Hilos POSIX

Introducción

El siguiente documento tiene como propósito dar una breve introducción al lector sobre hilos POSIX, abarcando diversos temas como conceptos básicos para su comprensión, orígenes, características, entre otros. Al final del documento se encuentran cinco ejercicios para medir el nivel de comprensión de los temas. Se da por entendido que el lector posee conocimientos intermedios o avanzados del lenguaje C.

Definición de conceptos

A continuación se describen algunos conceptos importantes:

- Hilo

Flujo independiente de instrucciones que puede programarse para ser ejecutado por el sistema operativo. Unidad mínima de procesamiento cuya ejecución puede ser programada por un sistema operativo, también se le llama proceso ligero (lightweight process).

- Asincronía/Asincronismo

Ocurrencia de eventos de forma independiente.

- Concurrencia

Describe el comportamiento de tareas en un sistema con un solo procesador. Las cosas parecen ocurrir al mismo tiempo, pero en realidad pueden ocurrir de forma secuencial. Algunos autores la describen como la *ilusión del paralelismo*.

- Monoprocesador

En términos generales, se refiere a una computadora con una sola unidad de ejecución visible por el programador.

- Multiprocesador

Computadora con dos o más procesadores que comparten un conjunto de instrucciones y acceden al mismo espacio de memoria. No es un requerimiento que todos los procesadores tengan el mismo acceso a toda la memoria física.

- Paralelismo

Describe secuencias concurrentes que proceden de forma simultánea. En otras palabras, es la ocurrencia de dos o más cálculos al mismo tiempo. El paralelismo verdadero solo puede ocurrir en sistemas multiprocesador, mientras que la concurrencia puede ocurrir tanto en sistemas monoprocesador como multiprocesador. Requiere que se ejecuten dos cálculos de manera simultánea, mientras que la concurrencia solo requiere que el programador sea capaz de pretender que dos cosas sucedan al mismo tiempo.

- Overhead

Ampliamente, se define como el tiempo adicional de cómputo indirecto necesario para llevar a cabo una ejecución; por ejemplo, la llamada a funciones en un programa.

Orígenes

POSIX es el acrónimo de Portable Operating System Interface. Históricamente, cada fabricante de hardware implementaba sus propios hilos y la forma de gestionarlos. Las implementaciones variaban notablemente, lo que dificultaba a los programadores desarrollar aplicaciones con hilos portables. Para aprovechar al máximo las bondades que ofrecen los hilos era necesario hacer una interfaz de programación estandarizada.

En 1995 se especificó la interfaz estándar IEEE 1003.1c. Las implementaciones basadas en este estándar son conocidas como hilos POSIX o Pthreads. Actualmente la mayoría de los fabricantes ofrecen

en su hardware Pthreads, además de sus propias APIs (Intel, AMD). A pesar que se creó en 1995, este estándar ha pasado por varios cambios y revisiones, la más reciente fue en el año 2008.

Características

Una de las características más importantes de los hilos es que permiten la ejecución concurrente de instrucciones asociadas a diferentes funciones dentro de un mismo proceso.

Los hilos de un mismo proceso comparten los siguientes elementos:

- Código
- Variables de memoria globales
- Archivos o dispositivos (entrada y salida estándar) que tuviera abiertos el hilo padre

Por otro lado, *no* comparten los siguientes elementos:

- Contador de programa
- Registros del CPU
- Pila
- Estado del hilo

Una de las bondades de los hilos es que consumen menos memoria que los procesos puesto que comparten un mismo espacio de direcciones, lo que resulta un bajo costo al crearlos. Al compartir variables globales ya tienen integrado un mecanismo de comunicación. Además resulta más fácil hacer cambios de contexto entre hilos debido a que producen menos overhead, por ello son herramientas muy útiles para llevar a cabo tareas concurrentes cooperativas.

El contexto de una tarea, ya sea un proceso o un hilo, consiste en el conjunto de información mínima utilizada por la tarea que se requiere almacenar para permitir su interrupción y posterior continuación. Cuando se diseña un sistema operativo, se busca optimizar los cambios de contexto de las tareas para un mejor desempeño de las aplicaciones o programas. Un cambio de contexto consiste en el procedimiento de almacenar y restaurar el estado del CPU con el objetivo de que la ejecución pueda retomarse desde el mismo punto después de cierto tiempo para atender interrupciones u otros procesos de acuerdo al planificador del sistema operativo.

Estados de un hilo

La siguiente imagen muestra un diagrama sencillo del clico de vida de un hilo POSIX:

(Imagen)

Por otra parte, los estados de un hilo son los siguientes:

1. Listo.- El hilo ya puede ser ejecutado, pero está esperando a un procesador.

2. Ejecutándose/Corriendo.- El hilo se está ejecutando; en sistemas multiprocesador pueden estar ejecutándose varios de forma simultánea.

3. Bloqueado.- El hilo no puede ejecutarse porque está esperando algo, por ejemplo variables de condición o una operación de E/S para completarse.

4. Terminado.- El hilo termina cuando se llamó a la función *pthread_exit*[1] o fue cancelado.

Es importante tener en cuenta que no hay sincronización cuando un hilo crea a otro hilo, es decir, el hilo creado con *pthread_create*[2] puede comenzar a ejecutarse e incluso acabar antes de que se regrese de la función.

[1] v. p. 6
[2] *Ibídem*

El hilo principal difiere de los hilos que crea en que:

1. El hilo recibe explícitamente dos parámetros argc y argv, no un apuntador a void

2. Cuando un hilo creado a partir del hilo principal termina, otros hilos pueden seguir ejecutándose. Cuando el hilo principal termina, todos los hilos creados a partir de él terminan.

3. El hilo principal trabaja en la pila por default del proceso, que puede llegar a crecer considerablemente, mientras que las pilas de los hilos creados a partir del principal son de menor tamaño. Debido a que la utilización de hilos no es una tarea estándar de UNIX, al compilar un programa que utilice hilos POSIX es necesario usar la opción *-pthread*.

Es importante mencionar que se debe de saber lo que se está haciendo y tener cuidado al programar usando hilos. Los hilos no siempre aportan la mejor solución a todos los problemas de programación, no siempre son fáciles de usar y no siempre mejoran el desempeño de un programa, por ejemplo, si la mayoría del programa depende directamente de los resultados de pasos anteriores usar hilos probablemente no ayudará. La utilización de hilos es recomendable en los siguientes casos:

1. Realizar cálculos extensos que se pueden paralelizar o descomponer en hilos y que están diseñados para correr en sistemas multiprocesador.

2. Cuando se esperan múltiples señales de E/S: varios hilos pueden esperar diferentes señales, por ejemplo, aplicaciones de servidores distribuidos donde se trabaja con múltiples clientes de forma simultánea.

Diferencias entre hilos y procesos

La diferencia típica es que los hilos se ejecutan en un espacio de memoria compartido, mientras que los procesos no. Los hilos comparten el espacio de direcciones del proceso que los creó. Cada proceso tiene su propio espacio de direcciones. Otra diferencia es que generalmente los procesos son independientes

entre sí, mientras que los hilos se consideran un subconjunto de un proceso. Es necesario comprender que tanto los hilos como los procesos son secuencias independientes de ejecución.

- Un mismo proceso puede tener varios hilos ejecutando diferentes funciones de dicho proceso.
- Un hilo *no* contiene un proceso.
- El estándar POSIX ve a los hilos como un conjunto de hilos iguales, no existe una jerarquía como tal, mientras que los procesos sí la tienen bien definida.
- Los procesadores ejecutan hilos, no procesos

Creación de multitareas/multihilos

A continuación se describen algunas de las funciones más comunes para utilizar hilos POSIX:

- *pthread_create* crea un nuevo hilo con los atributos attr dentro del proceso que la invoca y devuelve a través de thread su identificador. Si attr vale NULL se pasan los atributos por defecto. En caso de fallo, la función devuelve un número distinto de cero que indica que error producido. El hilo ejecuta el código de la rutina referenciada por start_routine a la que se pasan los argumentos referenciados por arg.
- *pthread_self* devuelve el identificador del hilo que la invoca
- *pthread_detach* indica al sistema que los recursos de almacenamiento ocupados por el hilo con identificador thread tienen que ser liberados cuando el hilo termine. La llamada a esta función no produce la terminación del hilo; no se recomienda llamarla más de una vez por cada hilo, de lo contrario el comportamiento es indefinido
- *pthread_exit* termina la ejecución del hilo que la invoca y le devuelve los datos referenciados por value_ptr al hilo que esté esperando su terminación. Esta función es también llamada de forma implícita cuando termina la rutina start_routine indicada al crear el hilo

- *pthread_join* suspende la ejecución del hilo que la invoca y espera hasta que la terminación del hilo con identificador thread. El apuntador devuelto por este hilo con una llamada pthread_exit es almacenado en la dirección referenciada por value_ptr. Si un hilo intenta ejecutar pthread_join sobre sí mismo se detecta un abrazo mortal y la llamada a la función falla devolviendo el valor EDEADLK
    - El segundo argumento es un apuntador a apuntador porque nos interesa el valor de dicho argumento, si no fuera así se obtiene una copia de la dirección de la variable
- *pthread_equal* devuelve 0 si los identificadores *t1* y *t2* son iguales y cualquier otro valor diferente de 0 si coinciden.

A continuación se muestran los prototipos de las funciones:

```
pthread_t thread;
int pthread_equal (pthread_t t1, pthread_t t2);
int pthread_create (pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start)(void *), void *arg);
pthread_t pthread_self (void);
int sched_yield (void);
int pthread_exit (void *value_ptr);
int pthread_detach (pthread_t thread);
int pthread_join (pthread_t thread, void **value_ptr);
```

Atributos de un hilo

Los atributos de un hilo son los siguientes:

- Vinculación (detach_state)
- Tamaño de la pila (stacksize)
- Dirección de la pila (stackaddr)
- Ámbito de contienda (scope)

- Herencia de la planificación (inherisched)
- Política de distribución (schedpolicy)
- Parámetros de planificación (schedparam)

Estos atributos se almacenan en objetos tipo *pthread_attr_t*, son inicializados con los valores por defecto llamando a la función *pthread_attr_init* y borrados con *pthread_attr_destroy*. A continuación se muestra una imagen con los prototipos de las funciones para modificar los atributos de un hilo:

```
pthread_attr_t attr;
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
int pthread_attr_getdetachstate (
    pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate (
    pthread_attr_t *attr, int detachstate);
#ifdef _POSIX_THREAD_ATTR_STACKSIZE
int pthread_attr_getstacksize (
    pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstacksize (
    pthread_attr_t *attr, size_t stacksize);
#endif
#ifdef _POSIX_THREAD_ATTR_STACKADDR
int pthread_attr_getstackaddr (
    pthread_attr_t *attr, void *stackaddr);
int pthread_attr_setstackaddr (
    pthread_attr_t *attr, void **stackaddr);
#endif
```

En el estándar ANSI, *size_t* es un tipo de variable que representa el tamaño máximo de cualquier objeto. La conocida función *sizeof* regresa un dato de este tipo. Junto con el tipo *ptrdiff_t*, describe cantidades relativas a la memoria, y fueron implementados porque el tamaño de los tipos de datos comunes está definido de acuerdo a las capacidades aritméticas del procesador que se está utilizando, y no a las capacidades de la memoria del sistema (como el espacio de direcciones disponible). A manera de ejemplo, el siguiente programa muestra cómo modificar el tamaño de la pila de un hilo

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

void *codigo_del_hilo(void *arg){
	pthread_attr_t *atributos = arg;
	int detachstate;
	size_t tam_pila;
	int error;

	error = pthread_attr_getdetachstate(atributos, &detachstate);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));
	else if(detachstate == PTHREAD_CREATE_DETACHED)
		printf("Hilo separado.\n");
	else
		printf("Hilo no separado.\n");

	error = pthread_attr_getstacksize(atributos, &tam_pila);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));
	else
		printf("Tamaño de la pila: %zd bytes = %zd x %d\n", tam_pila,
tam_pila/PTHREAD_STACK_MIN, PTHREAD_STACK_MIN);
	return NULL;
}

int main(){
	pthread_t hilo;
	pthread_attr_t atributos;
	size_t tam_pila;
```

```
	int error;

	//Inicialización de atributos
	error = pthread_attr_init(&atributos);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));

	//Activación del atributo PTHREAD_CREATE_DETACHED
	error = pthread_attr_setdetachstate(&atributos,
PTHREAD_CREATE_DETACHED);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));

	//Manipulación del tamaño de la pila
	error = pthread_attr_getstacksize(&atributos, &tam_pila);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));
	else{
		printf("Tamaño de la pila por defecto: %zd bytes\n", tam_pila);
		printf("Tamaño mínimo de la pila: %d bytes\n",
PTHREAD_STACK_MIN);
	}

	error = pthread_attr_setstacksize(&atributos, 3*PTHREAD_STACK_MIN);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));

	//Creación de un hilo con los atributos anteriores
	error = pthread_create(&hilo, &atributos, codigo_del_hilo, &atributos);
	if(error)
		fprintf(stderr, "Error %d: %s\n", error, strerror(error));
```

```
        printf("Fin del hilo principal.\n");
        pthread_exit(NULL);
}
```

Desvincular un hilo no lo afecta en ninguna manera, solo significa que el sistema operativo puede reclamar los recursos de dicho hilo cuando este termine de ejecutarse

- Ejercicio de pasar argumentos

Comunicación y concurrencia de hilos

Debido a que los hilos de un mismo proceso comparten el espacio de memoria, la sincronización del acceso a los recursos compartidos (del mismo proceso) es muy importante. A continuación se describen dos métodos para comunicar hilos:

MUTEX

Proviene del inglés ***mut****ual* ***ex****clusion*. Su función principal consiste en proteger los datos compartidos. La manera más común de sincronizar hilos es asegurarse de que todos los accesos a memoria a los mismos datos (o relacionados a ellos) sean mutuamente excluyentes, es decir, un solo hilo puede acceder a la región crítica a la vez y los demás deben esperar. Un mutex se puede entender como una forma especial de semáforo de Dijkstra, que generalmente se considera que son simples, flexibles y su implementación es eficiente

En un programa se representa con una variable de tipo *pthread_mutex_t*. Nunca se debe de hacer una copia de un mutex, si se hace el comportamiento del programa es indefinido, sin embargo se pueden hacer copias de apuntadores al mismo mutex para sincronizar varios hilos con uno solo. A continuación de muestra un programa en C que utiliza este tipo de variables:

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

/*
 * Definición de una estructura que contiene un mutex
 */
typedef struct my_struct_tag {
    pthread_mutex_t     mutex;  /* Protege el acceso al valor */
    int                 value;  /* El acceso es protegido por el mutex */
} my_struct_t;

int main (int argc, char *argv[])
{
    my_struct_t *data;
    int status;

    data = malloc (sizeof (my_struct_t));
    if (data == NULL)
        errno_abort ("Asignar estructura");
    status = pthread_mutex_init (&data->mutex, NULL);
    if (status != 0)
        err_abort (status, "Inicializar mutex");
    status = pthread_mutex_destroy (&data->mutex);
    if (status != 0)
        err_abort (status, "Destruir mutex");
    (void)free (data);
    return status;
}
```

A continuación se presentan algunas reglas básicas sobre el uso de mutex:

- Un hilo nunca debe de bloquear un mutex que ya tenga bloqueado, se puede producir el error que envía la señal EDEADLK o este quedará esperando eternamente.

- No se puede desbloquear un mutex que esté desbloqueado o que se encuentre bloqueado por otro hilo
- Un mutex bloqueado pertenece en ese momento al hilo que lo bloqueó, existen otros mecanismos de bloqueo que no toman pertenencia como los semáforos. Para ilustrar esto, a continuación se presentan un programa en el que dos hilos utilizan el mismo mutex para acceder a la región crítica:

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

pthread_mutex_t mutex;

void *restar(void *arg)
{
      int i;
      int *x = (int *)arg;
      for (i = 0; i < 9; i++) {
            pthread_mutex_lock(&mutex);
            *x = *x - 1;
            printf("%3d", *x);
            pthread_mutex_unlock(&mutex);
      }
      return NULL;
}

void *sumar(void *arg)
{
      int i;
      int *x = (int *)arg;
      pthread_mutex_lock(&mutex);
      for (i = 0; i < 9; i++) {
```

```
            *x = *x + 1;
            printf("%3d", *x);
      }
      pthread_mutex_unlock(&mutex);
      return NULL;
}

int main()
{
      pthread_t sumar_thread, restar_thread;
      int x = 0;

      pthread_mutex_init(&mutex, NULL);

      // crear dos hilos
      pthread_create(&sumar_thread, NULL, sumar, (void *)&x);
      pthread_create(&restar_thread, NULL, restar, (void *)&x);

      // esperar hasta que terminen de ejecutarse
      pthread_join(sumar_thread, NULL);
      pthread_join(restar_thread, NULL);

      printf("\n");
}
```

Existe otra función llamada *pthread_mutex_trylock* que en vez de bloquear al hilo, le regresa a este la señal EBUSY; es útil cuando este puede ejecutar otras instrucciones en vez de que espere a que se desbloquee el mutex. La única recomendación es desbloquear el mutex si la llamada a dicha función fue exitosa.

VARIABLES DE CONDICIÓN

Consisten en un mecanismo de sincronización de hilos que está asociado con variables de tipo mutex. Una variable de condición se utiliza para comunicar información sobre el estado de datos compartidos entre hilos. Mientras que un mutex implementa la sincronización de los hilos controlando el acceso de ellos a los datos, las variables de condición hacen posible la sincronización a partir del valor actual de los datos, previendo así un mecanismo al programador que garantice una sincronización determinada. Estas variables son útiles para evitar que los hilos revisen constantemente si un dato (muy frecuentemente en una región crítica) ha alcanzado un valor específico.

Se deben tomar en cuenta los siguientes puntos al usar variables de condición:

- Siempre deben de regresar con un mutex bloqueado
- Se utilizan para enviar señales, no para la exclusión mutua
- No se deben de realizar copias de una variable tipo *pthread_cond_t*, el comportamiento de un programa que utiliza una o más copias de una variable de este tipo no está definido. Se recomienda utilizar apuntadores a una misma variable
- Al igual que los mutex, las variables de condición se deben de inicializar

La función *pthread_cond_wait* bloquea al hilo que la llama hasta que la señal especificada (dentro de la función) se cumple. Esta función se debe de llamar con un mutex bloqueado, y de forma automática liberará dicho mutex mientras se espera. Una vez que se despierta a este hilo, el mutex se bloquea automáticamente por este hilo (el que llamó a *pthread_cond_wait*). El programador es responsable de liberar el mutex una vez que el hilo termina.

La rutina *pthread_cond_signal* se usa para avisar (o despertar) al hilo que está esperando que se cumpla la condición deseada. Si se necesita despertar a más de un hilo se utiliza la función *pthread_cond_broadcast*. Es muy importante bloquear y desbloquear los mutex adecuados en los momentos adecuados. Existen funciones para destruir tanto mutex como variables de condición cuando ya no son necesarios.

El siguiente código es similar al programa que utiliza un mutex para acceder a la región crítica, solo que ahora utiliza variables de condición, garantizando que uno de los hilos siempre entre primero a dicha región. Se deja al lector como trabajo determinar qué hilo siempre entra primero a la región crítica.

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

pthread_mutex_t mutex;
pthread_cond_t cond;

void *menos(void *arg)
{
	int i;
	int *x = (int *)arg;
	for (i = 0; i < 9; i++) {
		pthread_mutex_lock(&mutex);
		*x = *x - 1;
		printf("%3d", *x);
		pthread_mutex_unlock(&mutex);
	}
	pthread_cond_signal(&cond);
	return NULL;
}
```

```
void *mas(void *arg)
{
      int i;
      int *x = (int *)arg;
      pthread_mutex_lock(&mutex);
      if (*x == 0)
                  pthread_cond_wait(&cond, &mutex);
      for (i = 0; i < 9; i++) {
            *x = *x + 1;
            printf("%3d", *x);
      }
      pthread_mutex_unlock(&mutex);
      return NULL;
}

int main()
{
      pthread_t hilo_mas, hilo_menos;
      int x = 0;

      pthread_mutex_init(&mutex, NULL);
      pthread_cond_init(&cond, NULL);

      pthread_create(&hilo_mas, NULL, mas, (void *)&x);
      pthread_create(&hilo_menos, NULL, menos, (void *)&x);

      pthread_join(hilo_mas, NULL);
      pthread_join(hilo_menos, NULL);

      printf("\n");
}
```

EJERCICIOS

Indique qué programas imprimirán únicamente el número 42 en la consola explicando claramente porqué sí o no.

Programa 1

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

void *thread(void *vargp){
	pthread_exit((void*)42);
}

int main(){
	int i;
	pthread_t tid;
	pthread_create(&tid, NULL, thread, NULL);
	pthread_join(tid, (void **)&i);
	printf("%d\n",i);
}
```

Programa 2

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

void *thread(void *vargp){
	exit(42);
}

int main(){
	int i;
```

```
	pthread_t tid;
	pthread_create(&tid, NULL, thread, NULL);
	pthread_join(tid, (void **)&i);
	printf("%d\n",i);
}
```

Programa 3

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

void *thread(void *vargp){
	int *ptr = (int*)vargp;
	pthread_exit((void*)*ptr);
}

void *thread2(void *vargp){
	int *ptr = (int*)vargp;
	*ptr = 0;
	pthread_exit((void*)31);
}

int main(){
	int i = 42;
	pthread_t tid, tid2;
	pthread_create(&tid, NULL, thread, (void*)&i);
	pthread_create(&tid2, NULL, thread2, (void*)&i);
	pthread_join(tid, (void**)&i);
	pthread_join(tid2, NULL);
	printf("%d\n",i);
}
```

Programa 4

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

void *thread(void *vargp){
      pthread_detach(pthread_self());
      pthread_exit((void*)42);
}

int main(){
      int i = 0;
      pthread_t tid;
      pthread_create(&tid, NULL, thread, (void*)&i);
      pthread_join(tid, (void**)&i);
      printf("%d\n",i);
}
```

Programa 5

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

int i = 42;

void *thread(void *vargp){
      printf("%d\n",i);
}

void *thread2(void *vargp){
      i = 31;
}
```

```
int main(){
      pthread_t tid, tid2;
      pthread_create(&tid2, NULL, thread2, (void*)&i);
      pthread_create(&tid, NULL, thread, (void*)&i);
      pthread_join(tid, (void**)&i);
      pthread_join(tid2, NULL);
}
```

Bibliografía

BUTHENHOF, David R., "Programming with POSIX Threads", Addison-Wesley Professional Computing Series, 1ª. ed., 2006.