```
void monitor_fd(int fd[], int num_fds)
{
   int i;
   pthread_t *tid;

   if ((tid = (pthread_t *)calloc(num_fds, sizeof(pthread_t))) == NULL)
      return;
                        /* crear un hilo para cada descriptor de archivo */
   for (i = 0; i < num_fds; i++) {
      if (pthread_create((tid + i), NULL, process_fd, (void *)(fd + i)))
          fprintf(stderr, "Could not create thread %i: %s\n", i,
                  strerror(errno));
   }
   for (i = 0; i < num_fds; i++)
       pthread_join(*(tid + i), NULL);
}
```

Programa 9.7

9.2 Hilos POSIX

Un paquete de hilos representativo contiene un sistema de tiempo de ejecución para gestionar los hilos de forma transparente (es decir, sin que el usuario se dé cuenta de la existencia del sistema de tiempo de ejecución). El paquete de hilos por lo regular incluye llamadas para crear y destruir hilos, exclusión mutua y variables de condición. Las bibliotecas de hilos tanto de Sun Solaris 2 como del estándar POSIX.1c cuentan con llamadas. Estos paquetes contemplan la creación y destrucción dinámicas de hilos, así que no es necesario conocer el número de hilos antes de iniciarse la ejecución. En la tabla 9.1 se resumen las funciones de hilos comunes en POSIX y en Sun Solaris 2.

La mayor parte de las funciones de hilos devuelven 0 si tuvieron éxito y un código de error distinto de cero si no fue así. La función `pthread_create` (`thr_create`) crea un hilo para ejecutar una función especificada; `pthread_exit` (`thr_exit`) hace que el hilo invocador termine sin causar que todo el proceso llegue a su fin. La función `pthread_kill` (`thr_kill`) envía una señal a un hilo especificado; `pthread_join` (`thr_join`) hace que el hilo invocador espere a que termine el hilo especificado. Esta llamada es similar a `waitpid` en el nivel de procesos. Por último, `pthread_self` (`thr_self`) devuelve la identidad del invocador. El resto de las llamadas de la tabla tienen que ver con mecanismos de sincronización que veremos en el siguiente capítulo.

Los hilos POSIX.1c y Sun Solaris son muy similares. Una diferencia importante la constituye la forma en que se asocian propiedades a los hilos. Los hilos POSIX se valen de objetos atributos para representar propiedades de los hilos. Las propiedades como el tamaño de pila o la política de programación se establecen para un objeto atributo de hilo. Varios hilos pueden estar asociados al mismo objeto atributo de hilo. Si una propiedad del objeto cambia, se refleja en todos los hilos asociados a ese objeto. Los hilos Solaris establecen explícitamente las propie-

Descripción	POSIX	Solaris 2
Gestión de hilos	pthread_create	thr_create
	pthread_exit	thr_exit
	pthread_kill	thr_kill
	pthread_join	thr_join
	pthread_self	thr_self
Exclusión mutua	pthread_mutex_init	mutex_init
	pthread_mutex_destroy	mutex_destroy
	pthread_mutex_lock	mutex_lock
	pthread_mutex_trylock	mutex_trylock
	pthread_mutex_unlock	mutex_unlock
Variables de condición	pthread_cond_init	cond_init
	pthread_cond_destroy	cond_destroy
	pthread_cond_wait	cond_wait
	pthread_cond_timedwait	cond_timedwait
	pthread_cond_signal	cond_signal
	pthread_cond_broadcast	cond_broadcast

Tabla 9.1: Comparación de llamadas para hilos POSIX.1c y para hilos Sun Solaris 2.

dades de los hilos y otras primitivas, de modo que algunas llamadas tienen largas listas de parámetros para establecer dichas propiedades. Los hilos Solaris ofrecen mayor control sobre la correspondencia entre los hilos y los recursos de procesadores, mientras que los hilos POSIX ofrecen un método más robusto de cancelación y de terminación de hilos. En este libro nos concentraremos en los hilos POSIX.

9.3 Gestión de hilos básica

Un hilo tiene un identificador (ID), una pila, una prioridad de ejecución y una dirección de inicio de la ejecución. Hacemos referencia a los hilos POSIX mediante un ID de tipo `pthread_t`. Un hilo puede averiguar su ID llamando a `pthread_self`. La estructura de datos interna del hilo también puede contener información de programación y uso. Los hilos de un proceso comparten todo el espacio de direcciones de ese proceso; pueden modificar variables globales, acceder a descriptores de archivo abiertos o interferirse mutuamente de otras maneras.

Se dice que un hilo es *dinámico* si se puede crear en cualquier instante durante la ejecución de un proceso y si no es necesario especificar por adelantado el número de hilos. En POSIX, los hilos se crean dinámicamente con la función `pthread_create`, la cual crea un hilo y lo coloca en una cola de hilos preparados.

SINOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
void pthread_exit(void *value_ptr);
int pthread_join(pthread_t thread, void **value_ptr);
```

POSIX.1c

El parámetro `tid` de `pthread_create` apunta al ID del hilo que se crea. Los atributos del hilo se encapsulan en el objeto atributo al que apunta `attr`. Si `attr` es `NULL`, el nuevo hilo tendrá los atributos por omisión. El tercer parámetro, `start_routine`, es el nombre de una función a la que el hilo invoca cuando inicia su ejecución. `start_routine` requiere un solo parámetro que se especifica con `arg`, un apuntador a `void`. La función `start_routine` devuelve un apuntador a `void` que `pthread_join` trata como una situación de salida. No se asuste al ver el prototipo de `pthread_create`; es muy fácil crear y utilizar hilos.

La función `pthread_exit` termina el hilo que la invoca. El valor del parámetro `value_ptr` queda disponible para `pthread_join` si ésta tuvo éxito. Sin embargo, el `value_ptr` en `pthread_exit` debe apuntar a datos que existan después de que el hilo ha terminado, así que no puede asignarse como datos locales automáticos para el hilo que está terminando.

Ejemplo 9.5

El siguiente segmento de código crea un hilo con los atributos por omisión.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
pthread_t copy_tid;
int myarg[2];
void *copy_file(void *arg);

if ((myarg[0] = open("my.in", O_RDONLY)) == -1)
   perror("Could not open my.in");
else if ((myarg[1] = open("my.out",
      O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) == -1)
   perror("Could not open my.out");
else if (pthread_create(&copy_tid, NULL, copy_file, (void *)myarg))
   perror("Thread creation was not successful");
```

En el ejemplo 9.5, `copy_tid` contiene el ID del hilo que se creó y `copy_file` (copiar archivo) es el nombre de la función que el hilo debe ejecutar. `myarg` es un apuntador al valor de parámetro que debe pasarse a la función de hilo. En este caso el arreglo `myarg` contiene descriptores de archivo abiertos para los archivos `my.in` y `my.out`.

El programa 9.8 muestra una implementación de una función `copy_file` que lee de un archivo y envía la salida a otro archivo. El parámetro `arg` contiene un apuntador a un par de descriptores abiertos que representan los archivos de origen y de destino. A las variables `infile` (archivo de entrada), `outfile` (archivo de salida), `bytes_read` (bytes leídos), `bytes_written` (bytes escritos), `bytes_copied_p` (apuntador a bytes copiados), `buffer` y `bufp` (apuntador al *buffer*) se les asigna espacio en la pila local de `copy_file` y no están accesibles directamente a otros hilos. El hilo también asigna espacio con `malloc` para devolver el número total de bytes copiados. La implementación supone que `malloc` es segura respecto a los hilos; de no ser así, debe utilizarse una versión `malloc_r`. La función `copy_file` podría devolver el apuntador `bytes_copied` en lugar de llamar a `pthread_exit`. La función `pthread_exit` invoca controladores de terminación de hilos, cosa que `return` no hace.

Programa 9.8: La función `copy_file` copia los contenidos de `infile` en `outfile`.

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

#define BUFFERSIZE 100

void *copy_file(void *arg)
{
   int infile;
   int outfile;
   int bytes_read = 0;
   int bytes_written = 0;
   int *bytes_copied_p;
   char buffer[BUFFERSIZE];
   char *bufp;
        /* abrir descriptores para los archivos de origen y de destino */
   infile = *((int *)(arg));
   outfile = *((int *)(arg) + 1);
   if ((bytes_copied_p = (int *)malloc(sizeof(int))) == NULL)
     pthread_exit(NULL);
   *bytes_copied_p = 0;

   for ( ; ; ) {
      bytes_read = read(infile, buffer, BUFFERSIZE);
      if ((bytes_read == 0) || ((bytes_read < 0) && (errno != EINTR)))
         break;
      else if ((bytes_read < 0) && (errno == EINTR))
         continue;
      bufp = buffer;
      while (bytes_read > 0) {
         bytes_written = write(outfile, bufp, bytes_read);
```

```
            if ((bytes_written < 0) && (errno != EINTR))
               break;
            else if (bytes_written < 0)
               continue;
            *bytes_copied_p += bytes_written;
            bytes_read -= bytes_written;
            bufp += bytes_written;
         }
         if (bytes_written == -1)
            break;
      }
      close(infile);
      close(outfile);
      pthread_exit(bytes_copied_p);
}
```

Programa 9.8

El programa 9.9 muestra un programa principal con tres argumentos de línea de comandos, un nombre base de archivo de entrada, un nombre base de archivo de salida y el número de archivos copiadores. El programa crea `numcopiers` hilos. El hilo `i` copia `infile_name.i` en `outfile_name.i`.

Programa 9.9: Programa que crea hilos para copiar múltiples descriptores de archivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>

#define MAXNUMCOPIERS 10
#define MAXNAMESIZE 80

void *copy_file(void *arg);

void main(int argc, char *argv[])
{
   pthread_t copiertid[MAXNUMCOPIERS];
   int fd[MAXNUMCOPIERS][2];
   char filename[MAXNAMESIZE];
   int numcopiers;
   int total_bytes_copied=0;
   int *bytes_copied_p;
```

```
    int i;

    if (argc != 4) {
       fprintf(stderr, "Usage: %s infile_name outfile_name copiers\n",
          argv[0]);
       exit(1);
    }
    numcopiers = atoi(argv[3]);
    if (numcopiers < 1 || numcopiers > MAXNUMCOPIERS) {
       fprintf(stderr, "%d invalid number of copiers\n", numcopiers);
       exit(1);
    }
                                        /* crear los hilos copiadores */
    for (i = 0; i < numcopiers; i++) {
       sprintf(filename, "%s.%d", argv[1], i);
       if ((fd[i][0] = open(filename, O_RDONLY)) < 0) {
          fprintf(stderr, "Unable to open copy source file %s: %s\n",
                          filename, strerror(errno));
          continue;
       }
       sprintf(filename, "%s.%d", argv[2], i);
       if ((fd[i][1]=
          open(filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
          fprintf(stderr,
             "Unable to create copy destination file %s: %s\n",
             filename, strerror(errno));
          continue;
       }
       if (pthread_create(&copiertid[i], NULL, copy_file,
             (void *)fd[i]) != 0)
          fprintf(stderr, "Could not create thread %i: %s\n",
             i, strerror(errno));
    }
                                        /* esperar que se termine de copiar */
    for (i = 0; i < numcopiers; i++) {
       if (pthread_join(copiertid[i], (void **)&(bytes_copied_p)) != 0)
          fprintf(stderr, "No thread %d to join: %s\n",
             i, strerror(errno));
       else {
          printf("Thread %d copied %d bytes from %s.%d to %s.%d\n",
             i, *bytes_copied_p, argv[1], i, argv[2], i);
          total_bytes_copied += *bytes_copied_p;
       }
    }
    printf("Total bytes copied = %d\n", total_bytes_copied);
    exit(0);
}
```

Programa 9.9

Cuando un hilo `copy_file` completa su trabajo, termina invocando a `pthread_exit`. La situación de terminación de un hilo que llama a `pthread_exit` se conserva hasta que otro hilo se une a él o si ya es el último hilo del proceso. La función `pthread_join` es similar a `waitpid` para procesos hijo en cuanto a que el hilo invocador se bloquea hasta que el hilo indicado termina. El hilo invocador recupera el número de bytes copiado por el hilo a través del `status_value` devuelto por `pthread_join`. El hilo asigna espacio dinámicamente para `status_value` de modo que la variable persista después de que el hilo termine. Por último, el hilo principal termina.

Los hilos tienen un atributo `detachstate` (estado de desconexión) de `PTHREAD_CREATE_JOINABLE` (unible) por omisión. El otro valor que `detachstate` puede tener es `PTHREAD_CREATE_DETACHED` (desconectado). Los hilos desconectados deben llamar a `pthread_detach` en lugar de a `pthread_exit` para liberar sus recursos.

Ejercicio 9.1

¿Qué sucede en el programa 9.9 si falla la `malloc` de `copy_file`?

Respuesta:

En el programa principal después del retorno de `pthread_join`, `bytes_copied_p` es `NULL` y el programa se cae cuando trata de obtener la dirección a la que apunta este apuntador. El problema puede corregirse haciendo que el programa principal verifique si este apuntador es `NULL`.

En las implementaciones tradicionales de UNIX, `errno` es una variable global externa que se establece cuando las funciones del sistema producen un error. Esta implementación no funciona con multihilos (véase la sección 1.5), y en la mayor parte de las implementaciones de hilos `errno` es una macro que devuelve información específica para un hilo. En esencia, cada hilo tiene su propia copia de `errno`. El programa principal no tiene acceso directo al `errno` de un hilo unido, de modo que si se necesita esta información se deberá devolver a través del último parámetro de `pthread_join`.

Ejercicio 9.2

¿Qué sucede en el programa 9.9 si el `write` de `copy_file` fracasa?

Respuesta:

La función `copy_file` devuelve el número de bytes copiados con éxito y el programa principal no detecta ningún error. El problema puede corregirse haciendo que `copy_file` devuelva un apuntador a una estructura que contenga tanto el número de bytes copiados como un valor de error.

Un aspecto delicado es el paso de parámetros cuando hay múltiples hilos. En el programa 9.9 se crean varios hilos, pero a cada hilo le son pasados sus descriptores de archivo en diferentes entradas del arreglo `fd`, así que los hilos no se interfieren. Tenga cuidado al reutilizar variables que se pasan por referencia a los hilos cuando éstos se crean. Podría suceder que el hilo creado no se programara para ejecutarse a tiempo a fin de utilizar los valores antes de que se sobreescriban.

El programa 9.10 muestra una modificación del programa 9.9 que utiliza un solo par de posiciones para los descriptores de archivo. Este programa fracasa si el descriptor de archivo para el hilo `i` es sobreescrito por la siguiente iteración del ciclo `for` antes de que el hilo `i` haya tenido oportunidad de copiar el descriptor en su memoria local.

Programa 9.10: Programa que pasa incorrectamente múltiples descriptores de archivo a hilos.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>

void *copy_file(void *arg);
#define MAXNUMCOPIERS 10
#define MAXNAMESIZE 80

void main(int argc, char *argv[])
{
   pthread_t copiertid[MAXNUMCOPIERS];
   int fd[2];
   char filename[MAXNAMESIZE];
   int numcopiers;
   int total_bytes_copied=0;
   int *bytes_copied_p;
   int i;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s infile_name outfile_name copiers\n",
         argv[0]);
      exit(1);
   }
   numcopiers = atoi(argv[3]);
   if (numcopiers < 1 || numcopiers > MAXNUMCOPIERS) {
      fprintf(stderr, "%d invalid number of copiers\n", numcopiers);
      exit(1);
   }

                              /* crear los hilos copiadores */
   for (i = 0; i < numcopiers; i++) {
      sprintf(filename, "%s.%d", argv[1], i);
      if ((fd[0] = open(filename, O_RDONLY)) < 0) {
          fprintf(stderr, "Unable to open copy source file %s: %s\n",
                           filename, strerror(errno));
```

```
            continue;
        }
        sprintf(filename, "%s.%d", argv[2], i);
        if ((fd[1] =
            open(filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
            fprintf(stderr,
                "Unable to create copy destination file %s: %s\n",
                filename, strerror(errno));
            continue;
        }
        if (pthread_create(&copiertid[i], NULL, copy_file,
                (void *)fd) != 0)
            fprintf(stderr, "Could not create thread %i: %s\n",
                i, strerror(errno));
    }
                                        /* esperar que se termine de copiar */
    for (i = 0; i < numcopiers; i++) {
        if (pthread_join(copiertid[i], (void **)&(bytes_copied_p)) != 0)
            fprintf(stderr, "No thread %d to join: %s\n",
                i, strerror(errno));
        else {
            printf("Thread %d copied %d bytes from %s.%d to %s.%d\n",
                i, *bytes_copied_p, argv[1], i, argv[2], i);
            total_bytes_copied += *bytes_copied_p;
        }
    }
    printf("Total bytes copied = %d\n", total_bytes_copied);
    exit(0);
}
```

Programa 9.10

Ejercicio 9.3

Pruebe el código del programa 9.10. Haga que se exhiba en la pantalla cada descriptor de archivo cuando sea abierto y al principio de la ejecución hilo al cual se pasó como parámetro. ¿Son iguales los dos valores del descriptor? ¿Qué sucede si se inserta un `sleep(5)` dentro del ciclo `for` después de `pthread_create`?

Respuesta:

Se abre un par diferente de descriptores de archivo para cada hilo, pero el arreglo `fd` se reutiliza con cada hilo. Todos los hilos se crean con la misma prioridad que el hilo principal y se colocan en la cola de hilos listas para ejecución (ready). Si la siguiente iteración del ciclo comienza antes de que se programe para ejecución el hilo creado por la iteración anterior, se sobreescribirán los descriptores de archivo y el hilo copiará los archivos equivocados. Si se coloca un `sleep` después de `pthread_create`, es muy probable que el hilo tenga oportunidad de ejecutarse antes de que esto suceda y el programa deberá funcionar correctamente.

Los hilos copiadores individuales del programa 9.9 trabajan con problemas independien
tes y no iteractúan entre sí. En aplicaciones más complicadas puede suceder que un hilo no
termine después de completar su tarea asignada. En vez de ello, un hilo trabajador podría
solicitar tareas adicionales o compartir información. En el capítulo 10 se explica cómo puede
controlarse este tipo de interacción mediante mecanismos de sincronización como los canda-
dos mutex y las variables de condición.

Un problema poco oculto del uso de hilos es que éstos pueden invocar funciones de bibliote-
ca o hacer llamadas de sistema que no sean seguras respecto a los hilos, produciendo tal vez re-
sultados indeseables. Incluso funciones como `sprintf` y `fprintf` podrían no ser seguras
respecto de los hilos, así que tenga cuidado. POSIX.1c especifica que todas las funciones re-
queridas, incluida la biblioteca de C estándar, se implemente en forma segura respecto a los
hilos. Aquellas funciones cuyas interfaces tradicionales impiden hacerlas seguras respecto a
los hilos deben contar con una versión alternativa segura designada con un sufijo `_r`. Las pá-
ginas del manual de Sun Solaris 2 casi siempre indican si una función es o no segura respecto
de los hilos en el apartado `MT_LEVEL`.

9.4 Usuario de hilos *versus* hilos de núcleos (*kernel*)

Recuerde que un procesador funciona ejecutando su ciclo de instrucción y que el valor del
contador de programa determina cuál proceso se está ejecutando. El sistema operativo tiene
oportunidades de recuperar el control modificando el valor del contador de programa cuando
ocurren interrupciones y cuando los programas solicitan servicios mediante llamadas de siste-
ma. Cuando se usan hilos, surgen cuestiones de control similares en el nivel de proceso. Los
dos modelos tradicionales de control de hilos son el de *hilos a nivel de usuario* y el de *hilos a
nivel de núcleo (kernel)*.

Los paquetes de hilos a nivel de usuario suelen ejecutarse sobre un sistema operativo
existente. Los hilos dentro del proceso son invisibles para el núcleo. Estos hilos compiten
entre sí por los recursos asignados a un proceso, como se aprecia en la figura 9.3. Los hilos son
programados por un sistema de hilos de tiempo de ejecución que forma parte del código del
proceso. Los programas que utilizan un paquete de hilos a nivel de usuario por lo regular se
ligan con una biblioteca especial en la que cada función de biblioteca y llamada de sistema
está dentro de una envoltura. El código de envoltura llama al sistema de tiempo de ejecución
para que se encargue de la gestión de hilos.

Las llamadas de sistema como `read` o `sleep` podrían representar un problema para hilos
a nivel de usuario porque pueden hacer que el proceso se bloquee. A fin de evitar el problema
de que una llamada bloqueadora haga que todo el proceso se bloquee, cada llamada potencial-
mente bloqueadora se sustituye en la envoltura por una versión no bloqueadora. El sistema de
hilos de tiempo de ejecución verifica si la llamada puede hacer que el hilo se bloquee. Si no es
así, el sistema de tiempo de ejecución realiza la llamada de inmediato. Sin embargo, si la
llamada bloquearía el hilo, el sistema de tiempo de ejecución bloquea el hilo, añade la llamada
a una lista de cosas que intentará más tarde y escogerá otro hilo para ejecutarlo. Todo este
control queda fuera de la vista del usuario y del sistema operativo.

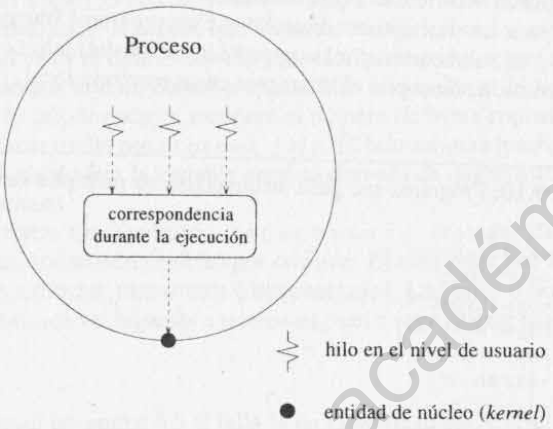

Figura 9.3: Los hilos en el nivel de usuario no pueden verse desde fuera del proceso.

Los hilos a nivel de usuario tienen muy poco gasto extra, pero presentan algunas desventajas. El modelo necesita tener hilos que permitan al sistema de hilos de tiempo de ejecución recuperar el control. Un *hilo limitado por la CPU* rara vez efectúa llamadas de sistema o de biblioteca y por tanto evita que el sistema de hilos de tiempo de ejecución recupere el control para programar otros hilos. El programador debe evitar la situación de exclusión obligando explícitamente a los hilos limitados por la CPU a ceder el control en los puntos apropiados. Un segundo problema de los hilos a nivel de usuario, más grave, es que los hilos sólo pueden compartir recursos de procesador asignados a su proceso encapsulante. Esta restricción limita el grado de paralelismo porque los hilos sólo se pueden ejecutar en un procesador a la vez. Puesto que una de las principales razones para usar hilos es aprovechar el paralelismo que ofrecen las estaciones de trabajo de multiprocesador, los hilos a nivel de usuario por sí solos no son una estrategia aceptable.

Con los hilos a nivel de núcleo *(kernel)*, el núcleo está consciente de cada hilo como una entidad programable. Los hilos compiten por los recursos de procesador en todo el sistema, como se muestra en la figura 9.4. La programación de la ejecución de los hilos a nivel de núcleo puede ser tan costosa como la programación de los procesos mismos, pero los hilos a nivel de núcleo pueden aprovechar la existencia de múltiples procesadores. La sincronización y el compartimiento de datos que los hilos a nivel de núcleo hacen posible es menos costosa que en el caso de procesos completos, pero estos hilos son mucho más costosos que los hilos a nivel de usuario.

Los *modelos de hilos híbridos* ofrecen las ventajas de los hilos tanto a nivel de usuario como a nivel de núcleo al proveer dos niveles de control. La figura 9.5 muestra un enfoque híbrido representativo. El usuario escribe el programa en términos de hilos a nivel de usuario y luego especifica cuántas entidades programables por el núcleo están asociadas al proceso. Durante la ejecución del proceso se establece una correspondencia entre los hilos a nivel de

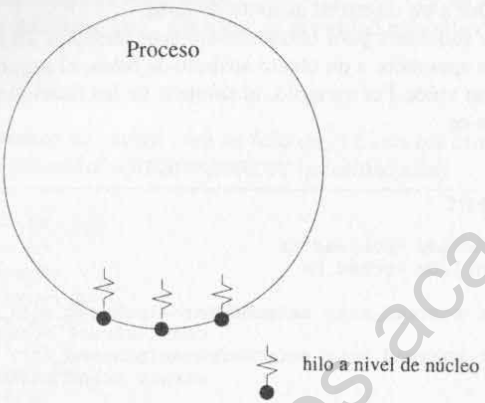

Figura 9.4: Los hilos a nivel de núcleo (*kernel*) se programan igual que los procesos individuales.

usuario y las entidades programables por el núcleo a fin de lograr el paralelismo. El grado de control que un usuario tiene sobre esta correspondencia depende de la implementación. En la implementación de hilos Sun Solaris, por ejemplo, los hilos a nivel de usuario se llaman hilos y las entidades programables por el núcleo se denominan *procesos ligeros (light weigth processes)*. El usuario puede especificar que un hilo en particular tenga un proceso ligero dedicado o que cierto grupo de hilos sea ejecutado por un conjunto de procesos ligeros.

El modelo de programación de hilos POSIX.1c es un modelo híbrido con la suficiente flexibilidad para apoyar hilos tanto a nivel de usuario como a nivel de núcleo en implementaciones específicas del estándar. El modelo consiste en dos niveles de programación: hilos y entidades de núcleo. Los hilos son análogos a aquellos a nivel de usuario. Las entidades de núcleo son programadas por el núcleo. La biblioteca de hilos decide cuántas entidades de núcleo necesita y cómo se establecerá la correspondencia con ellas.

POSIX.1c introduce la idea de *alcance de contención de programación de hilos* que proporciona al programador cierto control sobre la correspondencia entre las entidades de núcleo y los hilos. Un hilo puede tener un atributo `contentionscope` (alcance de contención) de `PTHREAD_SCOPE_PROCESS` (proceso) o `PTHREAD_SCOPE_SYSTEM` (sistema). Los hilos que son `PTHREAD_SCOPE_PROCESS` compiten con los demás hilos de su proceso por recursos de procesador. POSIX no especifica cómo un hilo de este tipo compite con los hilos externos a su propio proceso, así que los hilos `PTHREAD_SCOPE_PROCESS` pueden ser hilos estrictamente a nivel de usuario o bien pueden hacerse corresponder con un conjunto de entidades de núcleo de alguna otra forma más complicada.

Los hilos que son `PTHREAD_SCOPE_SYSTEM` compiten por los recursos de procesador dentro de todo el sistema, en forma parecida a como lo hacen los hilos a nivel de núcleo. POSIX deja la correspondencia entre los hilos `PTHREAD_SCOPE_SYSTEM` y las entidades de núcleo a

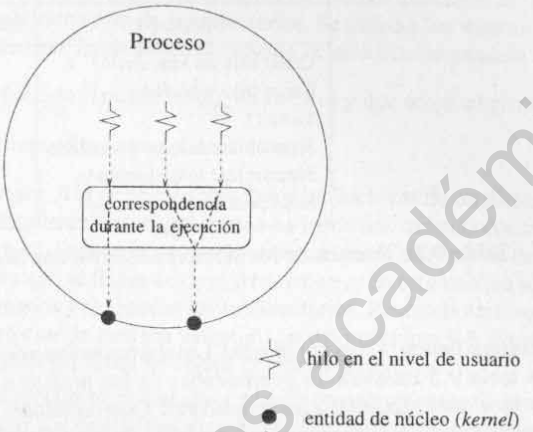

Figura 9.5: El modelo híbrido tiene dos niveles de programación, con una correspondencia entre los hilos a nivel de usuario y las entidades de núcleo.

la implementación, pero la correspondencia obvia es vincular un hilo de este tipo directamente con una entidad de núcleo. Una implementación de hilos POSIX puede apoyar hilos `PTHREAD_SCOPE_PROCESS` o `PTHREAD_SCOPE_SYSTEM`, o ambos.

En la tabla 9.2 se muestra el costo relativo de los hilos a nivel de usuario y a nivel de núcleo, como se presenta en el *Sun Solaris 2.3 Software Developer Answerbook*. Solaris 2 emplea un modelo de hilos de dos niveles similar a la especificación POSIX. En la terminología de Sun Microsystems, un *hilo no vinculado (unbound thread)* es un hilo a nivel de usuario, y un *hilo vinculado (bound thread)* es un hilo a nivel de núcleo porque está vinculado con un proceso ligero. El `fork` es el costo de la creación de un proceso completo. La sincronización se refiere a que dos hilos se sincronizan con semáforos como en el problema del productor-consumidor. Cuesta entre seis y siete veces más crear y sincronizar un hilo a nivel de núcleo que uno a nivel de usuario. Cuesta alrededor de treinta veces más crear un proceso completo con un `fork` que crear un hilo a nivel de usuario.

9.5 Atributos de los hilos

POSIX.1c adopta un enfoque orientado a objetos respecto de la representación y asignación de propiedades. Cada hilo POSIX.1c tiene un objeto atributo asociado que representa sus propiedades. Un objeto atributo de hilo puede estar asociado a varios hilos, y POSIX.1c tiene funciones para crear, configurar y destruir objetos atributo. El enfoque orientado a objetos permite a un programa agrupar entidades tales como los hilos y asociar el mismo objeto atributo a todos los miembros del grupo. Cuando cambia una propiedad del objeto atributo, todas las entidades

Operación	Microsegundos
Crear hilo no vinculado	52
Crear hilo vinculado	350
`fork()`	1700
Sincronizar hilo no vinculado	66
Sincronizar hilo vinculado	390
Sincronizar entre procesos	200

Tabla 9.2: Tiempos de los servicios de hilos con Solaris 2.3 en una SPARCstation 2.

del grupo tienen la nueva propiedad. Los objetos atributo de los hilos son de tipo `pthread_attr_t`. La tabla 9.3 muestra las propiedades de los atributos de hilo que se pueden establecer y las funciones asociadas a las propiedades. Otras entidades, como las variables de condición o los candados mutex, tienen sus propios tipos de objeto atributo y funciones, como se verá en el capítulo 10.

La función `pthread_attr_init` inicializa un objeto atributo de hilos con los valores por omisión. La función `pthread_attr_destroy` hace que el valor del objeto atributo sea no válido. POSIX no especifica el comportamiento del objeto una vez que ha sido destruido.

Propiedad	Función
Inicialización	`pthread_attr_init`
	`pthread_attr_destroy`
Tamaño de pila	`pthread_attr_setstacksize`
	`pthread_attr_getstacksize`
Dirección de pila	`pthread_attr_setstackaddr`
	`pthread_attr_getstackaddr`
Estado de desconexión	`pthread_attr_setdetachstate`
	`pthread_attr_getdetachstate`
Alcance	`pthread_attr_setscope`
	`pthread_attr_getscope`
Herencia	`pthread_setinheritsched`
	`pthread_getinheritsched`
Política de programación	`pthread_attr_setschedpolicy`
	`pthread_attr_getschedpolicy`
Parámetros de programación	`pthread_attr_setschedparam`
	`pthread_attr_getschedparam`

Tabla 9.3: Resumen de propiedades que se pueden establecer para objetos de atributo de hilos POSIX.1c.

Tanto `pthread_attr_init` como `pthread_attr_destroy` llevan un solo argumento que es un apuntador a un objeto de atributo de hilos.

Todas las funciones para obtener/establecer atributos de hilos tienen dos parámetros. El primero es un apuntador a un objeto atributo de hilos; el segundo es el valor del atributo o un apuntador a un valor. Por ejemplo, la sinopsis de las funciones para manipular la política de programación es

SINOPSIS

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

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               struct sched_param *param);
```

POSIX.1c

Un hilo tiene una pila cuya ubicación y tamaño se puede examinar o establecer mediante las llamadas `pthread_attr_getstackaddr` (obtener dirección), `pthread_attr_setstackaddr` (establecer dirección), `pthread_attr_getstacksize` (obtener tamaño) y `pthread_attr_setstacksize` (establecer tamaño). Cuando un hilo se desconecta, ya no se puede esperar con un `pthread_join`. Las funciones `pthread_attr_getdetachstate` y `pthread_attr_setdetachstate` pueden examinar y establecer el `detachstate` (estado de desconexión) de un hilo. Los posibles valores de `detachstate` son `PTHREAD_CREATE_JOINABLE` (unible) o `PTHREAD_CREATE_DETACHED` (desconectado). Por omisión, los hilos son unibles. Los hilos desconectados invocan a `pthread_detach` cuando terminan para liberar sus recursos.

Las funciones `pthread_attr_getscope` y `pthread_attr_setscope` examinan y establecen el atributo `contentionscope` (alcance de contención) que controla si el hilo compite por recursos dentro del proceso o bien en el nivel del sistema. Los posibles valores de `contentionscope` son `PTHREAD_SCOPE_PROCESS` y `PTHREAD_SCOPE_SYSTEM`.

La función `pthread_attr_getinheritsched` examina el atributo `inheritsched` que controla si los parámetros de programación se heredan del hilo creador o se especifican explícitamente. La función `pthread_attr_setinheritsched` establece este atributo. Los posibles valores de `inheritsched` son `PTHREAD_INHERIT_SCHED` (se heredan) y `PTHREAD_EXPLICIT_SCHED` (se especifican explícitamente).

La política de programación de un hilo se almacena en una estructura de tipo `struct sched_param`. El submiembro `sched_policy` de `struct sched_param` contiene la política de programación. Las posibles políticas de programación son "el primero que entra es el primero que sale" (`SCHED_FIFO`), turno circular (`SCHED_RR`) o definido por la implementación (`SCHED_OTHER`). La implementación más común de `SCHED_OTHER` es una política de prioridad apropiativa. Una implementación que se ajuste a POSIX podrá apoyar cualquiera de estas políticas de programación. El comportamiento real de la política en la implementación depende del alcance de programación y de otros factores.

La propiedad con mayores probabilidades de cambiar es la de un hilo, que forma parte de la política de programación. El submiembro `sched_priority` de `struct sched_param` contiene un valor de prioridad `int`. Un valor de prioridad más alto corresponde a una prioridad más alta.

Ejemplo 9.6

El siguiente segmento de código crea un hilo `do_it` *con los atributos por omisión y luego cambia la prioridad a* `HIGHPRIORITY` (prioridad alta).

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <sched.h>

#define HIGHPRIORITY 10

pthread_attr_t my_tattr;
pthread_t my_tid;
struct sched_param param;
int fd;

if (pthread_attr_init(&my_tattr))
   perror("Could not initialize thread attribute object");
else if (pthread_create(&my_tid, &my_tattr, do_it, (void *)&fd))
   perror("Could not create copier thread");
else if (pthread_attr_getschedparam(&my_tattr, &param))
   perror("Could not get scheduling parameters");
else {
   param.sched_priority = HIGHPRIORITY;
   if (pthread_attr_setschedparam(&my_tattr, &param))
      perror("Could not set priority");
}
```

La `pthread_create` del ejemplo 9.6 asocia el objeto de atributo de hilo `my_tattr` al hilo `my_tid` durante su creación a fin de poder modificar posteriormente los atributos asociados a este hilo. Observe que la prioridad del hilo `my_tid` se cambia modificando una propiedad del objeto de atributo de hilo `my_tattr` que se asoció al hilo.

9.6 Ejercicio: Copiado de archivos en paralelo

En esta sección creamos un copiador de archivos en paralelo como una extensión de la aplicación de copiado del programa 9.9. Asegúrese de utilizar llamadas seguras respecto de los hilos en la implementación. El programa principal requiere dos argumentos de línea de comando que son

nombres de directorio y copia todos los archivos del primer directorio al segundo. El programa de copiado preserva la estructura de subdirectorios. Se utilizan los mismos nombres de archivo para el origen y el destino. Implemente el copiado de archivos en paralelo como sigue:

- Escriba una función llamada `copy_directory` que tenga el prototipo

```
void *copy_directory(void *arg)
```

  La función `copy_directory` copia todos los archivos de un directorio a otro. Los nombres de los directorios se pasan en `arg` como dos cadenas consecutivas (separadas por un `NULL`). Suponga que ambos directorios, de origen y de destino, existen en el momento en que se llama a `copy_directory`. En esta versión sólo se copian los archivos ordinarios y se ignoran los subdirectorios. Para cada archivo por copiar, cree un hilo que ejecute la función `copy_file` del programa 9.8. Espere a que el hilo termine su ejecución antes de copiar el siguiente archivo.
- Escriba un programa principal que requiera dos argumentos de línea de comandos para especificar los directorios de origen y de destino. El programa principal crea un hilo para ejecutar `copy_directory` y luego efectúa un `pthread_join` para esperar que el hilo `copy_directory` termine. Utilice el programa principal para probar la primera versión de `copy_directory`.
- Modifique la función `copy_directory` de modo que, si el directorio de destino no existe, lo cree. Pruebe la nueva versión.
- Modifique `copy_directory` de modo que, después de crear un hilo para copiar un archivo, siga creando hilos para crear los demás archivos. Conserve el ID de hilo y los descriptores de archivo abiertos para cada hilo `copy_file` en una lista enlazada con una estructura de nodos similar a

```
typedef struct copy_struct {
    char *namestring;
    int source_fd;
    int destination_fd;
    pthread_t tid;
    struct copy_struct *next_thread;
} copyinfo_t;
copyinfo_t *copy_head = NULL;
copyinfo_t *copy_tail = NULL;
```

  Implemente la lista como un objeto colocando su declaración en un archivo aparte junto con funciones de acceso para insertar, recuperar y eliminar nodos. Una vez que la función `copy_directory` haya creado hilos para copiar todos los archivos del directorio, realizará un `pthread_join` con cada hilo de su lista y liberará la estructura `copyinfo_t`.
- Modifique la función `copy_file` del programa 9.8 de modo que su argumento sea un apuntador a una estructura `copyinfo_t`. Pruebe las nuevas versiones de `copy_file` y `copy_directory`.

- Modifique `copy_directory` de modo que, si un archivo es un directorio en lugar de un archivo ordinario, la función cree un hilo para ejecutar `copy_directory` en vez de `copy_file`. Pruebe la nueva función.
- Invente un método de cronometría para comparar un copiado ordinario con el copiado por hilos.
- Si el programa se ejecuta con un directorio grande, puede tratar de abrir más descriptores de archivo de los que están permitidos para un proceso. Invente un método para manejar esta situación. Algunos intérpretes de comandos (*shells*) permiten al usuario cambiar este límite.
- Determine si cambia el tiempo de ejecución cuando los hilos tienen el alcance `PTHREAD_SCOPE_SYSTEM` en vez de `PTHREAD_SCOPE_PROCESS`.

9.7 Lecturas adicionales

El libro *Distributed Operating Systems* de Tanenbaum [93] trata el tema de los hilos en forma sumamente accesible. Las diferentes estrategias de programación de hilos se analizan en los artículos [3, 10, 27, 48]. Un libro de Kleiman *et al.* [44] que está por aparecer con el título *Programming with Threads* analiza técnicas avanzadas de programación con hilos. Por último, el estándar POSIX.1c [54] es en sí una relación sorprendentemente amena de las consideraciones y elecciones en conflicto que debieron hacerse al implementar un paquete de hilos utilizable.