9. IPC Y THREADS

9.1 MECANIMOS IPC DEL UNIX

El paquete de comunicación entre procesos de UNIX se compone de tres mecanismos: los semáforos, que van a permitir sincronizar procesos; la memoria compartida, que va a permitir que los procesos compartan su espacio de direcciones virtuales, y las filas de mensajes, que posibilitan el intercambio de datos con un formato determinado.

Estos mecanismos están implementados como una unidad y comparten características comunes, entre las que se encuentran:

- Cada mecanismo tiene una tabla cuyas entradas describen el uso que se hace del mismo.
- Cada entrada de la tabla tiene una llave numérica elegida por el usuario.
- Cada mecanismo dispone de una llamada "get" para crear una entrada nueva o recuperar alguna ya existente. Dentro de los parámetros de esta llamada se incluye una llave y una máscara de indicadores. El kernel busca dentro de la tabla alguna entrada que se ajuste a la llave suministrada. Si la llave toma valor IPC_PRIVATE, el kernel ocupa la primera entrada que se encuentra libre y ninguna otra llamada "get" podrá devolver esta entrada hasta que la liberemos. Si dentro de la máscara de indicadores está activo el bit IPC_CREAT, el kernel crea una nueva entrada en caso de que no haya ninguna que corresponda con la llave suministrada. Si, además de IPC_CREAT, existe una entrada para la llave suministrada. Si todo funciona correctamente, el kernel devuelve un descriptor que se podrá usar en otras llamadas. Este descriptor cumple, para las llamadas relacionadas con las facilidades IPC, la misma finalidad que los descriptores de archivo para los casos de las llamadas relacionadas con el sistema de archivos.
- Para cada mecanismo IPC, el kernel aplica la fórmula siguiente a la hora de calcular el índice que da acceso a la tabla:

$$índice_{tabla} = número_{descriptor} \% (número\ de\ entradas\ en\ la\ tabla)$$

  Así, por ejemplo, si hay N procesos compartiendo la entrada número 3 de la tabla y la tabla consta de 100 entradas, cada uno de los procesos puede estar referenciado a la misma entrada a través de los descriptores 3, 103, 203, etc.
- Cada entrada de la tabla tiene un registro de permisos que incluye: identificador de usuario y grupo del proceso que ha reservado la entrada, identificador de usuario y de grupo modificados por la llamada "control" del mecanismo IPC, un conjunto de bits con los permisos de lectura, escritura y ejecución para el usuario, el grupo y otros usuarios.
- Cada entrada contiene información de estado, en la que se incluye el identificador del último proceso que ha utilizado la entrada.
- Cada mecanismo IPC tiene una llamada de "control" que permite lee y modificar el estado de una entrada reservada y también permite liberarla.

9.1.1 FORMACIÓN DE LLAVES

Una llave es una variable o constante del tipo `key_t` que vamos a usar para acceder a los mecanismos IPC previamente reservados o para reservar otros nuevos. Es normal que los mecanismos que están siendo utilizados como parte de un mismo proyecto compartan la misma llave.

Hay múltiples formas de crear llaves y es necesario que todo sistema defina algún procedimiento estándar para realizar esta función. Esto va impedir que procesos no relacionados, por no formar parte de un mismo proyecto, puedan interferirse involuntariamente debido a que tengan una misma llave.

La biblioteca de C aporta la función `ftok` para crear llaves de una forma estándar. Esta función tiene la siguiente declaración:

```
#include <types.h>
#include <sys/ipc.h>
key_t ftok(const char *path, char id);
```

`ftok` devuelve una llave basada en `path` y en `id`. Esta llave podrá usarse en llamadas futuras a `msgget`, `semget` y `shmget` para obtener un identificador que dé acceso a algún mecanismo IPC. `path` es un apuntador a la ruta de un archivo que debe existir dentro de nuestro sistema de archivos y que de ser accesible para el proceso que llama a `ftok`. `id` es un carácter que identifica el proyecto. `ftok` devolverá la misma llave para rutas enlazadas a un mismo archivo, siempre que se utilice el mismo valor para `id`. Para el mismo archivo, `ftok` devolverá diferentes llaves para distintos valores de id.

Si el archivo no es accesible para el proceso, bien porque no existe, bien porque sus permisos lo impiden, `ftok` devolverá el valor `(key_t) (-1)`, que indica que se ha producido un error en la creación de la llave.

Las siguientes líneas muestran la forma de llamar a `ftok` para crear una llave asociada al archivo "`prueba`" y al identificador '`A`':

```
key_t llave;
…
if ((llave = ftok("prueba", 'A')) == (key_t) -1) {
      /* error */
}
```

9.1.2 CONTROL DE LAS FACILIDADES IPC DESDE LA LÍNEA DE COMANDOS

Los programas estándar `ipcs` e `ipcrm` nos permiten controlar los recursos IPC que gestiona el sistema. `ipcs` se utiliza para ver los mecanismos que están asignados y a quien, junto con información estadística. `ipcrm` se emplea para liberar un mecanismo asignado y dejar libre la entrada que ocupa. Estos dos programas son bastante útiles para depurar programas que se valen de los mecanismos de IPC.

La forma de emplear `ipcs` es la siguiente:

```
$ ipcs [ opciones ]
```

Si no especifica ninguna opción, `ipcs` muestra un resumen de la información administrativa que se almacena para los semáforos, memoria compartida y filas de mensajes que hay asignados. Las principales opciones son:

Valores	Significado
-q	Muestra la información de las filas de mensajes activas.
-m	Muestra la información de los segmentos de memoria compartida que hay activos.
-s	Muestra la información de los semáforos activos.
-b	Muestra una información completa sobre los tres tipos de mecanismos IPC que hay activos.

La forma de emplear `ipcrm` es la siguiente:

```
$ ipcrm [ opciones ]
```

Algunas de las opciones que hay disponibles son:

Valores	Significado
-q msqid	Borra la fila de mensajes cuyo identificador coincide con `msqid`.
-m shmid	Borra la zona de memoria compartida cuyo identificador coincide con `shmid`.
-s semid	Borra el semáforo cuyo identificador coincide con `semid`.

9.2 SEMÁFOROS

Un semáforo es un mecanismo para prevenir la colisión que se produce cuando dos o más procesos solicitan simultáneamente el uso de un recurso que deben compartir.

9.2.1 CONCEPTOS GENERALES

Al igual que ocurre con los semáforos de tránsito, en programación los semáforos sólo tienen un carácter informativo. Si aun cuando un semáforo está prohibiendo el acceso a un recurso decidimos saltarnos esa prohibición, el resultado puede ser catastrófico.

Dijsktra define un semáforo como un objeto de tipo entero sobre el que se pueden realizar dos operaciones: `P` y `V`. La operación P decrementa el valor del semáforo y se utiliza para adquirirlo o bloquearlo. La operación `V` incremente el valor del semáforo y se utiliza para liberarlo o inicializarlo. Un semáforo no puede tomar un valor negativo y cuando vale 0 no se pueden realizar más operaciones `P` sobre él. Si el semáforo sólo puede tomar dos valores, se dice que es binario.

Las operaciones `P` y `V` deben ser atómicas para que funcionen correctamente. Esto quiere decir que una operación `P` no puede ser interrumpida por otra operación `P` o `V`, y lo mismo ocurre para la operación `V`. Esto garantiza que cuando varios procesos compitan por la adquisición de un semáforo, sólo uno de ellos podrá realizar la operación.

El mecanismo IPC de semáforos implementados en UNIX es una generalización del concepto de semáforo descrito por Dijsktra, ya que va a permitir manejar un conjunto de semáforos mediantes el uso de un

identificador asociado. También vamos a poder realizar operaciones `P` y `V` que actualizan de forma atómica todos los semáforos asociados bajo un mismo identificador.

Un semáforo en UNIX se compone de los siguientes elementos:

- El valor del semáforo.
- El identificador del último proceso que manipuló el semáforo.
- El número de procesos que hay esperando a que el semáforo se incremente.
- El número de procesos que hay esperando a que el semáforo tome el valor 0.

Las llamadas relacionadas con los semáforos son: `semget`, para crear un semáforo o habilitar el acceso a uno ya existente; `semctl`, para realizar operaciones de control e inicialización, y `semop`, para realizar operaciones `P` y `V` sobre el semáforo.

9.2.2 PETICIÓN DE SEMÁFOROS – SEMGET

Mediante `semget` vamos a poder crear o acceder a un conjunto de semáforos unidos bajo un identificador común. `semget` tiene la siguiente declaración:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Si la llamada funciona correctamente, devolverá un identificador con el que podremos acceder a los semáforos en sucesivas llamadas. Si algo falla, `semget` devuelve el valor -1 y en `errno` estará el código del error producido.

`key` es la llave que indica a qué grupo de semáforos queremos acceder. Esta llave puede crearse mediante una llamada a `ftok`. Si `key` toma el valor `IPC_PRIVATE`, la llamada crea un nuevo identificador sujeto a la disponibilidad de entradas libres en la tabla que gestiona el kernel para los semáforos.

`nsems` es el total de semáforos que van a estar agrupados bajo el identificador devuelto por `semget`.

`semflg` es una máscara de bits que indica el modo de adquisición del identificador. Si el bit `IPC_CREAT` está activo, la facilidad IPC se creará en el supuesto de que otro proceso no la haya creado ya. Si los bits `IPC_CREAT` e `IPC_EXCL` están activos simultáneamente, la llamada falla en el caso de que el conjunto de semáforos ya esté creado. Los 9 bits menos significativos de `semflg` indican los permisos del semáforo. Sus posibles valores son:

Valores	Significado
0400	Permiso de lectura para el usuario.
0200	Permiso de modificación para el usuario.
0060	Permiso de lectura y modificación para el grupo.
0006	Permiso de lectura y modificación para el resto de usuarios.

El identificador devuelto por `semget` es heredado por los procesos descendientes del actual.

Las siguientes líneas de código muestran cómo crear un nuevo identificador con 4 semáforos, asociado a la llave creada a partir del archivo "auxiliar" y la clave 'K'. Este identificador se va a crear con permisos de lectura y modificación para el usuario.

```
int llave, semid;
…
if ((llave = ftok("auxiliar", 'K')) == (key_t) -1) {
      /* error */
}
if ((semid = semget(llave, 4, IPC_CREAT | 0600)) == -1) {
      /* error */
}
```

9.2.3 CONTROL DE LAS ESTRUCTURAS DE SEMÁFORO – SEMCTL

Con `semctl` vamos a poder acceder a la información administrativa y de control de que dispone el kernel sobre un semáforo. La declaración de `semctl` es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
      int val;
      struct semid_ds *buf;
      ushort *array;
} arg;

int semctl(int semid, int semnum, int cmd, union semun arg);
```

`semctl` actuará sobre el conjunto de semáforos que responder al identificador `semid` devuelto por una llamada previa a `semget`.

`semnum` indica cuál es el semáforo, de los que hay bajo `semid`, al que queremos acceder.

La operación de control que se va a realizar sobre el semáforo viene indicada en `cmd`. Los siguientes son valores permitidos para `cmd`:

Valores	Significado
GETVAL	Se utiliza para leer el valor de un semáforo. El valor se devuelve a través del nombre de la función.
SETVAL	Permite inicializar un semáforo a un valor determinado que se especifica en `arg`.
GETPID	Se usa para leer el `PID` del último proceso que actuó sobre el semáforo. Este valor se devuelve a través del nombre de la función.
GETNCNT	Permite leer el número de procesos que hay esperando a que se incremente el valor del semáforo. Este número se devuelve a través del nombre de la función.
GETZCNT	Permite leer el número de procesos que hay esperando a que el semáforo tome el valor 0. Este número se devuelve a través del nombre de la función.

GETALL	Permite leer el valor de todos los semáforos asociados al identificador `semid`. Estos valores se almacenan en `arg`.
SETALL	Sirve para inicializar el valor de todos los semáforos asociados al identificador `semid`. Los valores de inicialización deben estar en arg.
IPC_STAT **IPC_SET**	Permiten leer y modificar la información administrativa asociada al identificador `semid`.
IPC_RMID	Le indica al kernel que debe borrar el conjunto de semáforos agrupados bajo el identificador `semid`. La operación de borrado no tendrá efecto mientras haya algún proceso que esté usando los semáforos.

La estructura `struct semid_ds` se define así:

```
struct semid_ds {
      struct ipc_perm sem_perm;      /* Permisos */
      time_t sem_otime; /* Fecha de la última operación semop. */
      time_t sem_ctime; /* Fecha del último cambio */
      struct sem *sem_base;   /* Apuntador al primer semáforo en
                                 el arreglo. */
      struct wait_queue *eventn;
      struct wait_queue *eventz;
      struct sem_undo *undo;  /* Petición de deshacer los
                                 cambios en el arreglo. */
      ushort sem_nsems;       /* Número de semáforos. */
};
```

Si la llamada a `semctl` se realiza satisfactoriamente, la función devuelve un número cuyo significado dependerá del valor de `cmd`. Así, son significativos los valores devueltos cuando `cmd` vale `GETVAL`, `GETZCNT` o `GETPID`. Para cualquier otro valor de `cmd`, la función devuelve 0 cuando se ejecuta satisfactoriamente. Si algo falla, la función devuelve -1.

A continuación mostramos una secuencia de código que crea un identificador con 4 semáforos asociados. Los 2 primeros se inicializan con el valor de 5 y los dos últimos con el valor de 8. Luego preguntamos por el valor del semáforo número 3.

```
int semid, valor;
ushort semarray[4];
…
/* Creación de los semáforos */
semid = semget(ftok("auxiliar", 'K'), 4, IPC_CREAT | 0600);
if (semid == - 1) {
      /* Tratamiento del error */
}
/* Inicialización de los semáforos */
semarray[0] = 5;
semarray[1] = 5;
semarray[2] = 8;
semarray[3] = 8;
semctl(semid, 3, GETVAL, 0);
```

9.2.4 OPERACIONES P Y V – SEMOP

Para utilizar las operaciones P y V con los semáforos de UNIX, tenemos que usar la llamada semop. Su declaración es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, int nsops);
```

semop realiza operaciones atómicas sobre los semáforos que hay asociados bajo el identificador semid.

sops es un apuntador a un arreglo de estructuras que indican las operaciones que vamos a llevar a cabo sobre los semáforos. nsops es el total de elementos que tiene el arreglo de operaciones. Cada elemento del arreglo es una estructura del tipo sembuf que se define como sigue:

```
struct sembuf {
      ushort sem_num;   /* Número del semáforo. */
      short sem_op;     /* Operación (incrementar o decrementar). */
      short sem_flg;    /* Máscara de bits. */
};
```

sem_num es el número del semáforo y se utiliza como índice para acceder a él. Su valor está en el rango de 0 a N-1, donde N es el total de semáforos que hay agrupados bajo el identificador.

sem_op es la operación a realizar sobre el semáforo especificado en sem_num. Si sem_op es negativo, el valor del semáforo se decrementará, lo que equivale a una operación P. Si sem_op es positivo, el valor del semáforo se incrementará, lo que equivale a una operación V. Si sem_op vale 0, el valor del semáforo no sufre ninguna alteración.

Las operaciones V siempre se ejecutan satisfactoriamente, ya que un semáforo puede tomar valores positivos; sin embargo, las operaciones P no siempre se pueden realizar. Si, por ejemplo, el semáforo tiene valor 2, no podemos restarle un número mayor que 2, porque el semáforo pasaría a tener valor negativo. Ante esta situación, semop responderá de diferentes formas, según el valor del campo sem_flg. Sus bits significativos son:

Valores	Significado
IPC_NOWAIT	La llamada a semop devuelve el control en caso de que no se pueda satisfacer la operación especificada en sem_op. La forma de trabajar por defecto es IPC_WAIT.
SEM_UNDO	La operación se deshace cuando el proceso termina.

El bit SEM_UNDO previene contra el bloqueo accidental de semáforos. Supongamos que un proceso decrementa el valor de un semáforo y termina de forma anormal (porque recibe una señal). Este semáforo quedará bloqueado para otros procesos. Esta solución previene activando el bit SEM_UNDO del campo sem_flg, ya que el kernel se encarga de actualizar el valor del semáforo, deshaciendo las operaciones realizadas sobre él, cuando termina el proceso.

Si la llamada a semop no se realiza con éxito, la función devuelve -1 y en errno estará el código producido.

Como ejemplo, vamos a ver la forma de realizar una operación P y otra V sobre el semáforo número 1 y 3 de un identificador que agrupa un total de 4 semáforos:

```
struct sembuf operaciones[4];
...
operaciones[0].sem_num = 1; /* semáforo número 1 */
operaciones[0].sem_op = 1; /* operación V */
operaciones[0].sem_flg = 0;
operaciones[1].sem_num = 3; /* semáforo número 3 */
operaciones[1].sem_op = -1; /* operación P */

operaciones[1].sem_flg = 0;

semop(semid, operaciones, 2);
...
```

9.2.5 EJEMPLO DE APLICACIÓN. SINCRONIZACIÓN DE PROCESOS

El ejemplo que vamos a implementar ilustra el uso de los semáforos para sincronizar dos procesos (padre e hijo). Pretendemos sincronizar los procesos para que se vaya ejecutando simultáneamente. Para conseguir esto, necesitamos al menos dos semáforos. Uno de los semáforos lo utiliza el padre para darle paso al hijo, y el otro lo utiliza el hijo para darle paso al padre. La secuencia de ejecución del padre es tomar su semáforo, realizar las operaciones que considere necesarias y abrir el semáforo del hijo cuando termine. Cuando el padre va a tomar otra vez su semáforo se da cuenta de que no puede abrir porque no lo ha levantado, por lo que el proceso padre se pone a esperar. El proceso hijo va a realizar algo parecido, tomando primero su semáforo y levantando luego el del padre. Mediante estos mecanismos conseguimos que los procesos se vayan pasando el turno y se ejecuten alternativamente. El código que ilustra eso es el siguiente:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_HIJO 0
#define SEM_PADRE 1

int main(int argc, char *argv[]) {
      int i = 10, semid, pid;
      struct sembuf operacion;
      key_t llave;

      /* Petición de un identificador con dos semáforos */
      llave = ftok(argv[0], 'K');
      if ((semid = semget(llave, 2, IPC_CREAT | 0600)) == -1) {
            perror("semget");
            return -1;
      }
      /* cerramos el semáforo del procesos hijo*/
      semctl(semid, SEM_HIJO, SETVAL, 0);
      /* abrimos el semáforo del proceso padre */
      semctl(semid, SEM_PADRE, SETVAL, 1);
```

```
	if ((pid = fork()) == -1) {
		perror("fork");
		return -1;
	} else if (pid == 0) {
		/* proceso hijo */
		while (i) {
			/* cerramos el semáforo del proceso hijo */
			operacion.sem_num = SEM_HIJO;
			operacion.sem_op = -1;
			operacion.sem_flg = 0;
			semop(semid, &operacion, 1);

			fprintf(stdout, "PROCESO HIJO: %d\n", i--);

			/* abrimos el semáforo del proceso padre */
			operacion.sem_num = SEM_PADRE;
			operacion.sem_op = 1;
			semop(semid, &operacion, 1);
		}
		/* borramos el semáforo */
		semctl(semid, 0, IPC_RMID, 0);
	} else {
		/* proceso padre */
		operacion.sem_flg = 0;
		while (i) {
			/* cerramos el semáforo del proceso padre */
			operacion.sem_num = SEM_PADRE;
			operacion.sem_op = -1;
			semop(semid, &operacion, 1);

			fprintf(stdout, "PROCESO PADRE: %d\n", i--);

			/* abrimos el semáforo del proceso hijo */
			operacion.sem_num = SEM_HIJO;
			operacion.sem_op = 1;
			semop(semid, &operacion, 1);
		}
		/* borramos el semáforo */
		semctl(semid, 0, IPC_RMID, 0);
	}
}
```

9.3 MEMORIA COMPARTIDA

La forma más rápida de comunicar dos procesos es hacer que compartan una zona de memoria. Para enviar datos de un proceso a otro, sólo hay que escribir en memoria y automáticamente esos datos están disponibles para que los lea cualquier otro proceso.

La memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtuales es local para ese proceso y cualquier intento de direccionar esa memoria desde otro proceso va a proporcionar una violación de segmento.

Para solucionar este problema, UNIX brinda la posibilidad de crear zonas de memoria con la característica de poder ser direccionadas por varios procesos simultáneamente. Esta memoria va a ser virtual, por lo que sus direcciones físicas asociadas podrán variar con el tiempo. Esto no va a plantear ningún problema, ya que los procesos no generan direcciones físicas, sino virtuales, y es el kernel el encargado de traducir de unas o otras.

Las llamadas para poder manipular la memoria compartida son: shmget, para crear una zona de memoria compartida o habilitar el acceso a una ya creada; shmctl, para acceder y modificar la información la información administrativa y de control que el kernel le asocia a cada zona de memoria compartida: shmat, para unir una zona de memoria compartida a un proceso, y shmdt, para separar una zona previamente unida.

9.3.1 PETICIÓN DE MEMORIA COMPARTIDA – SHMGET

Mediante `shmget` vamos a obtener un identificador con el que poder realizar futuras llamadas al sistema para controlar una zona de memoria compartida. Su declaración es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

`key` es una llave que tiene el mismo significado que vimos para la llamada `semget` (creación de semáforos). Esta es una característica que tienen todas las llamadas para crear mecanismos IPC.

`size` es el tamaño en bytes de la zona memoria que queremos crear.

`shmflg` es una máscara de bits que tiene el mismo significado que vimos en la máscara `semflg` de la función `semget`.

Si la llamada se ejecuta correctamente, devolverá el identificador (número entero no negativo) asociado a la zona de memoria. Si falla, devolverá el valor -1 y en `errno` estará el código del tipo de error producido.

El identificador devuelto por `shmget` es heredado por los procesos descendientes del actual.

Las siguientes líneas muestran cómo crear una zona de memoria de tamaño 4,096 bytes, donde sólo el usuario va a tener permisos de lectura y escritura:

```
int shmid;
...
if ((shmid = shmget(IPC_PRIVATE, 4096, IPC_CREAT | 0600)) == -1) {
      /* error */
}
```

9.3.2 CONTROL DE UNA ZONA DE MEMORIA COMPARTIDA – SHMCTL

Con `shmctl` podremos realizar operaciones de control sobre la zona de memoria previamente creada por una llamada a `shmget`. Su declaración es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, in cmd, struct shmid_ds *buf);
```

`shmid` es el identificador válido devuelto por una llamada previa a `shmget`.

`cmd` indica el tipo de operación de control a realizar. Sus posibles valores son:

Valores	Significado
IPC_STAT	Lee el estado de la estructura de control de la memoria y lo devuelve a través de la zona de memoria apuntada por `buf`.
IPC_SET	Inicializa algunos de los campos de la estructura de control de la memoria compartida. El valor de estos campos los toma de la estructura apuntada por `buf`.
IPC_RMID	Borra del sistema la memoria compartida identificada por `shmid`. Si el segmento de memoria está unido a varios procesos, el borrado no se hace efectivo hasta que todos los procesos liberan la memoria.
SHM_LOCK	Bloque en memoria del segmento identificado por `shmid`. Esto quiere decir que no se va a realizar intercambio sobre él. Sólo los procesos cuyo identificador de usuario efectivo (`EUID`) sea igual al del superusuario van a poder realizar esta operación.
SHM_UNLOCK	Desbloquea el segmento de memoria compartida, con lo que los mecanismos de intercambio van a poder trasladarlo de la memoria principal de la secundaria, y viceversa, cada vez que sea necesario. Sólo los procesos cuyo identificador de usuario efectivo (`EUID`) sea igual al del superusuario van a poder realizar esta operación.

La estructura `shmid_ds` se define como sigue:

```
struct shmid_ds {
      struct ipc_per shm_per; /* Estructura de permisos. */
      int sh_segsz;           /* Tamaño del area de memoria compartida */
      int pad1;               /* Usado por el sistema. */
      pid_d shm_lpid;         /* PID del proceso que hizo la última
                                 operación este segmento de memoria. */
      pid_d shm_cpid;         /*PID del proceso creador del segmento. */
      ushort shm_nattach;     /* Número de procesos unidos al segmento de
                                 memoria. */
      short pad2;             /* Usado por el sistema. */
      time_t shm_atime;       /* Fecha de la última unión al segmento de
                                 memoria. */
      time_t shm_dtime;       /* Fecha de la última separación del
                                 segmento de memoria. */
      time_t shm_ctime;       /* Fecha del último cambio en el segmento de
                                 memoria. */
};
```

La siguiente línea muestra cómo borrar del sistema una zona de memoria compartida:

```
shmctl(shmid, IPC_RMID, 0);
```

9.3.3 OPERACIONES CON LA MEMORIA COMPARTIDA – SHMAT Y SHMDT

Antes de usar una zona de memoria compartida, tenemos que asignarle un espacio de direcciones virtuales de nuestro proceso. Esto es lo que se conoce como unirse o atarse a un segmento de memoria compartida. Una vez que dejamos de usar un segmento de memoria, tenemos que desatarnos de él. Al realizar esta operación, el segmento deja de estar accesible para el proceso.

Las llamadas al sistema para realizar estas operaciones con `shmat` (para atar) y `shmdt` (para desatar), y sus declaraciones son:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg);
int shmdt(char *shmaddr);
```

`shmid` es el identificador de una zona de memoria creada mediante una llamada previa a `shmget`.

`shmaddr` es la dirección virtual donde queremos que empiece la zona de memoria compartida. Si la llamada a `shmat` funciona correctamente, devolverá un apuntador a la dirección virtual a la que está unido el segmento de memoria compartida. Esta dirección puede coincidir o no con `shmaddr`, dependiendo de la decisión que tome el kernel. Lo normal es que `shmaddr` valga 0, con lo cual se deja en manos del kernel la elección de la dirección de inicio. Si `shmaddr` no vale 0, el kernel intentará satisfacer la petición del usuario, pero no siempre consigue. En el caso de `shmdt`, shmaddr es la dirección virtual del segmento de memoria compartida que queremos separar del proceso.

`shmflg` es una máscara de bits que indica la forma de acceso a la memoria. Si el bit `SHM_RDONLY` está activo la memoria será accesible para leer, pero no para escribir.

Si las llamadas funciones correctamente, `shmat` devuelve la dirección a la que está unido el segmento de memoria compartida y `shmdt` devuelve 0. Si algo falla, ambas llamadas devuelven -1 y en `errno` estará el código del tipo de error producido.

Una vez que la memoria está unida al espacio de direcciones virtuales del proceso, el acceso a ella se realiza a través de apuntadores, como con cualquier memoria de datos asignada al programa.

A continuación mostramos la forma de crear una zona de memoria compartida en la que se va a almacenar un arreglo unidimensional de números reales:

```
#define MAX 10

int shmid, i;
float *arr;
key_t llave;
...
```

```
/* Creación de una llave. */
llave = ftok("prueba", 'K');

/* Petición de una zona de memoria compartida. */
shmid = shmget(llave, MAX * sizeof(float), IPC_CREAT | 0600);

/* Unión de la zona de memoria compartida a nuestro especio de
direcciones virtuales. */
arr = shmat(shmid, 0, 0);

/* Manipulación de la zona de memoria compartida. */
for (i = 0; i < MAX; i++) {
      arr[i] = i * i;
}

/* Separación de la zona de memoria compartida de nuestro espacio de
   Direcciones virtuales. */
shmdt(arr);

/* Borrado de la zona de memoria compartida. */
shmctl(shmid, IPC_RMID, 0);
```

9.3.2 EJEMPLO – MULTIPLICACIÓN EN PARALELO DE MATRICES

Como ejemplo, vamos a ver la implementación de un algoritmo para multiplicar dos matrices en paralelo.

El programa principal se va a encargar de leer dos matrices y comprobar si se pueden multiplicar. Acto seguido van a arrancar tantos procesos como le hayamos indicado en la línea de órdenes para multiplicar las matrices. Cada proceso se va a ocupar de generar un renglón de la matriz producto mientras queden renglones por generar. Naturalmente, las matrices que intervienen en la operación deben estar en memoria compartida. Para controlar el renglón que debe generar cada proceso, vamos a utilizar un semáforo que se inicializa en el total de renglones de la matriz producto y se va decrementando por cada renglón generado. El proceso principal se queda esperando a que todos los demás terminen para presentar el resultado.

Aunque en sistemas monoprocesador este método no resulta eficiente, es un buen ejercicio de sincronismo y comunicación entre procesos.

El código de este programa es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

typedef struct {
      int shmid;  /* identificador de la zona de memoria
                     compartida donde va estar la memoria. */
```

```
      int renglones, columnas;
      float **coef;     /* coeficientes de la matriz. */
} matriz;

matriz* crear_matriz(int renglones, int columnas) {
      int shmid, i;
      matriz *m;

      /* Petición de memoria compartida. */
      shmid = shmget(IPC_PRIVATE,
                  sizeof(matriz) + renglones * sizeof(float *) +
                  renglones * columnas * sizeof(float),
                  IPC_CREAT | 0600);
      if (shmid == -1) {
            perror("crear_matriz (shmget)");
            exit(-1);
      }
      /* Nos unimos al a memoria. */
      if ((m = (matriz *) shmat(shmid, 0, 0)) == (matriz *) - 1) {
            perror("crear_matriz (shmat)");
            exit(-1);
      }
      /* Inicialización de la matriz. */
      m->shmid = shmid;
      m->renglones = renglones;
      m->columnas = columnas;
      /* Le damos formato a la memoria para poder direccionar los
         coeficientes de la matriz. */
      m->coef = (float**)&m->coef + sizeof(float**);
      for (i = 0; i < renglones; i++) {
            m->coef[i] = (float*)&m->coef[renglones] +
                        i * columnas * sizeof(float);
      }
      return m;
}

matriz* leer_matriz() {
      int renglones, columnas, i, j;
      matriz *m;

      fprintf(stdout, "Renglones: ");
      fscanf(stdin, "%d", &renglones);
      fprintf(stdout, "Columnas: ");
      fscanf(stdin, "%d", &columnas);
      m = crear_matriz(renglones, columnas);
      for (i = 0; i < renglones; i++) {
            for (j = 0; j < columnas; j++) {
                  fscanf(stdin, "%f", &m->coef[i][j]);
            }
      }
```

```
	return m;
}

matriz* multiplicar_matrices(matriz *a, matriz *b, int numproc) {
	int p, semid, estado;
	matriz *c;

	if (a->columnas != b->renglones) {
		return NULL;
	}
	c = crear_matriz(a->renglones, b->columnas);

	/* Creación de dos semáforos. Uno de ellos se inicializa con el
	   total de renglones de la matriz producto. */
	semid = semget(IPC_PRIVATE, 2, IPC_CREAT | 0600);
	if (semid == -1) {
		perror("multiplicar_matrices (semget)");
		exit(-1);
	}
	semctl(semid, 0, SETVAL, 1);
	semctl(semid, 1, SETVAL, c->renglones + 1);

	/* Creación de tantos procesos como indique numproc. */
	for (p = 0; p < numproc; p++) {
		if (fork() == 0) {
			/* Código para los procesos hijo. */
			int i, j, k;
			struct sembuf operacion;

			operacion.sem_flg = SEM_UNDO;
			while (1) {
				/* Cada procesos hijo se encarga de
				   generar un renglón de la matriz
				   producto. Para saber qué columna tiene
				   que genera, consulta el valor del
				   semáforo. */
				/* Operación P sobre el semáforo 0. */
				operacion.sem_num = 0;
				operacion.sem_op = -1;
				semop(semid, &operacion, 1);
				/* Consultamos el valor del semáforo. */
				i = semctl(semid, 1, GETVAL, 0);
				if (i > 0) {
					/* Decrementamos el valor del
					   semáforo 1 en 1. */
					semctl(semid, 1, SETVAL, --i);
					/* Operación V sobre el semáforo
					   0 */
					operacion.sem_num = 0;
					operacion.sem_op = 1;
```

```
                        semop(semid, &operacion, 1);
                    } else {
                        exit (0);
                    }
                    /* Cálculo del renglon i-esimo de la
                       matriz producto. */
                    for (j = 0; j < c->columnas; j++) {
                        c->coef[i][j] = 0;
                        for (k = 0; k < a->columnas; k++) {
                            c->coef[i][j]  +=  a->coef[i][k]  *  b-
>coef[k][j];
                        }
                    }
                }
            }
        }
        /* Esperamos a que termine todos los procesos. */
        for (p = 0; p < numproc; p++) {
            wait(&estado);
        }
        /* Borramos el semáforo. */
        semctl(semid, 0, IPC_RMID, 0);
        return c;
}

void destruir_matriz(matriz *m) {
        shmctl(m->shmid, IPC_RMID, 0);
}

void imprimir_matriz(matriz *m) {
        int i, j;
        for (i = 0; i < m->renglones; i++) {
            for (j = 0; j < m->columnas; j++) {
                fprintf(stdout, "%g ", m->coef[j][j]);
            }
            fprintf(stdout, "\n");
        }
}

int main(int argc, char *argv[]) {
        int numproc;
        matriz *a, *b, *c;

        if (argc != 2) {
            numproc = 2;
        } else {
            numproc = atoi(argv[1]) + 1;
        }

        /* Lectura de las matrices. */
```

```
        a = leer_matriz();
        b = leer_matriz();
        /* Procesamiento de las matrices. */
        c = (matriz *) multiplicar_matrices(a, b, numproc);
        if (c != NULL) {
              imprimir_matriz(c);
        } else {
              fprintf(stderr, "Las matrices no se pueden multiplicar.");
        }
        destruir_matriz(a);
        destruir_matriz(b);
        destruir_matriz(c);
}
```

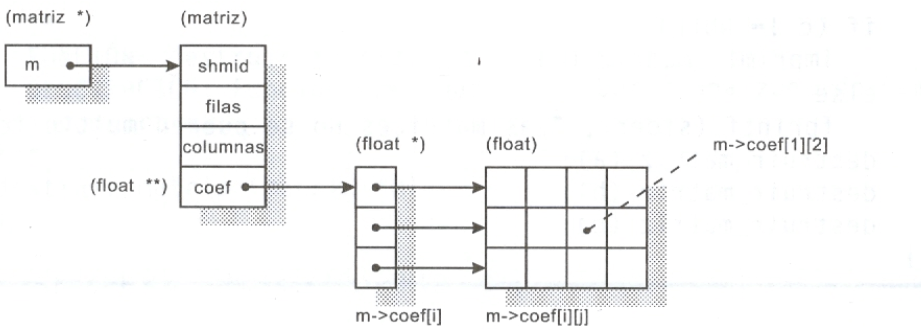
Es importante aclarar la forma de organizar la memoria asignada a una matriz, ya que puede haber quedado algo oscura en el código.

Recordemos que una matriz se define mediante la estructura:

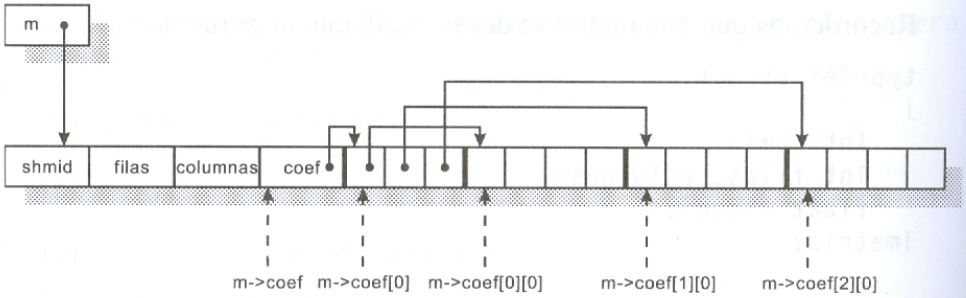
```
typedef struct {
        int shmid;  /* identificador de la zona de memoria
                       compartida donde va estar la memoria. */
        int renglones, columnas;
        float **coef;     /* coeficientes de la matriz. */
} matriz;
```

Las matrices se almacenan en zonas de memoria compartida para que puedan ser manejadas por varios procesos. Estas zonas se crean mediante una llamada a `shmget`, y mediante `shmat` podemos unirlas a un apuntador del programa. La memoria creada por `shmget` sólo cumple el requisito de estar alineada, por lo que tenemos que darle formato de una matriz.

Los campos `shmid`, renglones y columnas no plantean ningún problema, ya que son accesibles a través de un apuntador del tipo matriz. El problema lo van a plantear los coeficientes de la matriz, que se van a almacenar en el campo de `coef`. Este campo es un apuntador doublé y para que pueda utilizarse para indexar una matriz, debe responder a una organización como la reflejada en la figura (a). Esta es una representación gráfica para comprender el sentido de la doble indexación, pero la verdadera estructura se muestra en la figura (b).



a) Representación gráfica del uso de la doble indirección para indexar matrices.



b) Estructura real de una matriz.

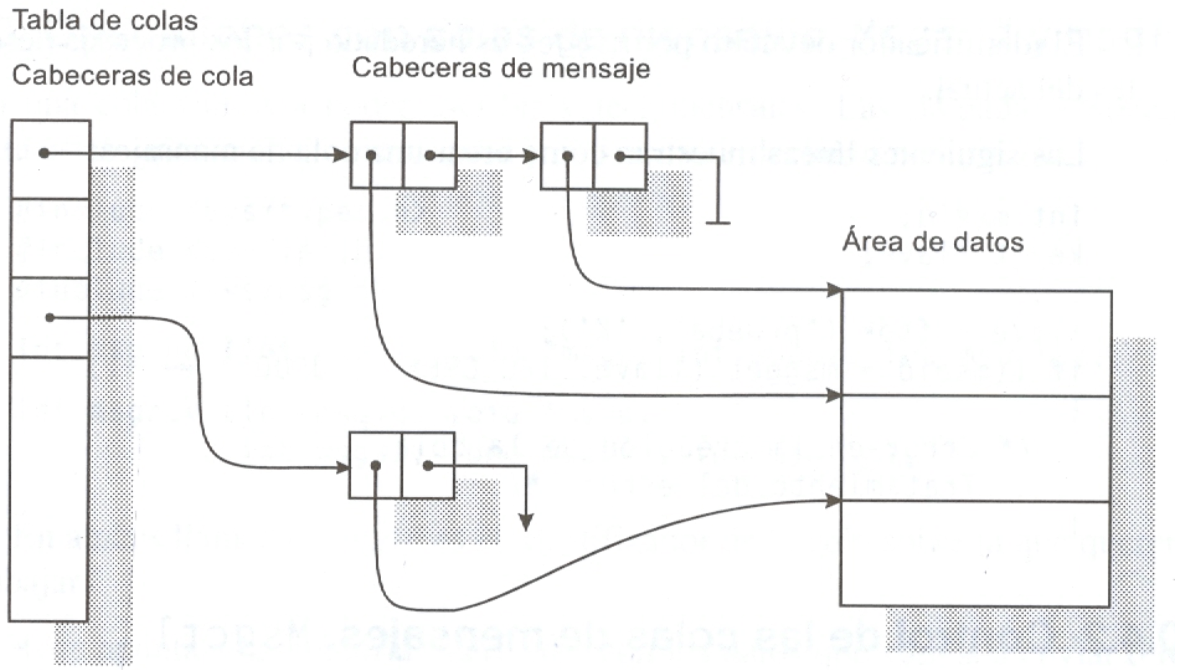
9.4 FILAS DE MENSAJES

Las filas de mensaje son otro de los mecanismos de comunicación entre procesos que brinda UNIX. Su filosofía es parecida a las tuberías, pero con una mayor versatilidad.

Una fila es una estructura de datos gestionada por el kernel, donde van a poder escribir varios procesos. Los mecanismos de sincronismo para que no se produzca colisión son responsabilidad del kernel. Los datos que se escriben en la fila deben tener formato de mensaje y son tratados como un todo indivisible.

A la vez, puede haber varios procesos leyendo de la fila. Cada operación de lectura va a sacar un mensaje. La fila se gestiona como un mecanismo FIFO, donde el primer mensaje que entra es el primero que sale. Para dotar de más flexibilidad a las filas, se pueden hacer peticiones de lectura para extraer un mensaje determinada, con lo que se rompe la gestión de tipo FIFO, aunque se sigue manteniendo para todos aquellos mensajes que son de un mismo tipo. Esta clasificación de los mensajes por tipos permite distinguir cuáles son los menajes que van destinados a cada uno de los procesos lectores. También se pueden hacer operaciones de lectura sin especificar el tipo de mensaje. Esto fuerza a que se lea al primer mensaje que haya en la fila.

La siguiente figura muestra la estructura que tienen las filas de mensajes.



Las llamadas que se emplean para manipular filas son `msgget`, para crear una fila o habilitar el acceso a una ya existente; `msgctl`, para acceder y modificar la información administrativa y de control que el kernel le asocia a cada fila de mensajes; `msgsnd`, para escribir un mensaje en la fila, y `msgrcv`, para sacer un mensaje de la fila.

9.4.1 PETICIÓN DE UNA FILA DE MENSAJES – MSGGET

La llamada que permite crear una fila de mensajes es `msgget` y su declaración es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
```

Si la llamada funciona correctamente, devuelve un identificador de la fila creada. En caso contrario, devuelve el valor -1 y en `errno` estará el código del error producido.

`key` es una llave que tiene el significado ya visto para los semáforos y la memoria compartida.

`msgflg` es un mapa de bits con el significado visto para los parámetros `semflg` y `shmflg` de las llamadas de creación de semáforos y memoria compartida.

El identificador devuelto por `msgget` es heredado por los procesos descendientes del actual.

Las siguientes líneas muestran cómo crear una fila de mensajes:

```
int msqid;
key_t llave;
...
```

```
llave = ftok("prueba", 'K');
if ((msqid = msgget(llave, IPC_CREAT | 0600) == - 1) {
      /* error */
}
```

9.4.2 CONTROL DE LAS FILAS DE MENSAJES – MSGCTL

La llamada para leer y modificar la información estadística y de control de una fila es, `msgctl` y su declaración es:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

`msqid` es un identificador creado mediante una llamada previa a `msgget`, e indica la fila sobre la que vamos a actuar.

`cmd` es el tipo de operación que queremos llevar a cabo. Sus posibles valores son:

Valores	Significado
IPC_STAT	Lee el estado de la estructura de control de la fila y lo devuelve a través de la zona de memoria apuntada por `buf`.
IPC_SET	Inicializa algunos de los campos de la estructura de control de la fila. El valor de estos campos los toma de la estructura apuntada por `buf`.
IPC_RMID	Borra del sistema la fila identificada por `smqid`. Si la fila está siendo usada por otros procesos, el borrado no se hace efectivo hasta que todos los procesos terminan su ejecución.

A continuación podemos ver una línea de código que borra una fila de mensajes:

```
msgctl(msqid, IPC_RMID, 0);
```

9.4.3 OPERACIONES CON FILAS DE MENSAJES – MSGSND Y MSGRCV

En una fila vamos a poder escribir y leer mensajes. Las llamadas respectivas para estas operaciones son: `msgsnd` y `msgrcv` y se declaran como sigue:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, void *msgp, int msgsz, int msgflg);
int msgrcv (int msqid, long msgtyp, int msgflg);
```

En ambas llamadas, `msqid` es el identificador de la fila sobre la que queremos trabajar.

`msgp` apunta a la memoria intermedia con los datos que vamos a enviar o recibir. La composición de esta memoria la define el usuario. Sólo es obligatorio que el primer campo sea de tipo long y se utiliza para identificar el tipo de mensaje. No existen tipos definidos por el sistema; por tanto, la clasificación de los mensajes depende del programador.

`msgsz` es el tamaño, en bytes, del mensaje que queremos enviar o recibir. En este tamaño no se incluyen los bytes que ocupa el campo "tipo de mensaje".

`msgtyp` sólo aparece en la llamada de lectura y especifica el tipo de mensaje que queremos leer. Puede tomar los siguientes valores:

Valores	Significado
msgtyp = 0	Leer el primer mensaje que haya en la fila.
msgtyp > 0	Leer el primer mensajes de tipo `msgtyp` que haya en la fila.
msgtyp < 0	Leer el primer mensaje que cumpla que su tipo sea menor o igual al valor absoluto del `msgtyp` y a la vez sea el más pequeño de los que hay.

`msgflg` es un mapa de bits y tiene distintos significados según aparezca en la llamada `msgsnd` o `smgrcv`. Para la llamada `msgsnd` (escritura en la fila), si la fila está llena:

- Si el bit `IPC_NOWAIT` está activo, la llamada devuelve el control inmediatamente y retorna el valor -1.
- Si el bit `IPC_NOWAIT` no está activo, el proceso suspende su ejecución hasta que haya espacio libre en la fila.

Para la llamada `msgrcv` (lectura de la fila), si no hay ningún mensaje del tipo especificado:

- Si el bit `IPC_NOWAIT` está activo, la llamada devuelve el control inmediatamente y retorna el valor -1.
- Si el bit `IPC_NOWAIT` no está activo, el proceso suspende su ejecución en espera de que haya un mensaje del tipo deseado.

Si se ejecutan satisfactoriamente, `msgsnd` devuelve el valor de 0 y `msgrcv` el total de bytes que realmente ha escrito en la memoria intermedia referenciada por `msgp`. Este tamaño no incluye los bytes que ocupa el campo "tipo de mensaje".

Si las llamadas fallas durante su ejecución, devuelven el valor -1 y en `errno` estará el código del error producido.

A continuación mostramos las líneas de código necesarias para enviar y recibir mensajes del tipo 1, que se compone de una cadena de 20 caracteres.

```
int msqid;
struct {
      long tipo;
      char cadena[20];
} mensaje;
int longitud = sizeof(mensaje) – sizeof(mensaje.tipo);
...
/* envío del mensaje */
mensaje.tipo = 1;
strcpy(mensaje.cadena, "HOLA");
if (msgsnd(msqid, &mensaje, longitud, 0) == -1) {
      /* error */
}
```

```
/* recepción del mensaje */
if (msgrcv(msqid, &mensaje, longitud, 1, 0) == -1) {
      /* error */
}
```

9.5 THREADS

En este tema veremos cómo usar múltiples hilos de control (threads) para realizar varias tareas dentro del ambiente de un solo proceso. Todos los hilos que crea un proceso tienen acceso a los recursos del mismo, como los descriptores de archivo y memoria.

9.5.1 CONCEPTO DE THREADS

Un típico proceso de UNIX puede ser visto como un solo hilo de control: cada proceso es haciendo sólo una cosa a la vez. Con múltiples hilos de control, nosotros podemos diseñar nuestros programas para hacer más de una cosa a la vez dentro de un solo proceso, con cada hilo manejando una tarea separada. Esta técnica tiene muchos beneficios:

- Podemos simplificar el código que trabaja con eventos asincrónicos asignados un hilo separado para manejar un evento determinado. Cada hilo puede manejar este evento usando un esquema de programación síncrono. Un esquema de programa síncrono es mucho más simple que uno asíncrono.
- Múltiples procesos usan mecanismos complejos proveídos por el sistema operativo para compartir memoria y descriptores de archivos. Los threads, por otro lado, automáticamente tienen acceso a los mismos espacios de memoria y descriptores de archivo.
- Algunos problemas pueden ser divididos de tal forma que el programa final puede ser mejorado. Un proceso sencillo que tiene múltiples tareas a realizar implícitamente las serializa, porque solo hay un hilo de control. Con varios hilos de control, el proceso de tareas independientes puede ser entrelazadas asignado una tarea separada a cada thread. Dos tareas pueden ser entrelazadas solo si ellas no dependen entre sí.
- De manera similar, programas interactivos pueden mejorar significativamente su tiempo de respuesta usando threads para separar aquellas porciones del programa que interactúan con el usuario del resto del programa.

Algunas gentes asocian la programación multihilos con sistemas multiprocesadores. Los beneficios de un modelo de programación mutihilos puede ser obtenidos incluso si nuestro programa se encuentra corriendo es una computadora de un solo procesador. Un programa puede ser simplificado usando threads sin importar el número de procesadores, porque el número de procesadores no afecta la estructura del programa. Además, aunque tu programa sea bloqueado a causa alguna tarea que está realizando, todavía podemos ver mejora en el tiempo de respuesta, ya que es posible que alguno de los threads puede estar corriendo cuando otros están bloqueados.

Un thread también guarda información relacionada a la tarea que está ejecutando. Esto incluye un identificador del thread dentro del esquema del proceso, un conjunto de registros, una pila, políticas y prioridad de ejecución, una máscara de señales, una variable errno e información específica del thead. Todo lo que existe dentro de un proceso es compartido entre los threads del mismo, incluyendo el texto del programa ejecutable, la memoria global y heap, las pilas y los descriptores de archivos.

9.5.1 CREACIÓN DE UN THREAD

La llamada `pthread_create` nos permite crear un thread y su definición es la siguiente:

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tipd,
      const thread_attr_t *restrict attr, void (*start_run) (void*),
      void *restrict arg);
```

`tipd` es un apuntador a la localidad de memoria en donde se encuentra el id que le fue asignado al proceso que acabamos de crear. `attr` es usado para especificar varios atributos del thread. El valor de estos atributos los veremos más adelante, por el momento estableceremos este valor en `NULL`.

`start_rtn` es la función que estará ejecutando el thread en cuanto sea creado. La función solo puede recibir un apuntado a un tipo de dato `void`. En caso de que sea necesario recibir algún parámetro de entrada, éste puede ser pasado a la función a través de `arg` que es un apuntador a un tipo de dato `void`.

Cuando un thread es creado, no existe garantía de cual se ejecute primero: el thread recientemente creado o el proceso que lo creo. Este thread tiene acceso al espacio de memoria del proceso y hereda las variables de ambiente y máscara de señales del proceso que lo generó; sin embargo, el conjunto de señales pendientes para ese thread empieza vacío.

Si la llamada se ejecuta correctamente, regresa cero. En caso de que exista un error, regresa el código del error generado.

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

pthread_t ntid;

void printids(const char *txt) {
      pid_t pid;
      pthread_t tid;

      pid = getpid();
      tid = pthread_self();
      fprintf(stdout, "%s pid %d tid %u (0x%x)\n", txt, pid, (unsigned
int) tid, (unsigned int) tid);
}

void* thr_fn(void *arg) {
      printids("new thread: ");
      return ((void *) 0);
}

int main(int argc, char *argv[]) {
      int err;
```

```
        err = pthread_create(&ntid, NULL, thr_fn, NULL);
        if (err != 0) {
                fprintf(stderr, "no se puede crear el thread: %s\n", (char *)
strerror(err));
                return -1;
        }
        printids("main thread:");
        sleep(1);
        exit(0);
}
```

En este ejemplo tiene dos peculiaridades. La primera es que es necesario poner a dormir el proceso principal. Si no se pone a dormir, el proceso puede terminar antes de que el thread tenga algún chance de correr. Este comportamiento depende de la implementación de threads que haga el sistema operativo, así como de los algoritmos de calendarización.

La segunda peculiaridad es que el nuevo thread obtiene su identificador llamando al método `pthread_self` en lugar de leerlo de la memoria compartida o de obtenerlo como resultado de una función que inicia el thread.

Para compilar correctamente este código es necesario usar el parámetro `-pthread`:

```
$ gcc thread1.c -phtread
```

9.5.3 IDENTIFICACIÓN DE UN THREAD

Un thread, al igual que un proceso, tiene un identificador. A diferencia del identificador de un proceso, el cual es único en todo el sistema, el identificador de un thread solo tiene significado dentro del contexto del proceso al que pertenece.

Un thread puede obtener su identificador llamando a la función `pthread_self`. Su definición es la siguiente:

```
#include <phtread.h>
pthread_t pthread_self(void);
```

Algunas veces será necesario comparar los identificadores de procesos y para esto tenemos la función:

```
#include <phtread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Que regresa un valor diferente de cero si es ambos identificadores son iguales. En otro caso, devuelve 0.

9.5.4 TERMINACIÓN DE UN THREAD

Si cualquier thread dentro un proceso hace una llamada a `exit`, `_Exit`, `_exit`, entonces el proceso entero termina. De manera similar, cuando la acción por omisión es terminar el proceso, una señal es enviada al thread para terminarlo.

Un thread sencillo puede salir de tres maneras, parando el flujo de control, sin terminar el proceso entero:

1. Un thread termina de la ejecución de su rutina. Regresa el código de salida.
2. El thread puede ser cancelado por otro thread del mismo proceso.
3. El thread puede ejecutar la función `pthread_exit`.

```
#include <phtread.h>
void pthread_exit(void *rval_ptr);
```

`rval_ptr` es un apuntador, similar al argumento que recibe `start_routine`. Este apuntador esta disponibles a otros threads del proceso a través de la llamada `pthread_join`.

```
#include <phtread.h>
int pthread_join(pthread_t tid, void **rval_ptr);
```

El proceso que invoca la llamada se bloquea a que el thread específico termina, ya sea a través de invocar la llamada `pthread_exit`, termine la ejecución de `start_routine` o si es cancelado. Si el proceso termina la ejecución de `start_routine`, `rval_ptr` contendrá el valor de retorno. Si el proceso es cancelado, el apuntador hará referencia a `PTHREAD_CANCELED`.

Con la llamada pthread_join, automáticamente ponemos al thread es un estado que nos permite recuperar sus recursos. Si el thread ya se encuentra en este punto la llamada devolverá EINVAL.

Si no estamos interesados en obtener el valor de terminación del thread, simplemente ponemos NULL en rval_ptr.

En el siguiente código podemos ver a dos threads. El primero terminará a través una llamada pthread_exit, mientras que el segundo se termina por el fin de ejecución de la función start_routine.