Comunicación entre procesos

Comunicación entre procesos

Índice

Procesos

Introducción

Muchas veces suele utilizarse erróneamente los conceptos de proceso y programa, por lo que comenzaremos este apunte llamando proceso a la ejecución de un programa. Es decir, el programa cargado en memoria y corriendo.

Un proceso posee un conjunto de registros que utiliza, tiene un contador de programa que indica la siguiente instrucción que debe ejecutar, y tiene una pila. Esto significa que tiene un flujo de control que ejecuta una instrucción tras otra.

Cuando una computadora posee varios procesadores, es posible ejecutar múltiples programas en paralelo, es decir, al mismo tiempo. En este caso son procesos concurrentes ya que se ejecutan en simultáneo. Si nuestra computadora posee un solo procesador, cada proceso "cree" que dispone de todos los recursos para sí mismo, sucede así con cada uno de los procesos que estén corriendo, es por ello que en realidad hablaremos de una ejecución pseudo-paralela.

Al crearse un proceso el sistema operativo le asigna una identificación que llamaremos ***"PID"*** (identificación del proceso).

Clasificación de procesos

- *Ejecución en 1er plano:* se trata de un proceso iniciado por el usuario o interactivo.
- *Ejecución en 2o plano:* se trata de un proceso no interactivo que no necesita ser iniciado por el usuario.
- *Demonio:* es un proceso en 2do plano siempre disponible, que da servicio a varias tareas (debe ser propiedad del usuario root).

Estados de un proceso

- *Proceso zombi:* se trata de un proceso detenido que queda en la tabla de procesos hasta que finalice su padre. Este hecho se produce cuando el proceso padre no recoge el código de salida del proceso hijo.
- *Proceso huérfano:* proceso en ejecución cuyo padre ha finalizado. El nuevo identificador de proceso padre (PPID) coincide con el identificador del proceso init (1).

Creación de un proceso (fork)

Para crear un nuevo proceso, hacemos uso de la llamada ***fork().*** Esta invocación crea una copia casi idéntica del proceso padre (se copia todo el código) y continúan ejecutándose en paralelo.

El proceso padre recibe de fork() la identificación de su hijo (pid – process id-), mientras que el hijo recibe el valor 0. El hijo hereda los recursos del padre (archivos abiertos, estado de las variables, entre otros).

El prototipo de la función es: **size_t fork(void);**

```
int main()
{
 pid_t idProceso;
 int variable = 1;
 int estadoHijo;

 idProceso = fork();

 if (idProceso == -1)
 {
      perror ("No se pudo crear el proceso");
        exit (-1);
 }

 if (idProceso == 0)
 {
      // soy el hijo
      printf ("Soy el hijo: mi pid es %d ",getpid());
      printf ("Soy el hijo: mi padre es %d ",getppid());

 }
 else
 {
        // soy el padre
      printf ("Soy el padre: mi pid es %d ",getpid());
      printf ("Soy el padre: mi padre es %d ",getppid());
      wait (&estadoHijo);
 }

 return 0 ;
}
```

Threads

Sabemos que un proceso es cualquier programa en ejecución y es totalmente independiente de otros procesos. Un proceso tiene su propia zona de memoria y se ejecuta "simultáneamente" a otros procesos.

Dentro de un proceso puede haber uno o varios hilos de ejecución (threads). Es decir, que un proceso podría estar haciendo varias cosas "a la vez". Los hilos o threads dentro de un proceso comparten todos, la misma memoria. Por lo tanto, si un hilo modifica una variable, todos los demás hilos del mismo proceso verán el nuevo valor de la variable. Es por esta razón que surgen mecanismos de protección o sincronización para la comunicación entre procesos.

Creación de un thread

```
#include <pthread.h>
#include <stdio.h>
void* mostrarX(void *);

int main ()
{
pthread_t thread_id; //crea la variable thread de tipo pid thread

//imprime el id del proceso padre por la salida estandar.
fprintf (stdout, "El pid del thread principal es %d\n", (int) getpid ());

/* realiza la llamada para crear elthread, el cual comenzara su ejecución
con la función    thread_function */
pthread_create(&thread_id,NULL,&mostrarX,NULL);
while(1)
    fputc('o',stdout);
return ;
}

void* mostrarX(void *sinUso)
{
  while(1)
    fputc('x',stdout);
  return NULL;

}
```

Analizando un poco más en detalle la función de creación, vemos que posee cuatro argumentos:

1. Un puntero a la variable de tipo pthread_t a donde dejará el pthread del thread creado.
2. Un puntero al objeto de atributos del thread recién creado, en este caso NULL.
3. Un puntero a la función a ejecutar por el thread. Esta función tiene que ser de tipo void* y un argumento de tipo void*.
4. Los argumentos (de tipo void*) a pasarle al thread recién creado

Herramientas de sincronización

Semáforos

Muchas veces es necesario que dos o más procesos o hilos (threads) accedan a un recurso común, por ejemplo, leer una zona de memoria, mostrar información en pantalla, escribir un archivo, etc. Para ello debemos utilizar herramientas de sincronización que permitan que los procesos trabajen en forma ordenada.

Un semáforo da acceso al recurso a uno de los procesos y se lo niega a los demás mientras el primero no termine. Los semáforos, junto con la memoria compartida y las colas de mensajes, son los recursos compartidos que suministra Linux para comunicación entre procesos. El funcionamiento del semáforo podemos asociarlo a un “flag” o “bandera”, es decir, recursos que hemos utilizado previamente en otros programas de la carrera.

Utilizaremos un semáforo para controlar el acceso a un archivo por parte de dos procesos. Imaginemos un semáforo que inicialmente tiene el valor 1 (está en "verde"). Cuando un proceso quiere acceder al archivo, primero debe decrementar el semáforo. El contador queda en valor 0 y como no es negativo, deja que el proceso siga su ejecución y, por tanto, acceda al recurso, en este caso, el archivo.

Ahora un segundo proceso lo intenta acceder, y para ello también decrementa el contador. Ahora el contador toma el valor (-1) y como es negativo, el semáforo se encarga de que el proceso quede "bloqueado" en una cola de espera. Este segundo proceso no continuará por tanto su ejecución y no accederá al recurso.

Supongamos ahora que el primer proceso termina de escribir el archivo. Al finalizar debe incrementar el contador del semáforo. Al hacerlo, este contador se pone a 0. Como no es negativo, el semáforo se encarga de mirar en la cola de procesos pendientes y "desbloquear" al primer proceso de dicha cola. Con ello, el segundo proceso que quería acceder al archivo continua su ejecución y accede al recurso.

Cuando este proceso también termine con el archivo, incrementa el contador y el semáforo vuelve a ponerse a 1, es decir, a estar en "verde".

Es posible hacer que el valor inicial del semáforo sea, por ejemplo, 3, con lo que pasarán los tres primeros procesos que lo intenten. Pueden a su vez quedar muchos procesos encolados simultáneamente, con lo que el contador quedará con un valor negativo grande. Cada vez que un proceso incremente el contador (libere el recurso común), el primer proceso encolado despertará. Los demás seguirán dormidos.

Mecanismos de comunicación entre procesos

Señales

Una señal es un "aviso" o “notificación” que puede enviar un proceso a otro. El sistema operativo se encarga de que el proceso que recibe la señal la trate inmediatamente. El sistema operativo envía señales a los procesos en determinadas circunstancias. Por ejemplo, si en el programa que se está ejecutando en un Shell, al presionar “ctrl-c”, estamos enviando una señal de terminación al proceso. Este la trata inmediatamente y sale. Si nuestro programa intenta acceder a una memoria no válida (por ejemplo, accediendo al contenido de un puntero a NULL), el sistema operativo detecta esta circunstancia y le envía una señal de terminación inmediata, con lo que el programa finaliza en forma anormal. Las señales se identifican por un número entero.

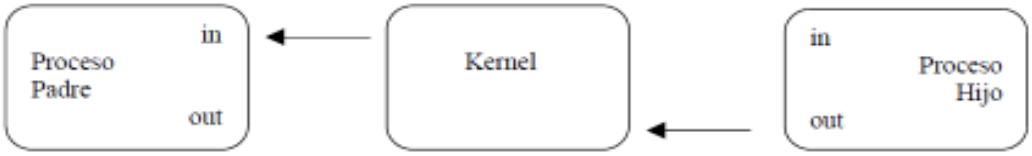
En síntesis, una señal es un evento que debe ser procesado y que puede interrumpir el flujo normal de un programa. Una señal puede ser capturada y asociarse con una función que procesa el evento ocurrido. Una señal también puede ignorarse, en ese caso el evento no interrumpe el flujo del programa. Las señales SIGINT y SIGSTOP no pueden ser ignoradas.

```
/*
      Este programa busca contar el número de CTRL-C que se realizan en
      un período de 15 segundos.
      Atención: este programa hace uso de variables globales!
*/

#include <stdlib.h>
#include <signal.h>

int numcortes=0;        /* Contador de CTRL-C */
int bucle=1;            /* Controlador de salida del bucle de espera */

void alarma (void);     /* Captura la señal de alarma SIGALRM */
void cortar (void);     /* Captura la señal de interrupción SIGINT */

int main ()
{
    signal (SIGINT, cortar);
    signal (SIGALRM, alarma);
    printf ("Ejemplo de signal.\n");
    printf ("Pulsa varias veces CTRL-C durante 15 segundos.\n");
    alarm (15);

    while (bucle);
    signal (SIGINT, SIG_IGN);
    printf ("Intentaste finalizar %d veces.\n", numcortes);
    printf ("Chau!!!.\n");

    return 0;
}
```

```
void alarma ()
{
     signal (SIGALRM, SIG_IGN);
     bucle=0;          /* Salir del bucle */
     printf ("¡Alarma!\n");
}


void cortar ()
{
     signal (SIGINT, SIG_IGN);
     printf ("Has pulsado CTRL-C\n");
     numcortes++;
     signal (SIGINT, cortar);
}
```

Pipes

Los pipes o también llamadas “tuberías sin nombre” es la forma más simple de comunicar dos procesos, se trata de una comunicación unidireccional. Es un método de conexión que une la salida estándar de un proceso con la entrada estándar de otro. Para establecer la comunicación se utilizan “descriptores de archivos”. Un descriptor se utiliza para obtener el camino de entrada a la tubería (write) mientras que el otro se utiliza para el camino de salida (read).




```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
int main()
{
      int fd[2];
      char buffer[30];

      /* pipe() devuelve 0 en caso de éxito o –1 en caso de error */
      if (pipe(fd) == -1)
      {
            perror("pipe");
            exit(1);
      }

      printf("Escribiendo en el descriptor de archivo #%d\n", fd[1]);
      write(fd[1], "prueba", 5);

      printf("Leyendo desde el descriptor de archivo #%d\n", fd[0]);
      read(fd[0], buffer, 5);
      printf("Leído \"%s\"\n", buffer);

      return 0;
}
```

Después de la invocación a ***pipe*** el arreglo de enteros “p” contiene en p[0] un descriptor de archivo para leer, y en p[1] un descriptor de archivo para escribir. Esto quiere decir que los pipes funcionan en una sola dirección.

Named pipes / FIFOS

Se lo conoce como FIFO o “tuberías con nombre”. El nombre es el de un archivo que múltiples procesos abrir, leer, escribir. Los named pipes se los encuentra como un archivo. Los procesos de diferentes padres pueden compartir datos mediante este mecanismo. Una vez finalizadas todas las operaciones el archivo permanece en el sistema de archivos.

Creación de una FIFO

- Desde el Shell
  - mknod MIFIFO p
  - mkfifo a=rw MIFIFO

- En nuestro código en C
  - mknod("/tmp/MIFIFO", S_IFIFO|0644 , 0);

Operaciones con FIFO

Las operaciones son prácticamente las mismas que con un pipe solo que debemos abrirlo y luego cerrarlo. Podemos tratar al named pipe como si fuese un stream, recordemos que en Linux “todo es un archivo”.

#define FIFO_FILE "MIFIFO“

A continuación se presentan dos programas que hacen uso de una FIFO

Programa A - Server

```
int main(void)
{
        FILE *fp;
        char readbuf[80];
        /* Crea el FIFO si no existe */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Cadena recibida: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

Programa B – Cliente

```
int main(int argc, char *argv[])
{
        FILE *fp;
        if ( argc != 2 ) {
                printf("USO: fifoclient [cadena]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);
        fclose(fp);

        return(0);
}
```

Cola de mensajes (message queues)

Un proceso puede crear una nueva cola de mensajes, o se puede conectar a una cola ya existente. De esta forma, dos procesos pueden compartir información. Una vez que se crea una cola de mensajes, esta no desaparece hasta que se destruya.

En C es posible hacer que dos procesos sean capaces de enviarse mensajes (estructuras de datos) y de esta forma pueden intercambiar información. Los procesos introducen mensajes en la cola y se van almacenando en ella. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo y dicho mensaje se borra de la cola.

También es posible hacer distintos "tipos" de mensajes, de forma que cada tipo de mensaje contiene una información asociada y se identifica por un número entero. Por ejemplo, los mensajes de ***tipo 1*** pueden contener el saldo de una cuenta de banco y el número de dicha cuenta, los de ***tipo 2*** pueden contener el nombre de una sucursal bancaria y su dirección.

De esta forma los procesos pueden retirar mensajes de la cola selectivamente por su tipo. Si un proceso sólo está "interesado" en saldos de cuentas, extraerá únicamente mensajes de ***tipo 1***.

Para poder trabajar con cola de mensajes necesitamos varias cosas:

- Una clave de tipo **key_t**, que sea común para todos los procesos que quieran compartir la cola de mensajes.
  Para ello disponemos de la función key_t ftok (char *, int).
  A dicha función se le pasa un archivo que exista y sea accesible por los procesos y un número entero. Con ellos construye una clave que nos devuelve. Si todos los programas utilizan el mismo archivo y el mismo entero, obtendrán la misma clave.

- Una vez que disponemos de la clave, se crea la cola de mensajes.
  Para ello está la función int msgget (key_t, int).
  Con dicha función creamos la cola y nos devuelve un identificador para la misma.
  El primer parámetro es la clave key_t obtenida anteriormente y que debería ser la misma para todos los programas.
  El segundo parámetro son flags que responden a permisos de lectura/escritura/ejecución.

Insertar datos en la cola

Para colocar un mensaje en la cola, disponemos de la función:
msgsnd (int, struct msgbuf *, int, int) ;

-El primer parámetro corresponde a un valor entero, que es el identificador de la cola obtenido con msgget().

-El segundo parámetro es el mensaje en sí. El mensaje debe ser una estructura cuyo primer campo sea un long. En dicho long se almacena el tipo de mensaje. Al pasar el mensaje como parámetro, se pasa un puntero al mensaje y es necesario realizar un "casteo" a struct msgbuf *.

-El tercer parámetro es el tamaño en bytes del mensaje exceptuando el long, es decir, el tamaño en bytes de los campos con la información.

-El cuarto parámetro son flags. Aunque hay varias opciones, la más habitual es poner un 0 o bien IPC_NOWAIT. En el primer caso la llamada a la función queda bloqueada hasta que se pueda enviar el mensaje. En el segundo caso, si el mensaje no se puede enviar, se vuelve inmediatamente con un error. El motivo habitual para que el mensaje no se pueda enviar es que la cola de mensajes esté llena.

Quitar datos de la cola

Para recoger un mensaje de la cola se utiliza la función:
msgrcv (int, struct msgbuf *, int, int, int) ;

-El primer parámetro es el identificador de la cola obtenido con msgget().

-El segundo parámetro es un puntero a la estructura donde se desea recoger el mensaje. Puede ser, como en la función anterior, cualquier estructura cuyo primer campo sea un long para el tipo de mensaje.

-El tercer parámetro entero es el tamaño de la estructura.

-El cuarto parámetro entero es el tipo de mensaje que se quiere retirar. Se puede indicar un entero positivo para un tipo concreto o un 0 para cualquier tipo de mensaje.

-El quinto parámetro son flags, que habitualmente puede ser 0 o bien IPC_NOWAIT. En el primer caso, la llamada a la función se queda bloqueada hasta que haya un mensaje del tipo indicado. En el segundo caso, se vuelve inmediatamente con un error si no hay mensaje de dicho tipo en la cola.

Liberar la cola

Una vez terminada de usar la cola, se debe liberar. Para ello se utiliza la función:
msgctl (int, int, struct msqid_ds *) ;

-El primer parámetro es el identificador de la cola de mensajes, obtenido con msgget().

-El segundo parámetro es el comando que se desea ejecutar sobre la cola, en este caso IPC_RMID.

-El tercer parametro son datos necesarios para el comando que se quiera ejecutar. En este caso no se necesitan datos y se pasará un NULL.

A continuación encontrarán dos programas, donde el primero **cola1.c** abre la cola de mensajes, envía un mensaje de tipo 1 para que lo lea cola2 y espera un mensaje de tipo 2. Cuando llega el mensaje de tipo 2, cola1 destruye la cola de mensajes. Por su parte, **cola2.c** abre la cola mensajes y espera un mensaje tipo 1. Cuando lo recibe, envía un mensaje de tipo 2 y se sale. Deja la destrucción de la cola para cola1.

cola1.c

```
typedef struct Mi_Tipo_Mensaje
{
      long Id_Mensaje;
      int Dato_Numerico;
      char Mensaje[10];
};

int main()
{
      key_t Clave1;
      int Id_Cola_Mensajes;
      Mi_Tipo_Mensaje Un_Mensaje;

      /*
      Igual que en cualquier recurso compartido (memoria compartida,
      semaforos o colas) se obtiene una clave a partir de un archivo
      existente cualquiera y de un entero cualquiera. Todos los procesos
      que quieran compartir este semaforo, deben usar el mismo archivo y
      el mismo entero.
      */

      Clave1 = ftok ("/bin/ls", 33);
      if (Clave1 == (key_t)-1)
      {
            printf("Error al obtener clave para cola mensajes");
            exit(-1);
      }

      /*
      Se crea la cola de mensajes y se obtiene un identificador para
      ella. El IPC_CREAT indica que cree la cola de mensajes si no lo
      está ya.
      El 0600 son permisos de lectura y escritura para el usuario que
      inicie los procesos. Es importante el 0 delante para que se
      interprete en octal.
      */

      Id_Cola_Mensajes = msgget (Clave1, 0600 | IPC_CREAT);
      if (Id_Cola_Mensajes == -1)
      {
            printf("Error al obtener identificador para cola mensajes");
            exit (-1);
      }


      /*
      Se rellenan los campos del mensaje que se quiere enviar.
```

```
El Id_Mensaje es un identificador del tipo de mensaje. Luego se
podrá recoger aquellos mensajes de tipo 1, de tipo 2, etc.
Dato_Numerico es un dato que se quiera pasar al otro proceso. Se
pone, por ejemplo 29. Mensaje es un texto que se quiera pasar al
otro proceso.
*/

Un_Mensaje.Id_Mensaje = 1;
Un_Mensaje.Dato_Numerico = 29;
strcpy (Un_Mensaje.Mensaje, "Hola");

/*
Se envia el mensaje. Los parámetros son:
- Id de la cola de mensajes.
- Dirección al mensaje, convirtiéndola en puntero a (struct
msgbuf*)
- Tamaño total de los campos de datos de nuestro mensaje, es decir
de Dato_Numerico y de Mensaje
- Unos flags. IPC_NOWAIT indica que si el mensaje no se puede
enviar (habitualmente porque la cola de mensajes esta llena), que
no espere y de un error. Si no se pone este flag, el programa queda
bloqueado hasta que se pueda enviar el mensaje.
*/

msgsnd (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,

sizeof(Un_Mensaje.Dato_Numerico)+sizeof(Un_Mensaje.Mensaje),
IPC_NOWAIT);

/*
Se recibe un mensaje del otro proceso. Los parámetros son:
- Id de la cola de mensajes.
- Dirección del sitio en el que queremos recibir el mensaje,
convirtiéndolo en puntero a (struct msgbuf *).
- Tamaño máximo de nuestros campos de datos.
- Identificador del tipo de mensaje que queremos recibir. En este
caso  se quiere un mensaje de tipo 2. Si ponemos tipo 1, se extrae
el mensaje que se acaba de enviar en la llamada anterior a
msgsnd().
- flags. En este caso se quiere que el programa quede bloqueado
hasta que llegue un mensaje de tipo 2. Si se pone IPC_NOWAIT, se
devolvería un error en caso de que no haya mensaje de tipo 2 y el
programa continuaría ejecutándose.
*/

msgrcv (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,
sizeof(Un_Mensaje.Dato_Numerico) + sizeof(Un_Mensaje.Mensaje),
2, 0);

printf("Recibido mensaje tipo 2");
printf("Dato_Numerico = %d", Un_Mensaje.Dato_Numerico);
printf("Mensaje = %s",Un_Mensaje.Mensaje);

/*
Se borra y cierra la cola de mensajes.
```

```
        IPC_RMID indica que se quiere borrar. El puntero del final son
        datos que se quieran pasar para otros comandos. IPC_RMID no
        necesita datos,   así que se pasa un puntero a NULL.
        */

        msgctl (Id_Cola_Mensajes, IPC_RMID, (struct msqid_ds *)NULL);

        return 0 ;
}
```

cola2.c

```
typedef struct Mi_Tipo_Mensaje
{
      long Id_Mensaje;
      int Dato_Numerico;
      char Mensaje[10];
};

int main()
{
      key_t Clave1;
      int Id_Cola_Mensajes;
      Mi_Tipo_Mensaje Un_Mensaje;

      /*
      Igual que en cualquier recurso compartido (memoria compartida,
      semáforos o colas) se obtiene una clave a partir de un archivo
      existente cualquiera y de un entero cualquiera.
      Todos los procesos que quieran compartir este semaforo, deben usar
      el mismo archivo y el mismo entero.
      */

      Clave1 = ftok ("/bin/ls", 33);
      if (Clave1 == (key_t)-1)
      {
            printf("Error al obtener clave para cola mensajes");
            exit(-1);
      }

      /*
      Se crea la cola de mensajes y se obtiene un identificador para
      ella.  El IPC_CREAT indica que cree la cola de mensajes si no lo
      está ya. El 0600 son permisos de lectura y escritura para el
      usuario que inicie los procesos. Es importante el 0 delante para
      que se interprete en  octal.
      */

      Id_Cola_Mensajes = msgget (Clave1, 0600 | IPC_CREAT);
      if (Id_Cola_Mensajes == -1)
      {
            printf("Error al obtener id para cola mensajes");
            exit (-1);
      }

      /*
      Se recibe un mensaje del otro proceso. Los parámetros son:
            - Id de la cola de mensajes.
            - Dirección del sitio en el que queremos recibir el mensaje,
            convirtiéndolo en puntero a (struct msgbuf *).
            - Tamaño máximo de nuestros campos de datos.
            - Identificador del tipo de mensaje que queremos recibir. En
            este caso se quiere un mensaje de tipo 1, que es el que envia
            el proceso cola1.c
            - flags. En este caso se quiere que el programa quede
            bloqueado hasta que llegue un mensaje de tipo 1. Si se pone
```

```
            IPC_NOWAIT, se devolvería un error en caso de que no haya
            mensaje de tipo 1 y el programa continuaría ejecutándose.
      */

      msgrcv (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,
                  sizeof(Un_Mensaje.Dato_Numerico) +
      sizeof(Un_Mensaje.Mensaje), 1, 0);

      printf("Recibido mensaje tipo 1");
      printf("Dato_Numerico = %d");
      printf("Mensaje = %s",Un_Mensaje.Mensaje);

      /*
      Se rellenan los campos del mensaje que se quiere enviar.
      El Id_Mensaje es un identificador del tipo de mensaje. Luego se
      podrá recoger aquellos mensajes de tipo 1, de tipo 2, etc.
      Dato_Numerico es un dato que se quiera pasar al otro proceso. Se
      pone,  por ejemplo 13.
      Mensaje es un texto que se quiera pasar al otro proceso.
      */

      Un_Mensaje.Id_Mensaje = 2;
      Un_Mensaje.Dato_Numerico = 13;
      strcpy (Un_Mensaje.Mensaje, "Adios");

      /*
            Se envia el mensaje. Los parámetros son:
                  - Id de la cola de mensajes.
                  - Dirección al mensaje, convirtiéndola en puntero a
                  (struct msgbuf *)
                  - Tamaño total de los campos de datos de nuestro
                  mensaje, es decir de Dato_Numerico y de Mensaje
                  - Unos flags. IPC_NOWAIT indica que si el mensaje no se
                  puede enviar habitualmente porque la cola de mensajes
                  esta llena), que no espere y de un error. Si no se pone
                  este flag, el programa queda bloqueado hasta que se
                  pueda enviar el mensaje.
      */
      msgsnd (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,

      sizeof(Un_Mensaje.Dato_Numerico)+sizeof(Un_Mensaje.Mensaje),
                  IPC_NOWAIT);

      return 0;
}
```

Memoria compartida (shared memory)

La memoria compartida junto con las colas de mensajes, son los recursos compartidos que disponemos para que dos o más procesos puedan intercambiar información. Esta técnica propone que dos procesos puedan compartir una zona de memoria en común y así intercambiar datos. Al igual que en cola de mensajes, necesitamos conseguir una clave, de tipo key_t, que sea común para todos los programas que quieran compartir la memoria.

Creación de la memoria compartida

Para crear la memoria compartida, haremos uso de la función:

int shmget (key_t, int, int) ;

Con dicha función creamos la memoria y nos devolverá un identificador para dicha zona.

-El primer parámetro es la clave **key_t** obtenida anteriormente y que debería ser la misma para todos los programas.

-El segundo parámetro es el tamaño en bytes que deseamos para la memoria.

-El tercer parámetro son unos flags vinculados a permisos al igual que en el caso de cola de mensajes.

Asociarnos con la memoria compartida creada

El último paso poder usar la memoria consiste en obtener un puntero que apunte la zona de memoria, para poder así escribir o leer sobre la misma. Para ello, declaramos en nuestro código un puntero al tipo que sepamos que va a haber en la zona de memoria (una estructura, un array, tipos simples, etc) y utilizamos la función:

char * shmat (int, char *, int) ;

-El primer parámetro es el identificador de la memoria obtenido en el paso anterior.

-Los otros dos bastará rellenarlos con ceros.

-El puntero devuelto es de tipo char *. Debemos hacerle un "casteo" al tipo que necesitemos o queramos utilizar.

Liberar la memoria compartida

Una vez terminada de usar la memoria, debemos liberarla. Para ello utilizamos las funciones:

```
int shmdt (char *) ;

int shmctl (int, int, struct shmid_ds *) ;
```

La primera función desasocia la memoria compartida de la zona de datos de nuestro programa. Alcanza con pasarle el puntero que tenemos a la zona de memoria compartida y llamarla una vez por proceso.

La segunda función destruye realmente la zona de memoria compartida. Hay que pasarle el identificador de memoria obtenido con shmget(), un flag que indique que queremos destruirla IPC_RMID, y un tercer parámetro al que bastará con pasarle un NULL.

Para poder trabajar con memoria compartida en forma ordenada y sin problemas de acceso, es recomendable utilizar semáforos. Es decir, podemos establecer un semáforo para acceder a la memoria, de forma que cuando un proceso lo está haciendo, el semáforo se pone "rojo" y ningún otro proceso puede acceder a ella. Cuando el proceso termina, el semáforo se pone "verde" y ya podría acceder otro proceso.

A continuación encontrarán dos programas, donde el primero creará la memoria compartida y la escribe y el segundo proceso los va mostrando en pantalla a un intervalo de tiempo.

Programa1.c

```
int main()
{
	key_t Clave;
	int Id_Memoria;
	int *Memoria = NULL;
	int i,j;

	/*
	Conseguimos una clave para la memoria compartida. Todos los
	procesos que quieran compartir la memoria, deben obtener la misma
	clave. Esta se puede conseguir por medio de la función ftok.
	A esta función se le pasa un archivo cualquiera que exista y esté
	accesible (todos los procesos deben pasar el mismo archivo) y un
	entero cualquiera (todos los procesos el mismo entero).
	*/

	Clave = ftok ("/bin/ls", 33);
	if (Clave == -1)
	{
		printf("No consigo clave para memoria compartida");
		exit(0);
	}

	/*
	Creamos la memoria con la clave recién conseguida.
	Para ello llamamos a la función shmget pasándole la clave el tamaño
	de memoria que queremos reservar (100 enteros en nuestro caso) y
	unos flags.
	Los flags son  los permisos de lectura/escritura/ejecucion para
	propietario, grupo y otros (es el 777 en octal) y el flag IPC_CREAT
	para indicar que cree la memoria.
	La función nos devuelve un identificador para la memoria recién
	creada.
	*/

	Id_Memoria = shmget (Clave, sizeof(int)*100, 0777 | IPC_CREAT);
	if (Id_Memoria == -1)
	{
		printf("No consigo Id para memoria compartida");
		exit (0);
	}

	/*
	Una vez creada la memoria, hacemos que uno de nuestros punteros
	apunte a la zona de memoria recién creada. Para ello llamamos a
	shmat, pasándole el identificador obtenido anteriormente y un
	par de parámetros extraños, que con ceros vale.
	*/

	Memoria = (int *)shmat (Id_Memoria, (char *)0, 0);
	if (Memoria == NULL)
```

```
	{
		printf("No consigo memoria compartida");
		exit (0);
	}

	/*
		Ya podemos utilizar la memoria.
		Escribimos cosas en la memoria. Los números de 1 a 10
		esperando un segundo entre ellos. Estos datos serán los que
		lea el otro proceso.
	*/

	for (i=0; i<10; i++)
	{
		for (j=0; j<100; j++)
		{
			Memoria[j] = i;
		}
		printf("Se escribio en la memoria");

		sleep (1);
	}

	/*
		Terminada de usar la memoria compartida, la liberamos.
	*/

	shmdt ((char *)Memoria);
	shmctl (Id_Memoria, IPC_RMID, (struct shmid_ds *)NULL);

	return 0;
}
```

Programa2.c

```
int main()
{
	key_t Clave;
	int Id_Memoria;
	int *Memoria = NULL;
	int i,j;

	/*
		Igual que en el programa anterior, obtenemos una clave para
		la memoria compartida
	*/

	Clave = ftok ("/bin/ls", 33);
	if (Clave == -1)
	{
		printf("No consigo clave para memoria compartida");
		exit(0);
	}

	/*
		Obtenemos el id de la memoria. Al no poner el flag IPC_CREAT,
		estamos suponiendo que dicha memoria ya está creada.
	*/

	Id_Memoria = shmget (Clave, sizeof(int)*100, 0777 );
	if (Id_Memoria == -1)
	{
		printf("No consigo Id para memoria compartida");
		exit (0);
	}

	/*
		Asignamos el puntero a la memoria compartida
	*/

	Memoria = (int *)shmat (Id_Memoria, (char *)0, 0);
	if (Memoria == NULL)
	{
		printf("No consigo memoria compartida");
		exit (0);
	}

	/*
		Leemos el valor de la memoria con demora de un segundo
		y mostramos en pantalla dicho valor. Debería ir cambiando
		según el otro proceso lo va modificando.
	*/

	for (i=0; i<10; i++)
	{
		printf("Leido %d", Memoria[i]);
		sleep (1);
	}
```

```
	/*
		Desvinculamos el puntero de la memoria compartida.
		Suponemos que el proceso que la ha creado, la liberará.
	*/

	if (Id_Memoria != -1)
	{
		shmdt ((char *)Memoria);
	}

	return 0;
}
```

Bibliografía

[1] *"Introducción a las abstracciones del sistema operativo usando Plan 9 from Bell Labs".* Laboratorio de sistemas - Universidad Rey Juan Carlos de Madrid. España. Disponible en: http://lsub.org/plan9introes

[2] Javier Abellan. 2002. Sitio: http://www.chuidiang.org/ (bajo licencia de Creative Commons)

[3] W. R. Stevens y otros. "*Unix Network programming. The sockets networking API"*. Volumen 1. Tercera edición. 2009.

[4] W. R. Stevens. *"Unix Network Programming. Interprocess communications*". Volumen 2. Segunda edición. 2009.

[5] *"La Guía Beej de Comunicación entre procesos Unix"*. Versión 0.9.3. 2004. Disponible en: http://www.ecst.csuchico.edu/~beej/guide/ipc/

[6] Marcelo Doallo. Presentaciones de clase. 2010. Disponible en: http://www2.electron.frba.utn.edu.ar/~mdoallo/