de error diferente de cero.

|  |  |
| --- | --- |
| **Error** | **Causa** |
| EINVAL | El hilo no corresponde a un hilo disponible para la unión |
| ESRCH | No existe un hilo identificado por ID |

**Referencias a hilos por su id (pthread\_self)**

Cualquier hilo puede conocer su ID invocando a:

pthread\_t pthread\_self(void);

Esta función retorna el ID del hilo que la invoca.

**Separar (pthread\_detach)**

Cuando un hilo termina no libera sus recursos al menos que el hilo este separado. La función pthread\_detach modifica las opciones internas del hilo para especificar que el espacio utilizado para el almacenamiento del mismo puede ser reclamado cuando culmine. Los hilos ya separados no reportan su estado cuando culminan.

int pthread\_detach( pthread\_t thread );

La función pthread\_detach recibe un solo parámetro, **thread**, que es el ID del hilo a ser separado.

Si pthread\_detach culmina exitosamente retorna uno. Si no culmina exitosamente retorna un valor diferente de cero. La siguiente tabla muestra los códigos de error de pthread\_detach.

|  |  |
| --- | --- |
| **Error** | **Causa** |
| EINVAL | El hilo no corresponde a un hilo disponible para la unión |
| ESRCH | No existe un hilo identificado por ID |

**Cancelar (pthread\_cancel)**

Un hilo invoca pthread\_cancel para solicitar que otro hilo sea cancelado. El resultado de la invocación es determinado por el tipo del hilo objetivo y su estado de cancelación.

int pthread\_cancel(pthread\_t thread);

El parámetro de pthread\_cancel es el ID del hilo objetivo a ser cancelado.

Si pthread\_cancel se ejecuta exitosamente retorna 0. Sino retorna un valor diferente de cero.

**Mutex para problemas de concurrencia**

La concurrencia es la propiedad de los sistemas que permiten que múltiples [procesos](http://www.alegsa.com.ar/Dic/proceso.php) sean [ejecutados](http://www.alegsa.com.ar/Dic/ejecutar.php) al mismo tiempo, y que potencialmente puedan interactuar entre sí.

Trabajar con hilos es trabajar nativamente con programas concurrentes, uno de los mayores problemas con los que nos podremos encontrar, y que es tácito en la concurrencia, es el acceso a variables y/o estructuras compartidas o globales, es decir usar variables que son modificadas por otros hilos.

Existen regiones críticas (RC) que son parte de código susceptible de verse afectada por la concurrencia.

Para solucionar el problema los hilos POSIX nos ofrecen los semáforos binarios, semáforos mutex o simplemente mutexs. Un semáforo binario es una estructura de datos que actúa como un semáforo porque puede tener dos estados: abierto o cerrado. Cuando el semáforo está abierto, al primer hilo que pide un bloqueo se le asigna ese bloqueo y no se deja pasar a nadie más por el semáforo. Mientras que si el semáforo está cerrado, porque algún hilo ya tiene el bloqueo, el thread que lo pidió parará su ejecución hasta que no sea liberado el susodicho bloqueo. Solo puede haber un solo hilo teniendo el bloqueo del semáforo, mientras que puede haber más de un hilo esperando para entrar en la RC, situados en la cola de espera del semáforo. Es decir, los threads se excluyen mutuamente (de ahí lo de mutex para el nombre) el uno al otro para entrar.

Las principales funciones para el uso de mutex son las siguientes:

|  |  |
| --- | --- |
| **Función** | **Descripción** |
| pthread\_mutex\_init | Inicializa un mutex con los atributos especificados |
| pthread\_mutex\_destroy | Destruye el mutex especificado |
| pthread\_mutex\_lock | Permite solicitar acceso al mutex, el hilo se bloquea hasta su obtención |
| pthread\_mutex\_trylock | Permite solicitar acceso al mutex,  el hilo retorna inmediatamente. El valor retornado indica si otro hilo lo tiene. |
| pthread\_mutex\_unlock | Permute liberar un mutex. |
| pthread\_mutex\_getprioceiling | Obtiene el valor de prioridad límite (prioceiling) del mutex especificado. |
| pthread\_mutex\_setprioceiling | Establece el valor de prioridad límite y devuelve el anterior valor (old-ceiling) en el mutex especificado. |

**Inicialización de mutex**

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr)

**mutex**: Es un puntero a un parámetro del tipo pthread\_mutex\_t, que es el tipo de datos que usa la librería Pthreads para controlar los mutex.

**attr**: Es un puntero a una estructura del tipo pthread\_mutexattr\_t y sirve para definir qué tipo de mutex queremos: normal, recursivo o errorcheck. Si este valor es NULL (recomendado), la librería le asignará un valor por defecto.

La función devuelve 0 si se pudo crear el mutex o -1 si hubo algún error. Esta función debe ser llamada antes de usar cualquiera de las funciones que trabajan con mutex.

**Petición de bloqueo**

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

**mutex:** Es un puntero al mutex sobre el cual queremos pedir el bloqueo o sobre el que nos bloquearemos en caso de que ya haya alguien dentro de la RC.

Esta función pide el bloqueo para entrar en una RC. Si queremos implementar una RC, todos los thread tendrán que pedir el bloqueo sobre el mismo semáforo. Como resultado, devuelve 0 si no hubo error, o diferente de 0 si lo hubo.

**Liberación de bloqueo**

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

**mutex**: Es el semáforo donde tenemos el bloqueo y queremos liberarlo.

Esta función libera el bloqueo que tuviéramos sobre un semáforo. Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo.

**Destrucción de mutex**

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex)

**mutex:** El mutex que queremos destruir.

Esta función le dice a la librería que el mutex que el estamos indicando no lo vamos a usar más, y que puede liberar toda la memoria ocupada en sus estructuras internas por ese mutex. La función devuelve 0 si no hubo error, o distinto de 0 si lo hubo.

**Programas con uso de hilos**

Como en nivel y la rapidez de procesamiento de las computadoras actuales es mas rápido que en épocas anteriores, la ejecución de un hilo es difícil de visualizar, pues prácticamente se realiza de una forma transparente para el usuario.

En esta parte se explicaran ciertos programas que hacen uso de hilos, cada uno realizando cierta acción para ver de una manera más fácil la ejecución de los hilos.

**Creación de hilos**

Este programa es muy sencillo ya que realiza la creación de dos hilos, los cuales imprimen la dirección en la cual se encuentran creado el hilo.

Lo primero que se realiza es llamar a la librería en donde se encuentran ubicadas las funciones para el uso de los hilos.

#include <pthread.h>

#include <stdio.h>

Lo siguiente es crear una función que será llamada posteriormente al crearse el hilo, esta función lo único que realiza es la impresión de la dirección del hilo sin recibir ningún parámetro. Para obtener la dirección del hilo se utiliza la función pthread\_self()

void func(void)

{

printf("Thread %x \n", pthread\_self()); //Id del hilo creado

pthread\_exit(NULL);

}

Después se crea la función main, en la cual se crean los hilos a utilizar y las variables donde estarán las direcciones de cada hilo.

int main()

{

pthread\_t th1, th2; //Direcciones donde estarán los hilos

//Se crean dos hilos sin parametros y con atributos por defectos

pthread\_create(&th1, NULL, (void \*)func, NULL);

pthread\_create(&th2, NULL, (void \*)func, NULL);

printf("El proceso ligero principal continua ejecutando\n");

//Se espera a que terminen los hilos

pthread\_join(th1, NULL);

pthread\_join(th2, NULL);

printf("Se terminaron de ejecutar los hilos\n");

return(0);

}

Lo que hace pthread\_create(&th1, NULL, (void \*)func, NULL); es crear un hilo tomando cierto parámetros para inicializarlo.

&th1 🡪 Contiene la dirección del hilo creado

NULL 🡪 Inicializa por defecto los atributos del hilo

(void \*)func 🡪 Llama a la función a ejecutar, necesita un casteo porque recibe como parámetro un puntero

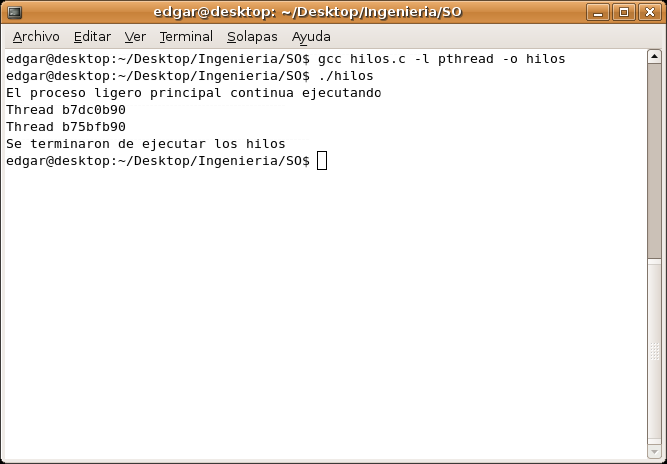
NULL 🡪 Argumentos de la función, en este caso no se necesita ninguno.

pthread\_join(th1, NULL); Espera a la terminación del hilo

th1 🡪 Identificador del hilo, espera a que termine de ejecutarse

NULL 🡪 Regresa un valor cuando ha terminado el hilo.

Ejecutando este programa lo que se obtendría seria:



**Crear varios hilos**

En este programa se crean varios hilos a la vez y al mismo tiempo comienza su ejecución.

En primer lugar se llaman las librerías a utilizar y se define una variable estática, la cual contendrá el número de hilos a crear durante el programa.

#include "pthread.h"

#include "stdio.h"

#include "stdlib.h"

#define NUM\_THREADS 5

Después se implementa la función que realizaran todos los hilos, esta función realiza la impresión del hilo que se encuentra ejecutando en ese momento. Recibe como parámetro el número de hilo, pero como un apuntador, por lo que se hace un cast para obtener su valor original.

// La funcion que es utilizada en un Hilo

// debera de retornar un puntero sin tipo

void \*imprimeSaludo(void \*threadid){

long tid;

// Aqui se castea lo que le fue pasado

// al hilo como atributos

tid = (long)threadid;

printf("Se esta ejecutando el hilo %ld \n", tid);

pthread\_exit(NULL); //Se finaliza la ejecucion del Hilo

}

Por ultimo dentro del main, se crea la variable que creara los hilos, tomando como longitud el número de la variable especificada anteriormente, creando cada hilo en un for y llamando a su ejecución al mismo tiempo.

int main (int argc, char \*argv[])

{

pthread\_t threads[NUM\_THREADS]; //Direccion donde se almacenaran los hilos (5 en total)

int rc;

int t;

for(t=0; t < NUM\_THREADS; t++){

printf("Creando Hilo %ld \n", t);

rc = pthread\_create(&threads[t], NULL, imprimeSaludo, (void \*)t);

if (rc){

printf("ERROR; return code from pthread\_create() is %d \n", rc);

exit(-1);

}

//pthread\_join(threads[t],NULL); //Se espera a que finalice el hilo

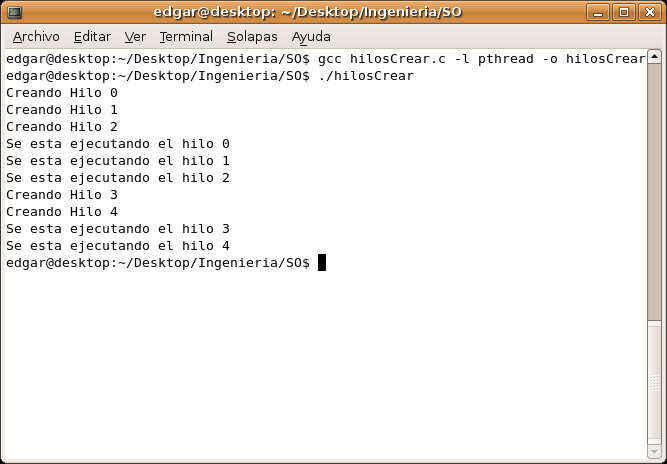
}

pthread\_exit(NULL); //Se finaliza la ejecucion del Hilo

}

En la función rc = pthread\_create(&threads[t], NULL, imprimeSaludo, (void \*); se utiliza una variable de tipo entero, la cual obtendrá un valor que determinara si el hilo fue creado correctamente en caso de que devuelva un 1, el hilo no creo bien y pasara a informar del error.

Los parámetros que recibe son los mismos que se mencionaron anteriormente, con una excepción, que al llamar a la función que ejecutara cada hilo, ya no se necesito de un cast a puntero, puesto que la función fue definida como tipo puntero desde su creación.



**Hilos que realizan operaciones**

Este programa crea dos hilos, los cuales realizaran una operación distinta cada uno, arrojando el resultado y la dirección del hilo.

Se definen que librerías serán utilizadas por el programa, en este caso se agrega la matemática la cual proveerá las funciones que se realizaran posteriormente.

#include<pthread.h>

#include<stdio.h>

#include<math.h>

Después se procede a crear la función de cada hilo, una que realice la división y otra la potencia.

void potencia(int \*argv){

int z;

printf("Thread %x \n",pthread\_self()); //Se imprime el Id. del Hilo creado

z=pow((argv[1]),(argv[2]));

printf("Potencia %d\n",z);

pthread\_exit(NULL); //Finaliza la ejecucion del hilo

}

void division(int \*argv){

float w;

printf("Thread %x\n",pthread\_self());

if(argv[2]==0)

printf("No se puede realizar operacion\n"); //Se imprime el Id. del Hilo creado

else{

w=((float)(argv[1])) / ((float)(argv[2]));

printf("Divison: %.4f\n",w);

}

pthread\_exit(NULL); //Finaliza la ejecucion del hilo

}

Cada función recibe como parámetro un apuntador a un arreglo que se define en la función main, cada valor dentro del puntero contiene el número con el que se realizara la operación en cada función.

Dentro del main se crea cada hilo, sin embargo toma los números para realizar las operación por argumentos al ejecutar el programa, por lo que se almacenan en una arreglo.

int main(int argc, char \*argv[]){

int i,h[3],j[3];

pthread\_t th1,th2; //Direccion donde se almacenaran los identificadores de los hilos

if(argc <= 1){

printf("DEBES INTRODUCIR DOS NUMEROS");

printf("\nEjemplo: \n./hilos 5 23\n");

}

else{

for(i=1;i<argc;i++)

h[i]=atoi(argv[i]);

pthread\_create(&th1,NULL,(void\*)potencia,&h);

//Se crea el Hilo para ejecutar la funcion Potencia

pthread\_create(&th2,NULL,(void\*)division,&h);

//Se crea el Hilo para ejecutar la funcion Division

pthread\_join(th1,NULL); //Se espera a que finalice el hilo

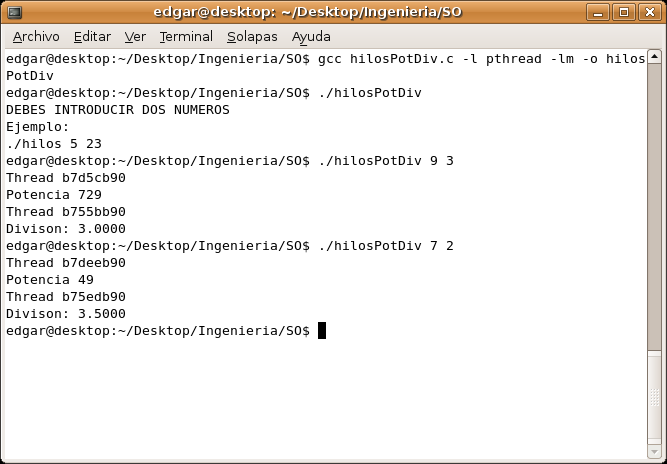
pthread\_join(th2,NULL); //Se espera a que finalice el hilo

}

return(0);

}

Cuando cada hilo ha finalizado, se sale del programa. Lo que arroja como resultado lo siguiente:



**Fuentes**

* http://profesores.elo.utfsm.cl/~agv/elo330/2s08/lectures/POSIX\_Threads.html
* http://profesores.elo.utfsm.cl/~agv/elo330/2s06/lectures/POSIX\_threads/POSIX\_Threads\_Synchronization.html
* http://www.cs.buap.mx/~rossainz/Primavera07/ProgConcPar/apuntes/Cap2\_Conc\_PosixThreads2.pdf
* www.geocities.com/linuxhgb/publicaciones/ThreadsHGB.pdf
* ares.unimet.edu.ve/programacion/bppr12/c9lases/pthreads.doc