COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS EN LINUX

Los medios de comunicación entre procesos (Inter-Process Communication o IPC) de Linux proporcionan un método para que múltiples procesos se comuniquen unos con otros. Hay varios métodos de IPC disponibles en Linux:

- o Pipes UNIX Half-duplex
- o FIFOs (pipes con nombre)
- o Colas de mensajes estilo SYSTEM V
- o Semáforos estilo SYSTEM V
- o Segmentos de memoria compartida estilo SYSTEM V

Pipes UNIX Semi-duplex

La forma más simple de IPC en Linux son los pipes o tuberías, han estado presentes desde los primeros orígenes del sistema operativo UNIX y proporcionan un método de comunicaciones en un sentido (unidirecional, semi-duplex) entre procesos.

Una tubería (pipe) es simplemente un método de conexión que une la salida estándar de un proceso a la entrada estándar de otro. Para esto se utilizan "descriptores de archivos" reservados, los cuales en forma general son:

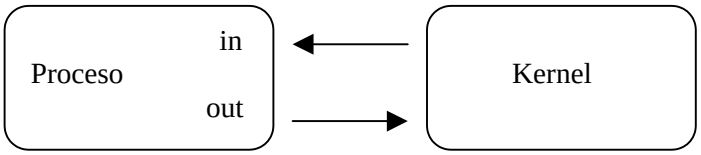
- 0: entrada estándar (stdin).
- 1: salida estándar (stdout).
- 2: salida de error (stderr).

Este mecanismo es ampliamente usado, incluso en la línea de comandos UNIX (en la shell):

```
ls | sort | lp
```

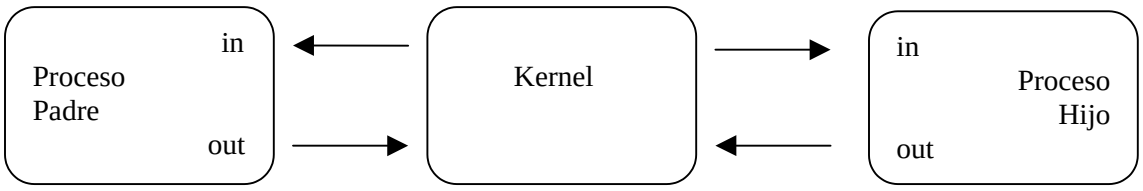
Lo anterior es un ejemplo de "pipeline", donde se toma la salida de un comando `ls` como entrada de un comando `sort`, quien a su vez entrega su salida a la entrada de `lp`. Los datos corren por la tubería semi-duplex, viajando (virtualmente) de izquierda a derecha por la tubería.

Cuando un proceso crea una tubería, el kernel instala dos descriptores de archivos para que los use la tubería. Un descriptor se usa para permitir un camino de entrada a la tubería (write), mientras que la otra se usa para obtener los datos de la tubería (read). A estas alturas, la tubería tiene un pequeño uso practico, ya que la creación del proceso solo usa la tubería para comunicarse consigo mismo. Se podría considerar esta representación de un proceso y del kernel después de que se haya creado una tubería:

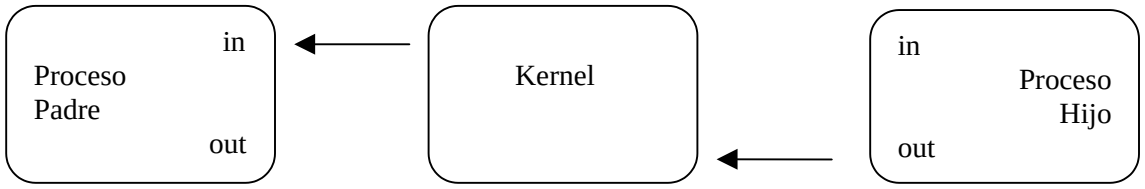


Del diagrama anterior, es fácil ver como se conectan los descriptores. Si el proceso envía datos por la tubería (fd0), tiene la habilidad obtener (leer) esa información de fd1. Sin embargo, hay un objetivo más amplio sobre el esquema anterior. Mientras una tubería conecta inicialmente un proceso a si mismo, los datos que viajan por la tubería se mueven por el kernel.

A estas alturas, la tubería es bastante inútil. ¿Para que crear una tubería si solo estamos hablando con nosotros mismos? Ahora, el proceso de creación bifurca un proceso hijo. Como un proceso hijo hereda cualquier descriptor de archivo abierto del padre, ahora tenemos la base para la comunicación multiprocesos (entre padre e hijo). La versión actualizada del esquema simple quedaría como:



Arriba, se ve que ambos procesos ahora tienen acceso al descriptor del archivo que constituye la tubería. En ésta fase se debe tomar una decisión critica. ¿En que dirección se quiere que viajen los datos? ¿El proceso hijo envía información al padre, o viceversa? Se debe proceder a "cerrar" el extremo de la tubería que no interesa. Suponiendo que el hijo ejecuta su código, y devuelve información por la tubería al padre. El esquema ya revisado aparecería como:



Ahora la construcción de la tubería esta completa. Lo único que queda por hacer es usar la tubería. Para acceder a una tubería directamente, se puede usar la misma llamada al sistema que se usa para un archivo E/S de bajo nivel.

Para enviarle datos a la tubería, se usa la llamada al sistema write(), y para recuperar datos de la tubería, se usa la llamada al sistema read(). Se debe tener presente que ciertas llamadas al sistema, como por ejemplo lseek(), no trabaja con descriptores a tuberías.

Creación de tuberías en C

Para crear una tubería simple con C, se usa la llamada al sistema *pipe()*. Toma un argumento solo, que es un arreglo de dos enteros, y si tiene éxito, la tabla contendrá dos nuevos descriptores de archivos para ser usados por la tubería.

Este primer ejemplo (bastante inútil), crea, escribe y lee desde un pipe:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main()
{
```

```
    int     fd[2];
    char    buffer[30];

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

Este ejemplo no tiene mucho sentido, ya que el proceso se está comunicando consigo mismo. Veamos que pasa si se llama a *fork()* y se hace que el proceso hijo mande la palabra "hola" al padre.

Primero, el padre debe llamar a *pipe()*. Segundo, se llama a *fork()*. El hijo recibe una copia de todos los descriptores de archivos del padre, incluyendo una copia de los descriptores de archivos del pipe. Esto permite que el hijo mande datos al extremo de escritura del pipe, y el padre los reciba del extremo de lectura.:

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

int main()
{
    int     fd[2];
    char    buffer[30];

    pipe(fd);

    if (!fork()) {
        printf(" HIJO: Escribiendo en el pipe\n");
        write(fd[1], "hola", 5);
        printf(" HIJO: Chao...\n");
        exit(0);
    } else {
        printf("PADRE: Leyendo desde el pipe\n");
        read(fd[0], buffer, 5);
        printf("PADRE: He leído \"%s\"\n", buffer);
        wait(NULL);
    }
}
```

La salida en pantalla debería ser la siguiente:

```
PADRE: Leyendo desde el pipe
 HIJO: Escribiendo en el pipe
 HIJO: Chao...
PADRE: He leído "hola"
```

En este caso, el padre intentó leer desde el pipe antes que el hijo escribiera. Cuando esto ocurre, se dice que el padre se *bloquea*, o duerme, hasta que llegan datos que leer. Al parecer el padre intentó leer, se durmió, el hijo escribió y terminó, y el padre despertó y leyó el dato.

Veamos ahora un ejemplo más completo y con más verificación de errores.

El primer elemento del arreglo fd (elemento 0) esta fijado y abierto para lectura, mientras el segundo entero (elemento 1) esta fijado y abierto para escritura. Visualmente hablando, la salida de fd1 se vuelve la entrada para fd0.

Todo los datos que se mueven por la tubería lo hacen por el kernel. Se debe recordar que un nombre de arreglo en C es un puntero a su primer miembro. Es decir, fd es equivalente a &fd0. Una vez establecida la tubería, se puede crear (mediante fork) un nuevo proceso hijo.

Si el padre quiere recibir datos del hijo, debe cerrar fd1, y el hijo debe cerrar fd0. Si el padre quiere enviarle datos al hijo, debe cerrar fd0, y el hijo debe cerrar fd1. Como los descriptores se comparten entre el padre y hijo, siempre se debe cerrar el extremo de la tubería que no interesa, nunca se devolverá EOF si los extremos innecesarios de la tubería no son explícitamente cerrados. Como se menciono previamente, una vez se ha establecido la tubería, los descriptores de archivo se tratan como descriptores a archivos normales.

```
/*************************************************************************
  Parte de la "Guía Linux de Programación - Capitulo 6"
  (C)opyright 1994-1995, Scott Burkett
  *************************************************************************
  MODULO: pipe.c
*************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hola a todos!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Cierre del descriptor de entrada en el hijo */
                close(fd[0]);

                /* Enviar el saludo vía descriptor de salida */
                write(fd[1], string, strlen(string));
                exit(0);
        }
```

```
        else
        {
                /*  Cierre  del  descriptor  de  salida  en  el  padre   */
                close(fd[1]);

                /*  Leer  algo  de  la  tubería...  el  saludo!  */
                nbytes  =  read(fd[0],  readbuffer,  sizeof(readbuffer));
                printf("He recibido el string:  %s",  readbuffer);
        }

        return(0);
}
```

Otro uso de los pipes, es la implementación de por ejemplo los comandos “ls | wc -l" en C. Esto requiere del uso de dos funciones: *exec()* y *dup()*. Las funciones *exec()* reemplazan el proceso actualmente en ejecución con cualquier proceso que se le pasa a *exec()*. Esta función se utilizará para ejecutar ls y wc -l. La función *dup()* recibe un descriptor de archivos abierto y lo duplica. Así es como se puede conectar la salida estándar de ls a la entrada estándar de wc.

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

int main()
{
    int fd[2];

    pipe(fd);

    if (!fork()) {
        close(1);          /* cerrar el stdout normal */
        dup(fd[1]);        /* hacer que stdout sea igual a fd[1] */
        close(fd[0]);      /* fd[0] no se necesita */
        execlp("ls", "ls", NULL);
    } else {
        close(0);          /* cerrar el stdin normal */
        dup(fd[0]);        /* hacer que stdin sea igual a fd[0] */
        close(fd[1]);      /* fd[1] no se necesita */
        execlp("wc", "wc", "-l", NULL);
    }
}
```

close(1) libera el descriptor de archivo 1 (salida estándar). dup(fd[1]) crea una copia del extremo de escritura del pipe en el primer descriptor de archivo que esté disponible, que es "1", ya que recién fue cerrado. De esta manera, cualquier cosa que ls escribe a la salida estándar (descriptor de archivo 1) irá a fd[1] (el extremo de escritura del pipe). La sección de código de wc funciona de la misma manera, excepto al revés.

Hay otra llamada al sistema, dup2 (), que se puede usar también. Esta llamada particular tiene su origen con la Versión 7 de UNIX, se realizo por una versión de BSD y ahora es requerida por el estándar POSIX.

Con esta particular llamada, se tiene la operación de cerrado, y la duplicación del descriptor actual, relacionado con una llamada al sistema. Además, se garantiza el ser atómica, que esencialmente significa que nunca se interrumpirá por la llegada de una señal (interrupción). Toda la operación transcurrirá antes de devolverle el control al kernel para despachar la señal.

Con la llamada al sistema dup() original, se tenia que ejecutar un close() antes de llamarla. Esto resultaba en dos llamadas del sistema, con un pequeño grado de vulnerabilidad en el breve tiempo que transcurre entre ellas. Si llega una señal durante ese tiempo, la duplicación del descriptor fallaría. dup2 () resuelve este problema.

```
.
.

childpid  =  fork();

if(childpid  ==  0)
{
          /*  Cerrar  entrada  estándar,  duplicando  a  esta  la
             salida  de  datos  de  la  tubería  */
          dup2(0, fd[0]);
          execlp("sort",  "sort",  NULL);
          .
          .
}
```

Operaciones atómicas con tuberías
Para que una operación se considere "atómica", no se debe interrumpir de ninguna manera. Todo su funcionamiento ocurre de una vez. La norma POSIX indica en /usr/include/posix1_lim.h que el tamaño máximo del buffer para una operación atómica en una tubería es:

```
#define  _POSIX_PIPE_BUF            512
```

Hasta 512 bytes se pueden escribir o recuperar de una tubería atómicamente. Cualquier cosa que sobrepase este limite se partirá. Bajo Linux sin embargo, se define el limite atómico operacional en "linux/limits.h" como:

```
 #define  PIPE_BUF          4096
```

Como se puede ver, Linux adapta el numero mínimo de bytes requerido por POSIX, y se le pueden agregar bastantes. La atomicidad del funcionamiento de la tubería se vuelve importante cuando implica mas de un proceso (FIFOS). Por ejemplo, si el numero de bytes escritos en una tubería excede el limite atómico para una simple operación, y procesos múltiples están escribiendo en la tubería, los datos serán "intercalados" o "chunked". En otras palabras, un proceso insertaría datos en la tubería entre la escritura de otro.

Notas acerca de las tuberías semi-duplex:

- Se pueden crear tuberías de dos direcciones abriendo dos tuberías, y reasignando los descriptores de archivo al proceso hijo.

- La llamada a pipe() debe hacerse ANTES de la llamada a fork(), o los hijos no heredaran los descriptores.

- Con tuberías semi-duplex, cualquier proceso conectado debe compartir el ancestro indicado. Como la tubería reside en el kernel, cualquier proceso que no sea ancestro del creador de la tubería no tiene forma de direccionarlo. Este no es el caso de las tuberías con nombre (FIFOS).

Tuberías con Nombre (FIFO - First In First Out)

Un FIFO también se conoce como una tubería con nombre. El nombre es de un archivo que múltiples procesos pueden abrir, leer y escribir. Una tubería con nombre funciona como una tubería normal, pero tiene algunas diferencias notables:

- Las tuberías con nombre existen en el sistema de archivos como un archivo de dispositivo especial.
- Los procesos de diferentes padres pueden compartir datos mediante una tubería con nombre.
- Una vez finalizadas todas las operaciones de E/S, la tubería con nombre permanece en el sistema de archivos para un uso posterior.

Creación de una FIFO
Hay varias formas de crear una tubería con nombre. Las dos primeras se pueden hacer directamente desde el shell:

```
mknod  MIFIFO  p
mkfifo  a=rw  MIFIFO
```

Los dos comandos anteriores realizan operaciones idénticas, con una excepción. El comando mkfifo proporciona una posibilidad de alterar los permisos del archivo FIFO directamente tras la creación. Con mknod será necesaria una llamada al comando chmod.

Los archivos FIFO se pueden identificar rápidamente en un archivo físico por el indicador "p" que aparece en la lista del directorio.

```
$  ls  -l  MIFIFO
prw-r--r--    1  root       root              0  Dec  14  22:15  MIFIFO|
```

También hay que observar que la barra vertical ("símbolo pipe") esta situada inmediatamente detraes del nombre de archivo.

Para crear un FIFO en C, se puede hacer uso de la llamada del sistema mknod() con sus respectivos argumentos:

```
mknod("/tmp/MIFIFO", S_IFIFO|0644 , 0);
```

En este caso el archivo "/tmp/MIFIFO" se crea como archivo FIFO. El segundo argumento es el modo de creación, que sirve para indicarle a `mknod()` que crea un FIFO (con `S_IFIFO`) y pone los permisos de acceso a ese archivo (644 octal, rw-r--r--) al igual como se puede hacer con el comando `chmod`. Finalmente, el tercer argumento de mknod() se ignora (al crear un FIFO) salvo que se este creando un archivo de dispositivo. En ese caso, se debería especificar los números mayor y menor del archivo de dispositivo.

Los permisos se ven afectados por la configuración de umask de la siguiente forma:

```
umask_definitiva  =  permisos_solicitados  &  ~umask_inicial
```

Un truco común es usar la llamada del sisterma umask() para borrar temporalmente el valor de umask:

```
umask(0);
mknod("/tmp/MIFIFO",  S_IFIFO|0666,  0);
```

Operaciones con FIFOs

Las operaciones E/S sobre un FIFO son esencialmente las mismas que para las tuberías normales, con una gran excepción. Se debe usar una llamada del sistema `open()` o una función de librería para abrir físicamente un canal para la tubería. Con las tuberías semi-duplex, esto es innecesario, ya que la tubería reside en el kernel y no en un sistema de archivos físico. En nuestro ejemplo trataremos la tubería como un stream, abiendolo con fopen(), y cerrándolo con fclose().

Consideramos un proceso servidor simple:

```
/*************************************************************************
  Parte de la "Guía Linux de Programación - Capitulo 6"
  (C)opyright 1994-1995, Scott Burkett
  *************************************************************************
  MODULO: fifoserver.c
  *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE        "MIFIFO"

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);
}
```

Como un FIFO bloquea por defecto, se debe ejecutar el servidor en segundo plano tras compilarlo:

```
$ fifoserver &
```

Se discutirá la acción de bloqueo de un FIFO más adelante. Primero consideremos el siguiente cliente:

```
/*************************************************************************
 Parte de la "Guía Linux de Programación - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
```

```
 ***************************************************************************
 MODULO:  fifoclient.c
 ***************************************************************************/

#include  <stdio.h>
#include  <stdlib.h>

#define  FIFO_FILE         "MIFIFO"

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);
}
```

Acciones Bloqueantes en una FIFO
Normalmente, el bloqueo ocurre en un FIFO. En otras palabras, si se abre el FIFO para lectura, el proceso estará "bloqueado" hasta que cualquier otro proceso lo abra para escritura. Esta acción funciona al revés también. Si este comportamiento no nos interesa, se puede usar la bandera O_NONBLOCK en la llamada a open() para desactivar la acción de bloqueo por defecto.

En el caso de nuestro servidor simple, lo hemos puesto en segundo plano, y permitido hacer su bloqueo allí.

Al tener el nombre del pipe que se encuentra en el disco hace que todo sea mucho más fácil. Procesos no relacionados entre ellos pueden ahora comunicarse mediante los pipes.

Colas de Mensajes

Una cola de mensajes funciona como una FIFO, pero con algunas diferencias. Generalmente, los mensajes son sacados de la cola en el orden en que se pusieron. Sin embargo, hay maneras de sacar cierto mensaje de la cola antes de que alcance llegar al inicio de la cola.

Un proceso puede crear una nueva cola de mensajes, o se puede conectar a una ya existente. De esta forma, dos procesos pueden compartir información mediante la misma cola de mensajes.

Una vez que se crea una cola de mensajes, esta no desaparece hasta que se destruya. Todos los procesos que alguna vez la usaron pueden finalizar, pero la cola todavía existirá. Una buena costumbre sería usar el comando `ipcs` para verificar si existe alguna cola de mensajes que ya no esté en uso y destruirla con el comando `ipcrm`.

Primero que nada queremos conectarnos a una cola, o crearla si no existe. Para hacer esto se utiliza la llamada al sistema `msgget()`:

```
int msgget(key_t key, int msgflg);
```

`msgget()` devuelve el ID de la cola de mensajes en caso de éxito, o `-1` en caso de error.

El primer argumento, *`key`*, es un identificador único que describe la cola a la cual uno se quiere conectar (o crear). Cualquier otro proceso que quiera conectarse a esta cola deberá usar el mismo *`key`*.

El segundo argumento, *`msgflg`*, le dice a `msgget()` qué hacer con la cola. Para crear una cola, este campo debe ser igual a `IPC_CREAT` junto a los permisos para la cola. (Los permisos de la cola son los mismos que los permisos estándar de archivos – las colas reciben los user-id y group-id del programa que las creó.)

Cómo crear el identificador *`key`*? Ya que el tipo `key_t` es un `long`, se puede usar cualquier número que se quiera. Pero que pasaría si otro programa utiliza el mismo número pero quiere usar otra cola? La solución es usar la función `ftok()` la cual genera un identificador a partir de dos argumentos:

```
key_t ftok(const char *path, int id);
```

Básicamente, *`path`* debe ser un archivo que el proceso pueda leer. El otro argumento, *`id`* es normalmente un char escogido arbitrariamente, como por ejemplo 'A'. La función `ftok()` usa información sobre el archivo y el *`id`* para generar un identificador, probablemente único, para ser usado en `msgget()`. Los programas que quieran usar la misma cola deben generar el mismo identificador *`key`*, así que deben pasarle algunos parámetros a `ftok()`.

```
#include <sys/msg.h>

key = ftok("/home/rmunoz/cualquier_archivo", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

En este ejemplo, se le ponen los permisos `666` (o `rw-rw-rw-`) a la cola. Ahora tenemos `msqid` que se puede utilizar para mandar y recibir mensajes desde la cola.

Enviar a una cola

Una vez que uno se conecta a la cola de mensajes usando `msgget()`, se puede mandar y recibir mensajes.

Para mandar mensajes:

Cada mensaje consiste de dos partes, las cuales están definidas en la estructura `struct msgbuf`, tal como aparece en `sys/msg.h`:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

El campo `mtype` es usado para recibir mensajes, y puede ser cualquier número positivo. `mtext` es el dato que se añadirá a la cola.

Se puede usar cualquier estructura para mandar mensajes a la cola, siempre y cuando el primer elemento es un long. Por ejemplo, se podría usar la siguiente estructura para almacenar todo tipo de información:

```
struct pirata_msgbuf {
   long mtype;          /* debe ser positivo */
   char nombre[30];
   char tipo_de_barco;
   int crueldad;
};
```

Una vez definida la estructura, solo resta enviar la información a la cola usando `msgsnd()`:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

`msqid` es el identificador de cola de mensajes devuelta por `msgget()`. El puntero *`msgp`* es un puntero a los datos que se quiere poner en la cola.. *`msgsz`* es el tamaño de los datos que se quieren enviar. Finalmente, *`msgflg`* permite poner parámetros adicionales, que por el momento se ignorarán poniéndolos en 0.

Este es el código que muestra como nuestros datos son añadidos a la cola de mensajes:

```
#include

key_t key;
int msqid;
struct pirata_msgbuf pmb = {2, "LeChuck", 'S', 80};

key = ftok("/home/rmunoz/cualquier_archivo", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

msgsnd(msqid, &pmb, sizeof(pmb), 0);  /* añadirlo a la cola */
```

Sólo faltaría verificar los errores. El valor de `mtype` se dejó arbitrariamente en `2`. Esto será importante en la próxima sección.

Recibir desde la cola

La contraparte de `msgsnd()` es `msgrcv()`. Esta sería la forma de recibir usando la llamada `msgrcv()`:

```
#include
```

```
key_t key;
int msqid;
struct pirata_msgbuf pmb; /* dónde se guardará a LeChuck */

key = ftok("/home/rmunoz/cualquier_archivo", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

msgrcv(msqid, &pmb, sizeof(pmb), 2, 0);  /* sacarlo de la cola! */
```

Hay algo nuevo que notar en la llamada `msgrcv(): el 2`! Qué significa esto? Aquí esta la sintaxis de la llamada:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

El `2` que especificamos en la llamada, es el *`msgtyp`* (tipo de mensaje) requerido. Recuerden que dejamos arbitrariamente `mtype` en 2, en la sección de `msgsnd()`, así que este será el tipo de mensaje que se recibirá.

De hecho, el comportamiento de `msgrcv()` puede ser modificado drásticamente seleccionando un *`msgtyp`* que sea positivo, negativo, o cero:

msgtyp	**Efecto en `msgrcv()`**
Cero	Recibir el próximo mensaje de la cola, independientemente de su `mtype`.
Positivo	Obtener el próximo mensaje con un `mtype` **igual** al especificado.
Negativo	Recibir el primer mensaje de la cola cuyo `mtype` es menor o igual al valor absoluto del argumento *`msgtyp`*.

Muchas veces se quiere simplemente recibir el próximo mensaje de la cola, sin importar su `mtype`. En ese caso, se dejaría el parámetro *`msgtyp`* en `0`.

Eliminar una cola de mensajes

Llega un momento cuando se quiere eliminar una cola de mensajes. Como ya se mencionó, las colas de mensajes quedan a menos que se eliminen explícitamente; es importante hacer esto para no gastar recursos del sistema. Existen dos formas de hacerlo:

1. Usar el comando de UNIX `ipcs` para obtener una lista de colas de mensajes definidas, luego usar el comando `ipcrm` para eliminar la cola.
2. Escribir el código necesario.

Generalmente, es más recomendada la segunda opción, ya que se puede desear que el programa limpie la cola en cierto momento. Para hacer esto se utiliza otra función: `msgctl()`.
La sintaxis de `msgctl()` es:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

msqid es por supuesto el identificador de la cola obtenido de `msgget()`. El argumento importante es **`cmd`** que le indica a `msgctl()` qué hacer. Pueden ser varias cosas, pero sólo veremos `IPC_RMID`, que se usa para eliminar colas de mensajes. El argumento *buf* se puede dejar en `NULL` ya que estamos utilizando `IPC_RMID`.

Para destruir la cola que creamos anteriormente:

```
#include
.
.
msgctl(msqid, IPC_RMID, NULL);
```

Y la cola desaparecerá.

Algunos ejemplos

Estos dos programas se comunican entre ellos usando colas de mensajes. El primero, `kirk.c` añade mensajes en la cola, y `spock.c` los saca.

Este es el código de kirk.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1); }
    printf("Ingrese líneas de texto, ^D para salir:\n");

    buf.mtype = 1;       /* no importa en este caso */
    while(gets(buf.mtext), !feof(stdin)) {
```

```
            if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == -1)
                perror("msgsnd");
        }

        if (msgctl(msqid, IPC_RMID, NULL) == -1) {
            perror("msgctl");
            exit(1);
        }

        return 0;
    }
```

El programa `kirk` permite que se ingresen líneas de texto. Cada líneas es insertada en un mensaje y añadida a la cola de mensajes. Luego la cola de mensajes es leída por `spock`.

Este es el código de spock.c:

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    struct my_msgbuf {
        long mtype;
        char mtext[200];
    };

    int main(void)
    {
        struct my_msgbuf buf;
        int msqid;
        key_t key;

        if ((key = ftok("kirk.c", 'B')) == -1) {  /*mismo key que kirk.c*/
            perror("ftok");
            exit(1);
        }

        if ((msqid = msgget(key, 0644)) == -1) { /* conectarse a la cola */
            perror("msgget");
            exit(1);
        }

printf("spock: listo para recibir mensajes, capitan.\n");

        for(;;) { /* Spock nunca termina */
           if (msgrcv(msqid,(struct msgbuf *)&buf,sizeof(buf),0,0) == -1) {
                perror("msgrcv");
                exit(1);
```

```
            }
            printf("spock: \"%s\"\n", buf.mtext);
        }

        return 0;
    }
```

Notar que `spock`, en la llamada a `msgget()`, no incluye la opción `IPC_CREAT`. Dejamos que `kirk` creara la cola de mensajes, y `spock` devolverá un error si esto no ha ocurrido.

Notar también que pasa si se ejecutan los dos y se termina el uno o el otro. También prueben ejecutar dos copias de `kirk` o dos copias de `spock` para ver que pasa al tener dos lectores o dos escritores. Otra demostración interesante es ejecutar a `kirk`, ingresar un montón de mensajes, luego ejecutar `spock` y ver como los recupera todos de un golpe.

SEMAFOROS

Los semáforos se pueden utilizar para controlar el acceso a los archivos, memoria compartida, y en general cualquier otro recurso compartido.

Obtener semáforos

Con IPC de System V, se obtienen *conjuntos* de semáforos. Obviamente, se puede obtener un conjunto de semáforos que sólo contiene un semáforo, pero la idea es que se puede tener una gran cantidad de semáforos al crear un solo conjunto de semáforos.

Cómo se hace para crear un conjunto de semáforos? Se hace con la llamada `semget()`, que devuelve el id (identificador) del semáforo (de aquí en adelante será referido como *`semid`*):

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

int semget(key_t key, int nsems, int semflg);
```

Que lo que es el *`key`*? Es un identificador único que es usado por procesos diferentes para identificar este conjunto de semáforos. (El *`key`* será generado usando `ftok()`, descrito en los apuntes sobre las colas de mensajes).

El siguiente argumento, *`nsems`*, es la cantidad de semáforos en este conjunto de semáforos.

Finalmente, el argumento *`semflg`* le indica a `semget()` entre otras cosas; qué permisos tendrá el nuevo conjunto de semáforos, si se está creando un nuevo conjunto o si sólo se quiere conectar a un conjunto ya existente. Para crear un nuevo conjunto, se puede combinar los permisos con el argumento `IPC_CREAT`.

Aquí hay un ejemplo donde se crea el *`key`* con `ftok()` y luego se crea un conjunto de 10 semáforos, con los permisos 666 (`rw-rw-rw-`):

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

key_t key;
int semid;

key = ftok(".", 'E');
semid = semget(key, 10, 0666 | IPC_CREAT);
```

Una vez ejecutado este ejemplo, se puede verificar la existencia del conjunto de semáforos con el comando `ipcs`. (No se olviden eliminar el semáforo con el comando `ipcrm`!)

`semop()`: poder atómico!

Todas las operaciones sobre semáforos utilizan la llamada al sistema `semop()`. Esta llamada al sistema es de propósito general, y su funcionalidad es dictada por una estructura que se le pasa, `struct sembuf`:

```
struct sembuf {
    ushort sem_num;
    short sem_op;
    short sem_flg;
};
```

`sem_num` es el número del semáforo en el conjunto que se quiere manipular. `sem_op` es lo que se quiere hacer con ese semáforo. Esto tiene varios significados, dependiendo si `sem_op` es positivo, negativo, o cero, tal como se muestra en la siguiente tabla:

`sem_op`	**Lo que pasa**
Positivo	El valor de `sem_op` es sumado al valor del semáforo. Así es como un programa usa un semáforo para marcar un recurso como ocupado.
Negativo	Si el valor absoluto de `sem_op` es mayor que el valor del semáforo, el proceso que lo llama se bloqueará hasta que el valor del semáforo alcanza el valor absoluto de `sem_op`. Finalmente, el valor absoluto de `sem_op` será restado del valor del semáforo. Así es como un proceso entrega un recursos vigilado por el semáforo.
Cero	Este proceso esperará hasta que el semáforo en cuestión alcance el 0.

Así que, básicamente, lo que se hace es cargar un `struct sembuf` con los valores que se quieran, y luego llamar a `semop()`, de esta manera:

```
int semop(int semid ,struct sembuf *sops, unsigned int nsops);
```

El argumento *`semid`* es el número obtenido de la llamada a `semget()`. *`sops`* es un puntero a una estructura `struct sembuf` que ya se haya rellenado con los comandos de los semáforos. Incluso se puede crear un arreglo de `struct sembuf`s, para realizar muchas operaciones al mismo tiempo.

El argumento `sem_flg` permite al programa especificar modificadores que modifican más aún los efectos de una llamada a `semop()`.

Uno de estos modificadores es `IPC_NOWAIT`, que causa que la llamada `semop()` devuelva el error `EAGAIN` si se encuentra con una situación donde normalmente se bloqueará. Esto sirve para cuando se quiere “sondear” con el fin de verificar si se puede ocupar algún.

Otro modificador muy útil es `SEM_UNDO`. Este causa que `semop()` grabe, de cierta forma, el cambio realizado en el semáforo. Cuando el programa termina su ejecución, el kernel automáticamente deshace todos los cambios que fueron marcados con el modificador `SEM_UNDO`.

Destruir un semáforo

Hay dos maneras de deshacerse de un conjunto semáforos: una es usando el comando `ipcrm` de UNIX. La otra manera es a través de la llamada `semctl()` con los argumentos adecuados.

Esta es la unión `union semun`, junto con la llamada a `semctl()` para destruir el semáforo:

```
union semun {
    int val;              /* usado sólo para SETVAL */
    struct semid_ds *buf; /* para IPC_STAT y IPC_SET */
    ushort *array;        /* usado para GETALL y SETALL */
};

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

Básicamente, se pone en *`semid`* el ID del semáforo que se quiere destruir. El *`cmd`* debe estár en `IPC_RMID`, lo que le indica a `semctl()` que debe sacar este conjunto de semáforos. Los parámetros *`semnum`* y *`arg`* no tienen ningún significado en el contexto de `IPC_RMID` y se puede dejar en cualquier cosa.

Esto es un ejemplo de cómo eliminar un semáforo:

```
union semun dummy;
int semid;
.
.
semid = semget(...);
.
.
semctl(semid, 0, IPC_RMID, dummy);
```

Cuando recién se crearon los semáforos, estos quedan inicializados en cero. Esto es grave, ya que significa que todos están marcados como ocupados; y se necesita otra llamada (ya sea a `semop()` o a `semctl()` para marcarlos como libres). Qué significa todo esto? Esto significa que la creación de semáforos no es atómica. Si dos procesos están tratando de crear, inicializar y usan un semáforo al mismo tiempo, se puede producir una condición de carrera.

Este problema se puede resolver teniendo un solo proceso que crea e inicializa el semáforo. El proceso principal sólo lo accesa, pero nunca lo crea o lo destruye.

Ejemplos

Hay tres ejemplos. El primero, `seminit.c`, crea e inicializa el semáforo. El segundo, `semdemo.c`, controla el acceso a un supuesto archivo. Finalmente, `semrm.c` es usado para destruir el semáforo (esto podría realizarse con el comando `ipcrm`)

La idea es ejecutar `seminit.c` para crear el semáforo. Se puede usar el comando `ipcs` desde la línea de comandos para verificar que el semáforo existe. Luego ejecutar `semdemo.c` en un par de ventanas y ver como ellos interactúan. Finalmente, se debe usar `semrm.c` para eliminar el semáforo.

Este es el código de `seminit.c` (debe ejecutarse de los primeros!):

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

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

    if ((key = ftok(".", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* crear un conjunto de semáforos con 1 semáforo: */
    if ((semid = semget(key, 1, 0666 | IPC_CREAT)) == -1) {
        perror("semget");
        exit(1);
    }

    /* inicializar el semáforo #0 en 1: */
    arg.val = 1;
    if (semctl(semid, 0, SETVAL, arg) == -1) {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

Este es el código de `semdemo.c`:

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

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb = {0, -1, 0};  /* para reservar recursos */

    if ((key = ftok(".", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }
```

```
    /* conectarse al conjunto de semáforos creado por seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    printf("Presione ENTER para bloquear: ");
    getchar();
    printf("Tratándo de bloquear...\n");

    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Bloqueado.\n");
    printf("Presione ENTER para desbloquear: ");
    getchar();

    sb.sem_op = 1; /* liberar recurso */
    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Desbloqueado\n");

    return 0;
}
```

Este es el código de `semrm.c`:

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

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

    if ((key = ftok(".", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* conectarse al semáforo creado por seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }
```

```
        /* eliminarlo */
        if (semctl(semid, 0, IPC_RMID, arg) == -1) {
            perror("semctl");
            exit(1);
        }

        return 0;
    }
```

Los semáforos son muy útiles en una situación de concurrencia. Sirven para controlar el acceso a un archivo, pero también a otro recurso como por ejemplo la memoria compartida.

Siempre cuando se tienen múltiples procesos ejecutándose dentro de una sección crítica de código, se necesitan los semáforos.