La Guía Beej de Comunicación entre procesos Unix

Versión 0.9.3 (16 de noviembre de 2004)
[http://www.ecst.csuchico.edu/~beej/guide/ipc /]

Intro

¿Sabe que es fácil? Usar la función fork () (fork significa bifurcación). Usted puede bifurcar un proceso todo el tiempo y tratar un problema en forma paralela y fraccionada en pedazos cortos.
Obviamente esto es más fácil si los procesos no se comunican entre sí mientras están corriendo y pueden quedarse haciendo lo propio si preocuparse por los demás procesos.
Sin embargo, cuando empieza a bifurcar procesos comienza a pensar en las aplicaciones multi-usuario que podría hacer si los procesos pudieran dialogar con los demás procesos en forma sencilla. También puede intentar hacer un array global y luego bifurcarse para ver si este es compartido. (es decir, ver si tanto el proceso hijo como el proceso padre usan el mismo array.)
Encontrará que el proceso hijo tiene su propia copia del array y el padre es olvidado para que el hijo pueda hacer cualquier cambio. Cada vez que un proceso crea a otro, le transfiere sus recursos, pero no se duplican. El kernel detecta el acceso para modificar un recurso por parte del hijo y en ese momento crea una copia propia.
¿Cómo puede conseguir que estos tipos hablen con otros, compartan estructuras de datos y sean generalmente amigables ? Este documento discute diversos métodos de comunicación entre procesos (“IPC´s) que pueden lograr este cometido alguno de los cuales son mejores y más satisfactorios para ciertas tareas que para otras.

Público

Si usted conoce C o C++ y es bastante bueno usando el entorno Unix (u otros entornos POSIX que soporten estas system calls), este documento es para usted. Si no está familiarizado, bueno, ¡ Está en el horno!. Este documento supone, sin embargo que usted cuenta con una basta experiencia en programación en C.
Este documento significa un trampolín para el usuario en el reino de las IPC´s entregando una apreciación global y concisa de varias técnicas en tal sentido. Éste no es el juego definitivo de documentos que cubren este asunto, sino que solo es un pie para introducirse en el mundo de las IPC´s.

Plataforma y compilador

Se compilaron los ejemplos en este documento bajo Linux que usa gcc. Ellos también se han podido compilar bajo HPUX 10.10 usando cc -Ae.

Documentos específicos:

Un fork () adecuado

La función fork () puede pensarse como un medio poderoso. Poder que a veces puede pensarse como un boleto a la destrucción. Por consiguiente, debe tener cuidado cuando utiliza el fork en su sistema. No es que nunca deba jugar con fork, solo tiene que ser cauto.
Un fork () es la forma cómo Unix empieza nuevos procesos. Básicamente, sucede así: el proceso padre (el que ya existe) se bifurca a un proceso hijo (el nuevo). El proceso hijo obtiene una copia de los datos del padre. ¡Voalá! ¡ Tiene dos procesos donde había uno solo !
Por supuesto, si llena la máquina de procesos el sistema se tornará tan lento que forzará a reiniciar las máquinas.
En primer lugar, debe conocer algo acerca de la conducta del proceso bajo Unix. Cuando un proceso se muere, realmente no se marcha completamente. Está muerto, porque ya no está corriendo, pero un remanente pequeño está esperando alrededor para ser recogido por el proceso padre. Este remanente contiene el valor del retorno del proceso del hijo y algo más.
Así después de una bifurcación del proceso padre a un proceso hijo, debe esperar (waitpid ( )) al proceso hijo para terminar. Es este acto de espera el que permite que todos los remanentes del hijo puedan desaparecer.
Naturalmente, hay una excepción a la regla anterior: el padre puede ignorar la señal SIGCLD y entonces no tendrá que esperar . Esto puede hacerse (en sistemas que lo soportan) del siguiente modo:

```
main()
{
    signal(SIGCLD, SIG_IGN);  /* ahora no tengo que esperar! */
    .
    .
    fork();fork();fork();  /* se reproduce como conejos! */
```

Ahora, cuando un proceso hijo se muere y no ha sido esperado , este se mostrará normalmente en una listado ps como < defunct> (difunto). Este seguirá siendo difunto mientras el padre lo espere o se reparte como se menciona debajo.
Hay otra regla que debe aprender ahora: cuando el padre se muere antes de que culmine la espera del hijo (asumiendo que no está ignorando SIGCLD), el hijo es reenparentado al proceso inicial init (con PID 1). Esto no es un problema si el niño todavía está viviendo bien y bajo control. Sin embargo, si el hijo ya está difunto, estaremos por un momento en un lazo, Vea que el padre original ya no puede esperar, porque está muerto. ¿Entonces cómo saben los init que deben esperar a estos procesos zombis ?
La respuesta: ¡es mágico! Bien, en algunos sistemas, el init destruye todos los procesos difuntos que posee periódicamente. En otros sistemas, él muy descarado se niega a volverse el padre de cualquier proceso difunto y en cambio los destruye inmediatamente . Si está usando uno de los sistemas anteriores, podría escribir una lazo que llene a la tabla de procesos con procesos difuntos poseídos por INIT.
¿No habría hecho feliz a su administrador de sistema?
Su misión: asegúrese que su proceso padre ignore SIGCLD, o espere a todos los procesos hijos bifurcados.
En definitiva: los hijos se vuelven difuntos a la espera del padre, a menos que el padre esté ignorando SIGCLD. Además, los hijos (vivos o difuntos) cuyos padres se mueren sin esperarlos a ellos (asumiendo que el padre de nuevo no está ignorando SIGCLD) se vuelven hijos del proceso init que los reparte manualmente.
La función fork le devuelve al padre el PID del hijo o –1 si hubo algún error mientras que al hijo le devuelve un 0.
¡Bien! Aquí esta un ejemplo de cómo usar fork():

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

main()
    {
     pid_t pid;
     int rv;
```

```
switch(pid=fork()) {
  case -1:
          perror("fork");  /* algo salió mal */
          exit(1);         /* el padre termina */

   case 0:
     printf(" HIJO: este es el proceso hijo!\n");
     printf(" HIJO: Mi PID es %d\n", getpid());
     printf(" HIJO: El PID de mi padre es %d\n", getppid());
     printf(" HIJO: Ingrese mi estado de salida (hágalo corto): ");
     scanf(" %d", &rv);
     printf(" HIJO: Aquí esta mi estado de salida!\n");
     exit(rv);

   default:
          printf("PADRE: Este es el proceso padre!\n");
          printf("PADRE: Mi PID es %d\n", getpid());
          printf("PADRE: El PID de mi hijo es %d\n", pid);
          printf("PADRE: Ahora espero a mi hijo para salir...\n");
          wait(&rv);
          printf("PADRE:El estado de salida de mi hijo es %d\n",
          WEXITSTATUS(rv));
          printf("PADRE: Aquí estoy afuera!\n");
   }
}
```

El pid_t es un tipo genérico del proceso. Bajo Unix, éste es un short. Para que al llamar al fork() se salve el valor retornado en la variable del pid fork() es sencilla ya que puede devolver sólo tres cosas:
0:
Si devuelve 0, es el proceso hijo. Puede conseguir los PID del padre llamando a la función getppid (). También puede conseguir su propio PID llamando a la función getpid ().

-1:
Si devuelve -1, algo salió mal, y ningún hijo fue creado. Use perror () para ver lo que pasó. Probablemente llenó la tabla de procesos.

Otro valor:
Cualquier otro valor devuelto por fork() significa que es el padre y que el valor devuelto es el PID de su hijo. Ésta es la única manera de obtener los PID de sus hijos,

Cuando el hijo llama finalmente a exit(), el valor retornado llegará al padre cuando este espera. Como usted puede ver de la llamada a la instrucción wait (), hay algunas rarezas que entran juego cuando imprimimos el valor retornado. Nos referimos al uso de WEXITSTATUS (). Esta es una macro que extrae el valor real retornado del hijo a la espera de que sus valores sean ingresados.
Sí, hay más información encerrada en ese entero le permitiré verlo más adelante por sí mismo.
La variable rv es un argumento para la función scanf ( ), un puntero a entero. Scanf devuelve en la variable de memoria (tipo int, en este caso) apuntada por rv, el número que le tipee por teclado, scanf te lo devuelve convertido a entero. O sea scanf = scan with format, como printf, solo que scanf te lee la entrada y printf te escribe en la salida ...
Lo que hace el programa es pedirte el código de retorno. Se lo ingresa por teclado, y retorna con ese parámetro en la función exit (). Nada útil... solo un grupo de excusas para ejercitar programación.... eso es lo que encontrará en la guía de beej.

¿ Se preguntará como hace wait() para saber que proceso espera? Ya que el padre puede tener varios procesos hijos. La respuesta es simple, mis amigos: espera por cualquier hijo que llegue primero para terminar. Si quiere, puede especificar a qué hijo esperar llamando a la instrucción waitpid () con el PID de su hijo como argumento.
Otra cosa interesante para notar del ejemplo anterior es que el padre y el hijo usan la variable rv. ¿Significa esto que es compartido entre los procesos? ¡NO! Si fuera, yo no habría escrito todo este material de IPC.

Cada proceso tiene su propia copia de todas las variables. Hay muchos otros datos que se copian también, pero tendrá que leer las páginas del MAN para ver eso.
Una nota final sobre el programa anterior: Yo utilicé un cambio de declaración para manejar el fork(), y esto no es lo más usual. A menudo verá una declaración tan corta como la siguiente:

```
if (!fork()) {
    printf("Yo soy el hijo!\n");
    exit(0);
} else {
    printf("Yo soy el padre!\n");
    wait(NULL);
}
```

El ejemplo anterior también demuestra esperar con wait () si no le importa el valor que retorna el hijo puede solo llamarla con NULL como argumento.

Conclusiones

El fork() es una system call para crear procesos. Los procesos bifurcados corresponden a una misma copia física del código y los datos apuntada por 2 descriptores de procesos diferentes. (una copia = 2 procesos es lo que caracteriza al Lighweigt process)

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN:

- exit()
- fork()
- ps
- signal()
- signal listing
- wait()
- waitpid()

Señales

Hay un muy fácil, simple, y a veces útil método para que un proceso avise algo a otro: las señales. Básicamente un proceso puede activar una señal y tener un efecto deliberado sobre otro. El handler de la señal (justamente una función) es invocado y el proceso puede manejarlo.
Los procesos pueden señalizarse entre sí o puede hacerlo el kernel. Algunas señales pueden ignorarse, otras pueden interceptarse y cambiar el handler.
Por ejemplo, un proceso podría querer detener a otro, y esto puede ser hecho enviando la señal SIGSTOP. Para continuar, el proceso tiene que recibir la señal SIGCONT. ¿Cómo sabe el proceso que debe hacer cuándo recibe una cierta señal? Bien, muchas señales son predefinidas y el proceso tiene un handler ya establecido para manejarla.

¿Un handler predefinido? Sí. Tome SIGINT por ejemplo. ¡Ésta es la señal de interrupción que un proceso recibe cuando el usuario pulsa ^C. El handler predefinido para SIGINT hace que el proceso termine! ¿Le suena familiar? Bien, como puede imaginarse, puede atropellar la señal SIGINT para hacer cualquier cosa que quiera (o nada en absoluto!) Usted podría tener su proceso de printf ()
Ahora sabe que puede tener su proceso para responder a casi cualquier señal de casi cualquier manera que quiera. Hay excepciones naturalmente, como por otra parte sería demasiado fácil entender. ¿Tome la señal más popular SIGKILL , señal #9. Tecleando KILL –9 mata un proceso en curso.
Usted estaba enviándole SIGKILL. Ahora también podría recordar que ningún proceso puede escapar de un "kill -9",. SIGKILL es una de las señales a las que usted no puede agregar su propio handler para la señal.
El SIGSTOP mencionado también está en esta categoría.
El comando KILL – permite enviar señales desde el prompt.

Además: usted usa a menudo el comando Unix "KILL" sin especificar que está enviando.... ¿ pero que señal es esta ? Respuesta: SIGTERM. Usted puede escribir su propio handler para SIGTERM para que su proceso no responda a un regular "KILL", y el usuario deberá entonces usar "KILL -9" para destruir el proceso.)
¿Todas las señales están predefinidas? ¿Que quiere enviar una señal que tenga importancia y que sólo usted entienda el proceso? Hay dos señales que no están reservadas: SIGUSR1 y SIGUSER2. Usted es libre de usarlas para cualquier cosa que quiera y de manejarlas de la manera que escoja. (Por ejemplo, mi programa de reproductor de CD podría responder a SIGUSR1 adelantando a la próxima pista. De esta manera, yo podría controlarlo de la línea de comandos tipeando KILL -SIGUSR1 nnnn".)

¡ No puede enviar SIGKILL al Presidente!

Como puede suponer comando KILL del Unix es una manera de enviar señales a los procesos. Por pura coincidencia, hay una system call llamada KILL ( ) qué hace la misma cosa. Toma por argumento un número definido y un process ID(como se definió en signal.h). Hay también, una rutina de la librería llamada raise() qué puede usarse para levantar una señal dentro del mismo proceso.
La pregunta ardiente permanece: ¿cómo toma una señal SIGTERM rápidamente? Necesita usar la señal mencionada y pasarle un puntero a la función que quiere que sea su handler.
¿ Nunca usó punteros a función? Debe comprobar la rutina qsort () algún día!). No se preocupe, son simples: si "el foo ("hola"!)"; es una llamada a la función foo (), entonces "foo" es un puntero a esa función. Usted no tiene que usar ni siquiera la dirección del operador.
Sin embargo, aquí está la señal () ruptura:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Bien, la situación básica es esta: vamos a pasar la señal a ser manejada así como la dirección de su handler como argumentos en la llamada a signal (). La función del handler de la señal que usted define toma un solo int como argumento, y retorna un void. Luego, signal () retornará un error, o un puntero a la función del handler previo. Para que tengamos como llamar a signal() la cual acepta como argumentos una señal y un puntero al handler, y retorna un puntero al handler previo.

Afortunadamente usarlo es mucho más fácil de lo que parece. Todo lo que necesita es un handler que tome un entero como argumento y retorne void. ¿Entonces preparar el llamado a una señal es fácil ? Hagamos un programa simple que manejará SIGINT y detendrá el usuario a través de ^C, llamado sigint.doc :
sigint.doc:

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

int main(void)
{
    void sigint_handler(int sig); /* prototipo */
    char s[200];

            /* prepara el handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        perror("signal");
        exit(1);
    }

    printf("Ingrese un string:\n");

    if (gets(s) == NULL)
        perror("gets");
    else
        printf("usted ingresó: \"%s\"\n", s);

    return 0;
}

    /* este es el handler */
```

```
void sigint_handler(int sig)
{
    printf("Ahora no!\n");
}
```

Este programa tiene dos funciones: el main () qué prepara el handler (usando la llamada a la función signal ()), y sigint_handler () qué es el handler de la señal.

¿Qué pasa cuándo lo ejecuta? Si está en medio del ingreso del string y teclea ^C, la llamada a gets () falla y setea la variable global errno de EINTR. Además se llama a sigint_handler () que hace su rutina, para que usted realmente vea:

Ingrese un string:
gets: system call interrumpida

Aquí hay una parte vital de información que omití mencionar antes: ¡cuando el handler de la señal es llamado, el handler particular de esta señal predefinido se reestablece.
El resultado práctico de esto es que nuestro sigint_handler ( ) atraparía el ^C que tecleamos la primera vez, pero no los posteriores. La solución rápida y sucia es restablecer al handler de la señal dentro de sí mismo como se muestra a continuación:

```
void sigint_handler(int sig)
{
    signal(SIGINT, sigint_handler); /* reestablese esta función */
    printf("ahora no!\n");
}
```

El problema con este arreglo es que si ocurre una interrupción y el handler es llamado, pero ocurre otra antes de que la primera pueda restablecer el handler de interrupción, el handler predefinido será el invocado. Sea consciente que si está esperando muchas señales, podría ocasionar problemas.

Todo lo que usted sabía estaba mal

La llamada a la system call signal () es el método histórico para preparar señales. El estándar POSIX ha definido un montón de nuevas funciones para enmascarar las señales que quiera recibir, verifica qué señales están pendientes, y prepara los handler´s de las señales. Dado que muchas de estas llamadas operan en grupos, o juegos, de señales, hay varias funciones que tratan de manipular las señales.

En conclusión, el nuevo método de manejo de señales supera ampliamente al viejo. Yo incluiré una descripción del mismo en una próxima versión de este documento, si el tiempo lo permite.

Algunas señales para hacerlo popular

Aquí hay una lista de señales que (probablemente) tiene a su disposición:

Señal	Description
SIGABRT	Aborta el proceso de la señal.
SIGALRM	Alarma del reloj.
SIGFPE	Operación aritmética errónea.
SIGHUP	Hangup.
SIGILL	Instrucción no válida.
SIGINT	Señal de interrupción de terminal.
SIGKILL	Matar (no puede atraparse ni ignorarse).
SIGPIPE	Escribir en un pipe que no fue leído.
SIGQUIT	Dejar la señal de terminal.
SIGSEGV	Referencia a memoria no válida.

SIGTERM	Señal de terminación.
SIGUSR1	Señal 1 definida por el usuario.
SIGUSR2	Señal 2 definida por el usuario.
SIGCHLD	Proceso hijo finalizado o detenido.
SIGCONT	Continuar la ejecución si estaba stopped.
SIGSTOP	Detener la ejecución (no puede atraparse ni ignorarse).
SIGTSTP	Señal de detención de terminal.
SIGTTIN	Intento de lectura de un proceso background.
SIGTTOU	Intento de escritura de un proceso background.
SIGBUS	Error de bus.
SIGPOLL	Evento detectable por encuesta
SIGPROF	Perfil de tiempo expirado,
SIGSYS	System call erronea
SIGTRAP	Trace / breakpoint trap.
SIGURG	Dato de alta prioridad disponible en un socket
SIGVTALRM	Timer virtual expirado.
SIGXCPU	Tiempo de CPU límite excedido.
SIGXFSZ	Tamaño límite de archivo excedido.

Tabla 1. Señales más comunes

Cada señal tiene su propio handler predefinido, cuya conducta se define en las páginas locales del M.A.N..

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN!

- kill
- kill()
- raise()
- signal()
- signals

Aquí están las páginas del MAN para algunos de los nuevos handler´s de señales:

- sigaction()
- sigprocmask()
- sigpending()
- sigsuspend()
- sigsetops

Pipes

No hay ninguna forma de IPC que sea más simple que los pipes. Llevados a cabo en cada sabor que provee Unix, pipes() y fork () constituyen la funcionalidad detrás del" | " en " ls | y más". Ellos son raramente usados para hacer cosas buenas, pero son un buen método para aprender acerca de los métodos básicos de IPC´s.
Como son muy fáciles, no vamos a emplear mucho tiempo en ellos. Tendremos apenas algunos ejemplos y algún material.

"¡Estos pipes están vacíos!"

¡Espere! No tan rápido. A esta altura yo podría necesitar definir unos "descriptores del archivo". Permítame ponerlo esta manera: ¿qué tanto sabe usted sobre el " FILE * " de stdio.h ? ¿Sabe que tiene todas esas

funciones buenas como el fopen (), fclose (), fwrite (), y así ? Bien, ésas son ahora funciones de alto nivel que se implementan usando descriptores de archivos o “file descriptors”, los que usan system calls tales como open (), creat (), close (), y write ().
Los descriptores de archivo simplemente son enteros que son análogos a losFILE *´s en stdio.h.
FILE * es una estructura definida en POSIX para funciones de manejo de streams (flujos de información) de mas alto nivel como la antes mencionadas fopen, fread, fwrite, etc. Estas funciones permiten algunas operaciones más elaboradas con los descriptores de los archivos.
Las que usamos nosotros son las “primitivas”, por llamarlas de algún modo(al viejo estilo UNIX).

Por ejemplo, stdin es el descriptor de archivo "0", el stdout es el "1", y el stderr es el "2". Del mismo modo, puede abrir cualquier archivo usando fopen () con lo cual obtiene sus propios descriptores del archivo, aunque este detalle esté oculto de para usted. (Este descriptor del archivo puede ser recuperado del FILE * usando la macro fileno () de stdio.h.)

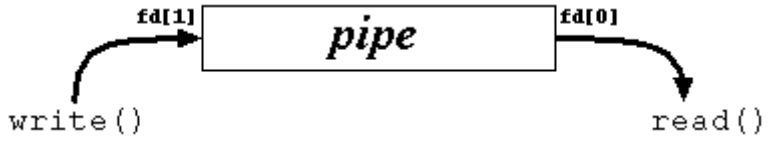

Figura 1. Cómo se organiza un pipe.

Básicamente, una llamada a la función pipe ()retorna un par de file descriptors. Uno de estos descriptores se conecta al extremo de escritura y el otro al de lectura. Algo puede escribirse en un extremo del pipe, y ser leído en el otro extremo en el orden en el que vino. En muchos sistemas, los pipes se llenarán después de que usted escriba aproximadamente 10K en ellos sin leer.
Como un ejemplo inútil, el programa siguiente crea, escribe y lee unos pipes.

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

int main()
{
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("file descriptor para escritura #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("file descriptor para lectura #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("leo \"%s\"\n", buf);
}
```

Como usted puede ver, pipe() toma un array de dos enteros como argumento. Si no acontece ningún error, conecta los dos file descriptors y los devuelve en el array. El primer elemento del array es el file descriptor del extremo de lectura del pipe, y el segundo es del extremo de escritura.

¡fork () y pipe ()-- tienes el poder!

En el ejemplo anterior, resulta difícil ver que tan útiles son. Bien, ya que éste es un documento de IPC, pongamos un fork() en la mezcla y a ver que pasa. Suponga que usted es un agente federal que se asignó para conseguir que un proceso hijo envíe la palabra "test" al padre. No es muy fascinante, pero ni en la informática ni en la vida sería Mulder de los expedientes X..
Primero, tendremos que hacerle un pipe al padre. Luego, haremos un fork (). Ahora, la página del MAN referida al fork () nos dice que el hijo recibirá una copia de los descriptores de archivo del padre, y esto

incluye los del pipe. Ahora el hijo podrá escribir en un extremo del pipe y el padre podrá leerlos por el otro extremo.
Esto se hace así:

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

int main()
{
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" HIJO: escribiendo en el pipe\n");
        write(pfds[1], "test", 5);
        printf(" HIJO: terminando\n");
        exit(0);
    } else {
        printf("PADRE: leyendo el pipe\n");
        read(pfds[0], buf, 5);
        printf("PADRE: leí \"%s\"\n", buf);
        wait(NULL); /* espera que termine un hijo cualquiera */
    }
}
```

Tenga en cuenta que sus programas deben tener muchos más chequeos de errores que los míos. Yo los omito para ayudar a que las cosas queden claras y no complicarlas inútilmente.
De todos modos, este ejemplo es justo como el anterior, excepto que ahora hacemos fork () de un nuevo proceso que escribe en el pipe mientras que el padre lo lee. Lo que se observará será algo como esto:

PADRE: leyendo el pipe.
HIJO: escribiendo en el pipe
HIJO: terminando
PADRE: leo "test"

En este caso, el padre intenta leer del pipe ante que el hijo lo escriba. Cuando esto pasa, se dice que el padre se bloquea, o duerme, hasta que los datos lleguen para ser leídos. Parece que el padre intentó leer, como no había datos se fue a dormir, cuando el hijo terminó de escribir los datos el padre se despertó y los leyó.

Yo apuesto a que todavía está pensando que no hay muchos usos para pipe () y, bien, tiene razón. Las otras formas de IPC´s son generalmente más útiles y a menudo más exóticas.

La búsqueda del pipe como nosotros sabemos

En un esfuerzo para hacerle pensar que los pipes son bestias realmente razonables, yo le daré un ejemplo de cómo usar pipe ( ) en una situación más familiar. El desafío: implementar " ls | wc -l " en C.
Esto requiere del uso de un par de funciones de las que nunca habrá oído hablar: exec () y dup (). La familia de funciones exec ( ) reemplaza el proceso que está corriendo por cualquiera que se pase con exec (). Ésta es la función que usaremos para ejecutar ls y wc -l.
Son funciones que permiten reemplazar un proceso en memoria por otro que se pasa en la lista de argumentos. Lo que se reemplaza es el código y los datos pero se hereda el process descriptor (es decir la misma estructura task_struc) que tenía el proceso reemplazado. Obviamente, al efectuar el reemplazo, se actualizan ciertos campos de task_struc , como por ejemplo, los punteros a las estructuras mm_struc que describen cada bloque de memoria física asignado al proceso por parte del sistema operativo. Se heredan los file descripotrs abiertos (pero OJO no se heredan las variables que los contienen ya que los segmentos del proceso son reemplazados por los del nuevo proceso!!!!), se hereda el PID, el PPID (PID del padre), etc.

dup () toma un descriptor de un archivo abierto que se le pasa como argumento y hace un clon (un duplicado) de él. El duplicado se lo hace en el primer descriptor que encuentra disponible. Así es cómo

conectaremos la salida estándar del ls a la entrada normal de wc. Vea, stdout del flujo del ls dentro del pipe, y el flujo de stdin del wc dentro del pipe. ¡El pipe encaja justo en el medio!

Si antes se hace close (0) o close (1) según corresponda al padre o al hijo y luego dup (), esta última busca el primer file descriptor libre a partir de 0 y copia allí el file descriptor que recibió como argumento.
En ese sencillo acto, redirigió el archivo al descriptor que encontró. En un caso redirige el file descriptor de escritura del pipe a la salida estándar (pantalla) y en el otro redirige la entrada estándar (teclado) al file descriptor de lectura del pipe.
Entonces el efecto logrado es que cuando escriba en el pipe, sale por la pantalla y cuando lea el pipe leerá lo que entra por teclado
De todos modos, aquí está código:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
 int main()
 {
     int pfds[2];

     pipe(pfds);

     if (!fork()) {
         close(1);       /* cerrar la stdout normal */
         dup(pfds[1]);   /* hace lo mismo que stdout con pfds[1] */
         close(pfds[0]); /* no necesitamos esto */
         execlp("ls", "ls", NULL);
     } else {
         close(0);       /* cierra la stdin normal */
         dup(pfds[0]);   /* hace lo mismo que stdin con pfds[0] */
         close(pfds[1]); /* no necesitamos esto */
         execlp("wc", "wc", "-l", NULL);
     }
 }
```

Yo voy a hacer otra nota acerca de la combinación close () /dup () ya que es bastante rara.
close(1) libera al file descriptor 1 (salida estándar). dup(pfds[1]) hace una copia de lo escrito en el final de pipe en el primer file descriptor disponible, que es el 1 ya que acabamos de cerrarlo. De esta manera, algo que ls escribe a la salida estándar (file descriptor 1) cambiará al pdfs[1] (el que escribe al final del pipe).
De igual modo se procede con la sección wc de código de trabajo, excepto que es al revés.

Conclusiones

Probablemente el mejor uso para los pipes es uno al que usted está acostumbrado: enviar a la salida estándar de un comando a la entrada estándar de otro. Para otros usos, es muy limitado y a menudo existen otras técnicas de IPC´s que trabajan mejor.

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN!

- dup()
- exec()
- fileno()
- fork()
- pipe()
- read()
- write()

FIFO´s

Un FIFO es conocida a veces como un pipe nombrado. ¡Es decir, es como un pipe, sólo que tiene un nombre! En este caso, el nombre es el de un archivo que los procesos múltiples pueden abrir con open ( ), leer y escribir.
Este último aspecto de las FIFO´s se diseña para permitirles ir mas allá de una de las limitaciones de los pipes normales:

usted no puede agarrar un extremo de un pipe normal que fue creado por un proceso no relacionado. Vea, si yo ejecuto dos copias individuales de un programa, ambos pueden llamar a la función pipe ( ), y aunque ellos quieran comunicarse entre sí, todavía no pueden. (Esto es porque debe ejecutar pipe () y luego fork ( ) para obtener un proceso hijo que pueda comunicarse con el padre por medio de pipe.) Con las FIFO´s, sin embargo, cada proceso no relacionado puede abrir simplemente un pipe ( ) y transferir datos a través de él.

Una nueva FIFO nace

Puesto que la FIFO realmente es un archivo en disco, usted tiene que hacer algunos manejos creativos para crearla. No es difícil. Apenas tiene que llamar a la función mknod ( ) con los argumentos apropiados. Aquí hay una llamada a mknod ( ) que crea una FIFO:

```
mknod("myfifo", S_IFIFO | 0644 , 0);
```

En el ejemplo anterior, el archivo de la FIFO se llamará "myfifo". El segundo argumento es el modo de creación que se usa para decir le a mknod () que haga una FIFO (S_IFIFO forma parte en la operación OR) y los permisos de acceso para leer a ese archivo (octal 644, o rw-r--r--) qué también puede ponerse usando las macros de sys / stat.h. Este permiso es justamente de la forma que lo pondría usando el comando chmod. Finalmente, se pasa un número de dispositivo. Esto se ignora cuando se crea una FIFO, para poner lo que se quiera.
(Aparte: una FIFO también puede crearse de la línea de comandos usando el Unix mknod command.)
Los permisos de cada fichero se pueden ver con el comando **ls -l**. Para cambiar los permisos de un fichero se emplea el comando chmod, que tiene el formato siguiente:

chmod [quien] oper permiso files

quien Indica a quien afecta el permiso que se desea cambiar. Es una combinación cualquiera de las letras ***u*** para el usuario, ***g*** para el grupo del usuario, ***o*** para los otros usuarios, y ***a*** para todos los anteriores. Si no se da el ***quien***, el sistema supone ***a***.
oper Indica la operación que se desea hacer con el permiso. Para dar un permiso se pondrá un ***+***, y para quitarlo se pondrá un ***-***.
permiso Indica el permiso que se quiere dar o quitar. Será una combinación cualquiera de las letras anteriores : ***r,w,x,s***.
files Nombres de los ficheros cuyos modos de acceso se quieren cambiar. Por ejemplo, para quitar el permiso de lectura a los usuarios de un fichero el comando es:
chmod a -r fichero.txt
Los permisos de lectura, escritura y ejecución tienen un significado diferente cuando se aplican a directorios y no a ficheros normales. En el caso de los directorios el permiso ***r*** significa la posibilidad de ver el contenido del directorio con el comando ***ls***; el permiso ***w*** da la posibilidad de crear y borrar ficheros en ese directorio, y el permiso ***x*** autoriza a buscar y utilizar un fichero concreto.

Productores y Consumidores

Una vez que la FIFO se ha creado, el proceso puede comenzar y puede abrirla para leerla o escribirla usando la system call estándar open( ).
Puesto que es más fácil entender el proceso una vez que se tiene algún código concebido, presentaré aquí dos programas que enviarán datos a través de un FIFO. Uno es speak.c que envía datos a través del FIFO, y el otro se llama tick.c, que saca datos de la FIFO.
Aquí está speak.c:

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

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

#define FIFO_NAME "american_maid"

main()
{
    char s[300];
    int num, fd;

    /* no olvide chequear errores!! */
    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("esperando lectores...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("tengo un lector—tipee algo\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("escribir");
        else
            printf("speak: escribió %d bytes\n", num);
    }
}
```

La condición “while (gets(s), !feof(stdin))” quiere decir mas o menos que mientras por stdin no llegue un EOF ..... hace lo que está entre llaves.
Funciona en base a dos funciones : gets (s) que lee stdin (teclado) y devuelve en s el puntero al array de caracteres leídos, y feof (file descriptor), que evalúa la lectura realizada desde un stream de datos y devuelve 1 cuando se lee el carácter Fin de Archivo. stdin devuelve EOF cuando se pulsa ENTER. De modo que el conjunto hace que mientras no se pulse ENTER se almacenen los caracteres leídos en la string s. Cuando pulse ENTER devuelve el puntero a la string.....

Lo que speak hace es crear la FIFO, entonces intenta abrirla con open (). Ahora, lo que pasará es que el llamado a open () bloqueará hasta que algún otro proceso abra el otro extremo del pipe para leer. (Hay una manera referida a esto—ver debajo O_NDELAY,) Ese proceso es tick.c, mostrado aquí,:

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

#define FIFO_NAME "american_maid"
main()
{
    char s[300];
    int num, fd;

    /* no olvide chequear esto!! */
    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("esperando a quien escriba...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("conseguí un escritor:\n");

    do {
```

```
            if ((num = read(fd, s, 300)) == -1)
                perror("read");
            else {
                s[num] = '\0';
                printf("tick: leí %d bytes: \"%s\"\n", num, s);
            }
        } while (num > 0);
    }
```

Tal como sucedía con speak.c, tick, aquí se bloqueará la ejecución con open ( ) si no hay ni una escritura en la FIFO. En cuanto alguien abra la FIFO para escribir, tick volverá a la vida.
¡Pruébelo! Ejecute speak y se bloqueará hasta que usted arranque tick en otra ventana. (Recíprocamente, si usted empieza tick, bloqueará hasta que usted ejecute speak en otra ventana.) Teclee en la ventana de speak y tick lo tomará.

Ahora, salga de speak. Aviso lo que pasa: los read ( ) en tick retornan 0, lo que significa fin de archivo. De esta manera, el lector puede decir cuando todos los escritores han cerrado su conexión a la FIFO. ¡¿Qué? ¡ Pregunta si puede haber escritores múltiples al mismo pipe? Efectivamente! Eso puede ser muy útil. Quizás muestre después en el documento cómo puede explotarse esta ventaja.
Pero por ahora, permítame terminar este tema viendo lo que pasa cuando sale de tick mientras speak está corriendo. "¡Pipe roto"! ¿Quién hizo esto? Bien, lo que ha pasado es que cuando todos los lectores de una FIFO cierran y el escritor está todavía abierto, el escritor recibirá la señal SIGPIPE la próxima vez que intente escribir.
El handler predefinido de la señal escribe "Pipe Roto" y termina. Por supuesto, usted puede manejar esto más airosamente tomando SIGPIPE a través de la llamada a la función signal ().

Finalmente ¿qué pasa si tiene lectores múltiples? Bien, las cosas extrañas pasan. A veces uno de los lectores consigue todo. A veces alterna entre los lectores. Sin embargo, ¿Para qué quiere tener lectores múltiples ?

¡O_NDELAY! ¡Soy IMPARABLE!

Antes, mencioné que podría bloquear llamando a la función open () si no había ningún lector o escritor correspondiendo. La manera de hacer esto es llamar a open () con el flag O_NDELAY puesta en el argumento del siguiente modo:

```
fd = open(FIFO_NAME, O_RDONLY | O_NDELAY);
```

Esto provocará que open () devuelva -1 si no hay ningún proceso que tenga el archivo abierto para leerlo. Igualmente, puede abrir el proceso del lector usando el flag O_NDELAY, pero esto tiene un efecto diferente: todos los intentos de leer el pipe retornarán 0 bytes leídos si no hay ningún dato en el pipe. (Es decir, los read () ya no bloquearán hasta que halla algún dato en el pipe.) Note que ya no puede decir si read () está devolviendo 0 porque no hay ningún dato en el pipe, o porque el escritor ha terminado. Éste es el precio de poder, pero mi sugerencia es intentar bloquear siempre que sea posible.

Escritores múltiples--¿Cómo hago para multiplicarlos?

Permítame decirle que tiene un pipe con un lector y un escritor conectados a él. ¡No hay ningún problema para el lector, ya que hay sólo un lugar de donde sus datos podrían provenir ¡ (a saber, del un escritor.) De repente otro escritor brinca gruñendo de las sombras! ¡Sin previo aviso empieza a vomitar datos en el pipe! ¿Cómo va a ordenar los datos de los 2 escritores el pobre lector?
Bien; hay muchas maneras, y todas ellas dependen de qué tipo de datos está pasando de un lado a otro. Una de las maneras más simples tiene lugar cuando todos los escritores envían la misma cantidad de datos por vez (supongamos 1024 bytes). Entonces el lector podría leer 1024 bytes en un momento y se asegura que está obteniendo un solo paquete (o en caso contrario 512 bytes de un escritor y 512 de otro.) Sin embargo, todavía, no hay ninguna manera de decir que escritor envió cada paquete.
Una de las soluciones más buenas a esto es usar ,para cada escritor , los primeros bytes de cada paquete para algún tipo de identificador único. El lector puede recoger este identificador y determinar qué escritor envió el paquete. Este "id" puede pensarse como un pequeño encabezado de paquete.

Permitiendo un encabezado del paquete tendremos mucha más flexibilidad en lo que podemos enviar a través de un pipe. Por ejemplo, podría agregar un campo de longitud que le dice al lector cuántos bytes de datos acompañan el encabezado. Una muestra de la estructura de datos que podría tener este paquete sería la siguiente:

```
typedef struct {
        short id;
        short length;
        char data[1024]
    } PACKET;
```

Transmitiendo un paquete con una estructura similar a la anterior, podría tener un número arbitrario de escritores que envían paquetes de longitudes variables. El lector podrá ordenarlos a todos ya que obtiene el "id" de la fuente que lo escribió y la longitud del paquete.

Notas concluyentes

¡Los procesos no relacionados se pueden comunicar vía pipes! (Ésta es una capacidad deseada para el uso de los pipes normales.) Sin embargo, todavía la funcionalidad de los pipes podría no ser realmente la que usted necesita para sus aplicaciones. Las colas del mensaje podrían ser más veloces, si su sistema las soporta.

páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN !

- mknod()
- mknod
- open()
- read()
- signal()
- write()

File locking (archivo cerrado bajo llave)

El file locking proporciona un mecanismo muy simple e increíblemente útil para coordinar los accesos a archivos. Antes de que empiece con los detalles permítanme contarles unos secretos:
Hay dos tipos de mecanismos de cerradura: el mandatory (obligatorio) y el advisory (asesor). Los sistemas mandatory protegerán a los archivos de la escritura y la lectura (write () y read( )).
Varios sistemas de Unix los soportan. No obstante, yo voy ignorarlos a lo largo de este documento, prefiriendo hablar solamente sobre los advisory locks.
Con un advisory lock, los procesos pueden aún leer y escribir en un archivo mientras están lockeados. ¿Inútil? Realmente no, ya que hay una manera de verificar la existencia de un lock antes de que un proceso lo lea o escriba. Vea, es una especie de sistema de cierre cooperativo. Esto es más que suficiente para casi todos casos donde el file locking es necesario.

De ahora en adelante siempre que me refiera a un lock en este documento, me estaré refiriendo a los advisory locks.

Ahora, permítame estropear un poco más el concepto de un lock. Hay dos tipos de advisory locks: read locks y write locks (también llamadas cerraduras compartidas y cerraduras exclusivas, respectivamente.) La diferencia de trabajar con read locks es que ellas no interfieren con otras read locks. Por ejemplo, varios procesos pueden tener un mismo file locking para leerlo. Sin embargo, cuando un proceso tiene un write lock en un archivo, ningún otro proceso puede activar un read lock ni un write lock hasta que se abandone. Una manera fácil de pensar en esto es decir que puede haber lectores múltiples simultáneamente, pero en un momento dado sólo puede haber un escritor.

Una última cosa antes de empezar: hay muchas maneras de hacer un file locking en sistemas Unix. Los sistemas que usan lockf (), personalmente, pienso que son una porquería. Los mejores sistemas soportan flock () qué ofrece mejor control sobre el lock, pero todavía, de cierta forma, le falta.
Para la portabilidad e integridad, hablaré sobre cómo hacer lock files que usan fcntl (). Para usar una función de alto nivel del estilo de flock () que si satisface sus necesidades, quiero demostrar el poder que usted tiene a sus manos. (Si su Sistema Unix no soporta fcntl () de POSIX-y, tendrá que recurrir a la información de lockf () en las páginas del MAN.)

Poniendo una cerradura

La función fcntl () hace casi todo en el planeta, pero nosotros la usaremos apenas para hacer files locking. Poner la cerradura consiste en llenar una struct flock (declarada en fcntl.h) que describe el tipo de cerradura requerido, abrir con open () el archivo con el modo emparejado y llamar a fcntl () con los argumentos apropiados, como estos:

```
struct flock fl;
int fd;

fl.l_type   = F_WRLCK;  /* F_RDLCK, F_WRLCK, F_UNLCK    */
fl.l_whence = SEEK_SET; /* SEEK_SET, SEEK_CUR, SEEK_END */
fl.l_start  = 0;        /* Offset from l_whence         */
fl.l_len    = 0;        /* length, 0 = to EOF           */
fl.l_pid    = getpid(); /* our PID                      */

fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl);  /* F_GETLK, F_SETLK, F_SETLKW */
```

Empecemos con la estructura struct flock ya que los campos en ella se usan para describir la acción de locking que tiene lugar. Aquí están algunas definiciones de los campos:

l_type
Él indica el tipo de cerradura con el contenido que usted le pone. Dicho contenido puede ser F_RDLCK, F_WRLCK, o F_UNLCK si quiere poner una cerradura de lectura, escritura o liberar la cerradura, respectivamente.

l_whence
Este campo determina donde comienza el campo l_start (es como un offset para el offset). Puede ser SEEK_SET, SEEK_CUR, o SEEK_END, para el posicionamiento en el comienzo del archivo, en la posición actual, o en el final.

l_start
Éste es el offset inicial en bytes de la cerradura, respecto del campo l_whence.

l_len
Ésta es la longitud de la región de la cerradura en bytes (la que comienza en l_start que es relativa a l_whence).

l_pid
El process ID del proceso que trata con la cerradura. Use getpid () para obtenerlo.

En nuestro ejemplo, hicimos una cerradura de tipo F_WRLCK (una cerradura de escritura), empezando respecto de a SEEK_SET (el principio del archivo), usando offset 0, longitud 0 (un cero significa “lock al fin de archivo) con el PID seteado por getpid ().

El próximo paso es abrir el archivo, ya que flock () necesita un file descriptor del archivo que está siendo lockeado. Note que cuando abre el archivo, necesita abrirlo en el mismo modo que usó cuando especificó la cerradura, como se muestra en la tabla 1. Si usted abre el archivo en el modo equivocado para un tipo de la cerradura dado, open () devolverá EBADF.

l_type	*modo*
F_RDLCK	O_RDONLY *o* O_RDWR
F_WRLCK	O_WRONLY *o* O_RDWR

Tabla 1. Tipos de cerraduras y sus correspondientes modos de apertura con `open()`.

Finalmente, la llamada a fcntl () realmente setea, libera, u obtiene la cerradura. Vea, el segundo argumento de fcntl () (el cmd) dice qué hacer con los datos pasados en la estructura del tipo struct flock. La siguiente lista resume que hace cada cmd con fcntl ():

F_SETLKW
Este argumento le dice a fcntl () que intenta obtener la cerradura requerida en la estructura flock. Si la cerradura no se puede obtener (por que algún otro la tiene), fcntl () esperará (se bloqueará) hasta que la cerradura se libere entonces él podrá obtenerla. Éste es un comando muy útil. Yo lo uso todo el tiempo.

F_SETLK
Esta función es casi idéntica a F_SETLKW. La única diferencia es que en esta uno no esperará si no puede obtener una cerradura. Si no que retornará inmediatamente un -1. Esta función puede usarse para liberar una cerradura poniendo un F_UNLCK en el campo l_type de la struct flock.

F_GETLK
Si sólo quiere verificar si hay una cerradura, pero no quiere poner una, puede usar este comando. Este comando mira todos los file locks hasta que encuentre uno que esté en conflicto con el tipo de cerradura que especificó en la struct flock. Entonces copia la información de la cerradura en conflicto en la estructura y se la devuelve. Si no puede encontrar una cerradura en conflicto, fcntl () retorna la estructura como usted la pasó, excepto que le pone al campo l_type un F_UNLCK.

En nuestro ejemplo anterior, llamamos a fcntl () con F_SETLKW como argumento, para que bloquee hasta que pueda poner la cerradura, entonces la setea y continúa.

Liberando una cerradura

Después de todo lo que lockeamos llegó el momento de algo fácil: deshacer el locking. Realmente esto es una porción del pastel comparado con lo anterior. Yo solamente volveré a usar el primer ejemplo y agregaré el código para deshacer el locking al final:

```
struct flock fl;
    int fd;

    fl.l_type   = F_WRLCK;  /* F_RDLCK, F_WRLCK, F_UNLCK    */
    fl.l_whence = SEEK_SET; /* SEEK_SET, SEEK_CUR, SEEK_END */
    fl.l_start  = 0;        /* Offset from l_whence         */
    fl.l_len    = 0;        /* length, 0 = to EOF           */
    fl.l_pid    = getpid(); /* our PID                      */

    fd = open("filename", O_WRONLY);  /* obtengo el file descriptor */
    fcntl(fd, F_SETLKW, &fl);  /* seteo el lock, esperando si es necesario/
    .
    .
    .
    fl.l_type   = F_UNLCK;  /* le digo que deshaga el lock en la región */
    fcntl(fd, F_SETLK, &fl); /* seteo dicha acción */
```

Ahora, dejé el viejo código del locking para que compare y note la diferencia, pero usted puede decir que apenas cambié el campo de l_type a F_UNLCK (dejando los otros completamente inalterados!) y llamé a fcntl () con F_SETLK como comando. ¡Es así de fácil!

Un programa de demostración

Aquí, yo incluiré un programa de demostración, lockdemo.c que espera para que el usuario provoque el retorno entonces lockea su propia fuente, espera por otro retorno, luego deshace el locking. Ejecutando este programa en dos (o más) ventanas, usted puede ver cómo los programas interactúan mientras esperan por las cerraduras.

Básicamente, el uso es este: si ejecuta lockdemo sin los argumentos de línea de comando, intentará atraer un write lock (F_WRLCK) en su fuente (lockdemo.c). Si lo empieza a todos con algún argumento en la línea de comando, intentará atraer un read lock (F_RDLCK) en él.

Aquí esta el fuente:

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

int main(int argc, char *argv[])
{
                        /* l_type   l_whence  l_start  l_len  l_pid   */
        struct flock fl = { F_WRLCK, SEEK_SET, 0,       0,     0 };
        int fd;

    fl.l_pid = getpid();

        if (argc > 1)
                fl.l_type = F_RDLCK;

        if ((fd = open("lockdemo.c", O_RDWR)) == -1) {
                perror("open");
                exit(1);
        }

        printf("Presione <RETURN> para intentar obtener el lock: ");
        getchar();
        printf("Intentando obtener el lock...");

        if (fcntl(fd, F_SETLKW, &fl) == -1) {
                perror("fcntl");
                exit(1);
        }

        printf("got lock\n");
        printf("Presione <RETURN> para liberar el lock: ");
        getchar();

        fl.l_type = F_UNLCK;  /* libera el lock en la misma región */

        if (fcntl(fd, F_SETLK, &fl) == -1) {
                perror("fcntl");
                exit(1);
        }

        printf("lock liberado.\n");

        close(fd);
}
```

Compile este cachorro y comience el desastre con un par de ventanas. Note que cuando un lockdemo tiene una read lock, otras instancias del programa pueden obtener sus propias read locks sin problema. Solo cuando se obtiene una write lock es que otros procesos no pueden conseguir una cerradura de ninguna clase.

Otra cosa para notar es que no se puede obtener un write lock si hay una read lock en la misma región del archivo. El proceso que intente conseguir la write lock esperará a que las read locks sean liberadas. Como resultado de esto es que puede obtener un montón de read locks (porque una read lock no detiene a otros

procesos que obtienen read locks) y cualquier proceso que espera por una write lock se sentará allí y se morirá de hambre. No hay ninguna regla que impida agregar más read locks si hay un proceso que espera por una write lock. Debe tener cuidado.
Sin embargo, en la práctica, probablemente use principalmente write locks para garantizar el acceso exclusivo a un archivo por una cantidad corta de tiempo mientras está actualizándose; ése es el uso más común de cerraduras hasta donde yo he visto.

Conclusiones

Las cerraduras gobiernan. A veces, sin embargo, podría necesitar más control sobre sus procesos en una situación de productor-consumidor. Por esta razón, debe ver el documento sobre sistemas de semáforos si su sistema soporta a semejante bestia. Ellos cumplen una función equivalente pero más completa a la de los file locks.

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN!

- `fcntl()`
- `lockf()`
- `lseek()`—para el campo `l_whence` de la `struct flock`
- `open()`

Colas del mensaje

¡Esas personas que nos trajeron Sys V se han visto atacadas para incluir algunos detalles de IPC que se han implementado en varias plataformas (incluso Linux, por supuesto.) Este documento describe el uso y funcionalidad de los fabulosos Sys V y de las Colas de mensaje!
Como de costumbre, yo quiero entregarle alguna apreciación global antes de entrar en el terreno más pantanoso. Una cola de mensaje trabaja en forma similar a una FIFO, pero soporta alguna funcionalidad adicional. También tiene lectura destructiva pero se le puede enviar cualquier cosa (no solo strings). No son nodos del file system sino memoria. Generalmente se sacan mensajes de la cola en el orden en el que fueron colocados. Sin embargo, hay maneras específicas de arrancar ciertos mensajes de la cola antes de que ellos alcancen el frente de la misma. Esto es como cortar la línea.

En términos de uso, un proceso puede crear una nueva cola del mensaje, o puede conectarse a una que exista. Es por esto último que dos procesos pueden intercambiar información a través de la misma cola del mensaje.

Una cosa más sobre los Sys V de IPC: cuando usted crea una cola del mensaje, esta no se marcha hasta que usted la destruya. Todos los procesos que la han usado alguna vez pueden dejarla, pero la cola todavía existirá. Una buena práctica es usar los comandos para IPC´s para verificar si quedan colas del mensaje sin uso que simplemente están merodeando. Es preferible que las destruya con el comando ipcrm a que queden improductivas todas las colas del sistema.

¿Dónde está mi cola?

¡Continuemos! En primer lugar, quiere conectarse a una cola, o crearla si no existe. La función a la que debe llamar para lograr esto es msgget () y debe hacerlo del siguiente modo:

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

msgget () retorna el ID de la cola de mensaje si tuvo éxito, o -1 si fracasó (y por supuesto, setea errno)

Los argumentos son un poco raros, pero pueden entenderse con unos golpes en la frente. El primero, al que llamamos key, es un identificador único de la cola con la que se quiere conectar o que quiere crear. Cualquier otro proceso que quiera conectarse a esta cola tendrá que usar el mismo identificador.

El otro argumento, le dice a msgget ( ) qué hacer con la cola en cuestión. Este campo se completa con el resultado de hacer la operación OR entre IPC_CREAT y los permisos para la cola. (Los permisos de la cola son iguales que los permisos de archivos normales--las colas asumen el identificador de usuario y de grupo del programa que la crea.)
Una muestra de la llamada se da en la siguiente sección.

"¿Es usted el Key Master ?"

¿Qué es esta cosa sin sentido de la clave ? ¿Cómo creamos una ? Bien, ya que el tipo key_t es en realidad un long, usted puede usar el número que quiera. ¿Pero si usted hardcodea este número y algunos otros programas no relacionados hardcodean el mismo número pero necesitan otra cola ?. La solución es usar la función ftok ( ) que genera una clave en base a dos argumentos. Esta función sirve para identificar unívocamente colas de mensaje, semáforos y shared memories y se usa del siguiente modo:

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

Ok, esto está poniéndose raro. Básicamente, el path tiene que ser el de un archivo que este proceso puede leer. El otro argumento, el id , normalmente se pone simplemente un char arbitrario como por ejemplo ' A'. La función ftok () usa información sobre el archivo nombrado (como el inode number, etc.) y el id para generar una clave probablemente-única para msgget (). los programas que quieren usar la misma cola deben generar la misma clave, para lo que deben pasar los mismos parámetros a ftok ().
Finalmente, es hora de hacer la llamada:

```
#include <sys/msg.h>
key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

En el ejemplo anterior, yo puse los permisos para la cola en 666 (o rw-rw-rw -). Y ahora tenemos el msqid que se usará par enviar y recibir mensajes de la cola.

Enviando a la cola

Una vez que se ha conectado a la cola de mensaje usando msgget (), está listo para enviar y recibir mensajes. Primero, el envío:

Cada mensaje se hace en dos partes que se definen en la estructura struct msgbuf como se definió en sys / msg.h :

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

El campo mtype se usa después al recuperar mensajes de la cola, y puede ponerse a cualquier número positivo. El campo mtext es el dato que se agregará a la cola.

¡¿que?! Puede poner sólo una array de bytes en una cola del mensaje? ! Bien, no exactamente. Usted puede usar cualquier estructura que quiera para poner mensajes en la cola, con tal de que el primer elemento sea un long. Por ejemplo, podríamos almacenar toda clase de estructuras :

```
struct pirata_msgbuf {
    long mtype;  /* must be positive */
    char name[30];
    char tipo_de_nave;
    int notoriedad;
    int crueldad;
    int valor_del_botin;
};
```

¿Ok, pero que cómo pasamos esta información a una cola del mensaje ? La respuesta es simple, mis amigos,: sólo usando msgsnd () :

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

msqid es el identificador de la cola de mensaje devuelto por msgget (). msgp es un puntero al dato que quiere poner en la cola. msgsz es el tamaño en bytes de los datos a agregar a la cola.

Finalmente, msgflg le permite poner algunos flags como parámetros optativos que ignoraremos por ahora para poniéndolo a 0.

Y aquí está el código que muestra una de nuestras estructuras pirata agregándose a la cola del mensaje:

```
#include

key_t key;
int msqid;
struct pirata_msgbuf pmb = {2, "L'Olonais", 'S', 80, 10, 12035};

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

msgsnd(msqid, &pmb, sizeof(pmb), 0);  /* pongo el dato en la cola */
```

Recordar la verificación de errores con el valor retornado de todas estas funciones. Oh, sí: nota que yo arbitrariamente he puesto el campo del mtype a 2. Eso será importante en la próxima sección.

Recepción desde la cola

¿Ahora que tenemos al temido pirata Francis L'Olonais pegado en nuestra cola del mensaje, cómo lo sacamos? Como puede imaginar, hay un colega de msgsnd (): es msgrcv (). Que imaginativo.
Una llamada a msgrcv ( ) sería algo así:

```
#include

key_t key;
int msqid;
struct pirata_msgbuf pmb; /* donde se alojará L'Olonais */
key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

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

Hay algo nuevo que notar en la llamada msgrcv () l ¡el 2! ¿Qué significa? Aquí está la sintaxsis 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 pedido. Recuerde que pusimos el mtype arbitrariamente a 2 en msgsnd () en la sección de este documento, para que fuera el que se recupere de la cola.
La conducta de msgrcv () puede ser modificada drásticamente escogiendo el msgtyp positivo, negativo o cero:

msgtyp	**Efectos sobre `msgrcv()`**
cero	Recuperan el próximo mensaje en la cola, sin tener en cuenta su mtype.
Positivo	Obtiene el próximo mensaje con un mtype igual al msgtyp especificado
Negativo	Recupera el primer mensaje en la cola cuyo campo mtype es menor o igual al valor absoluto del argumento msgtyp.

Tabla 1. Efecto del argumento msgtyp sobre msgrcv ( ).

En la mayoría de los casos simplemente querrá el próximo en la cola, sin importar qué valor tiene mtype. En tal caso, pondrá el parámetro msgtyp a 0.

Destrucción de una cola del mensaje

Llega el momento de destruir una cola del mensaje. Como dije antes, las colas que no se eliminen deambularán hasta que explícitamente las quite; es importante que haga esto para que no malgaste los

recursos del sistema. Ok, usted usó esta cola del mensaje todo el día, y está envejeciendo. Por eso quiere borrarla. Hay dos maneras:

1. Usar los comandos IPC de Unix para obtener una lista de las colas de mensaje definidas y luego usar el comando ipcrm para anular la cola.

2. escribir un programa para hacerlo usted mismo.

A menudo, la última opción es el más apropiada, ya podría querer su programa para liberar la cola en algún momento u otro. Para hacer esto requiere la introducción de otra función: msgctl ().
La sintaxis msgctl () es:

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

Por supuesto, msqid es el identificador de la cola obtenido msgget (). El argumento importante es cmd que le dice a msgctl () cómo comportarse. Puede ser una variedad de cosas, pero sólo vamos a hablar sobre IPC_RMID que se usa para remover la cola del mensaje. El argumento buf puede ponerse a NULL para los propósitos de IPC_RMID.

Diga que tenemos la cola que creamos anteriormente para registrar a los piratas. Usted puede destruir esa cola emitiendo la siguiente llamada:

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

Y la cola del mensaje no está más.

Programas de muestra

Incluiré un manojo de programas que se comunicarán usando colas del mensaje. El primero, kirk.c agrega mensajes a la cola, y spock.c los recupera.
Aquí esta el código fuente para 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_msg buf {
    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 lineas de texto, ^D para salir:\n");
```

```
        buf.mtype = 1; /* realmente no nos cuidamos en ente 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 trabajo de kirk es el de permitirle ingresar líneas de texto. Cada línea está atada dentro de un mensaje y se agrega a la cola de mensaje que será luego leída por spock.

La condición “while (gets(s), !feof(stdin))” quiere decir mas o menos que mientras por stdin no llegue un EOF ..... hace lo que está entre llaves.
Funciona en base a dos funciones : gets (s) que lee stdin (teclado) y devuelve en s el puntero al array de caracteres leídos, y feof (file descriptor), que evalúa la lectura realizada desde un stream de datos y devuelve 1 cuando se lee el carácter Fin de Archivo. stdin devuelve EOF cuando se pulsa ENTER. De modo que el conjunto hace que mientras no se pulse ENTER se almacenen los caracteres leídos en la string s. Cuando pulse ENTER devuelve el puntero a la string.....

Aquí está el código para 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) {  /* misma clave que kirk.c */
       perror("ftok");
       exit(1);
        }

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

        printf("spock: listo para recibir mensajes, capitán.\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;
    }
```

Note que en spock, la llamada a msgget ( ), no incluye la opción de IPC_CREAT. Lo pusimos en kirk para crear la cola del mensaje ya que, de lo contrario, spock devolverá un error si kirk no hubiera creado la cola antes.

Aviso lo que pasa cuando está ejecutando las dos en ventanas separadas y mata uno u otro programa. Luego intente correr dos copias de kirk o dos copias de spock para darse una idea de lo que pasa cuando tiene dos lectores o dos escritores. Otra demostración interesante es ejecutar kirk, entre en un manojo de mensajes, luego ejecutar spock y ver si este recupera todos los mensajes. Simplemente enviar mensajes entre estos dos programitas le ayudará a adquirir una comprensión de que sucede realmente.

Conclusiones

Hay mucho más acerca de las colas del mensaje que lo que esta guía didáctica puede presentar. Asegúrese de ver las páginas del MAN para conocer el resto de las cosas que puede hacer sobre todo con la función msgctl ( ). También encontrará allí más opciones con las que puede controlar el manejo de las colas mediante msgsnd () y msgrcv () si la cola está llena o vacía, respectivamente.

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique sus páginas locales del MAN!

- ftok()
- ipcs
- ipcrm
- msgctl()
- msgget()
- msgsnd()

Semáforos

¿Recuerda los file´s locking? Bien, los semáforos pueden pensarse como mecanismos file locking de tipo advisory muy genéricos. Usted puede usarlos para controlar el acceso a los archivos, memoria compartida, y, bueno, casi cualquier cosa que quiera. La funcionalidad básica de un semáforo es que usted pueda ponerlo, verificarlo, o esperarlo hasta que se libere para luego tomarlo ("test-n-set"). Sin importar cuan complejo sea el material que sigue permite recordar esos tres funcionamientos.
Este documento proporcionará una apreciación global de funcionalidad del semáforo, y acabará con un programa que usa semáforos para controlar el acceso a un archivo. (Esta tarea, reconocida, podría manejarse fácilmente con file locking, pero es un buen ejemplo ya que es más fácil para darle una idea respecto a lo que es memoria compartida o shared memory).

Agarrando algunos semáforos

Con los Sys V de IPCs, usted no agarra solo semáforos; si no juegos de semáforos. Puede, por supuesto, agarrar un juego de semáforos que sólo tenga un semáforo en él, pero el punto es que puede tener un montón de semáforos muertos con solo crear un juego de ellos.

¿Cómo crea el juego de semáforos? Se hace con una llamada a la función semget () que retorna el identificador del semáforo (de ahora en adelante llamado el semid):

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

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

¿Cuál es la clave? Es un único identificador que es usado por diferentes procesos para identificar este juego de semáforos. (Esta clave la genera ftok () del mismo modo que se describió en el documento de Colas de Mensaje.)

El próximo argumento, nsems, es (como lo supuso) el número de semáforos en este juego. El número exacto depende del sistema, pero probablemente está entre 500 y 2000. Si necesita más (! desgraciado insaciable!), simplemente consiga otro juego de semáforos.

Finalmente, el argumento semflg. Esto dice a semget () que permisos deben darse en el nuevo juego de semáforos, si usted está creando un nuevo juego o simplemente quiere conectarse a uno que existe y otras cosas que le parezcan. Para crear un nuevo juego, puede hacer la operación OR a los permisos de acceso con IPC_CREAT.

Aquí hay un ejemplo de cómo llamar a ftok () para generar una clave y crear un set de 10 semáforos con permisos en 666 (o rw-rw-rw -) :

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

key_t key;
int semid;

key = ftok("/home/beej/somefile", 'E');
semid = semget(key, 10, 0666 | IPC_CREAT);
```

¡Felicitaciones! ¡Ha creado un nuevo juego de semáforos! Después de ejecutar el programa puede comprobarlo con los comandos de IPC. (No se olvide de quitarlos con ipcrm !)

semop () : ¡ el poder Atómico!

Todas las operaciones que crean semáforos, los setean o hacen con ellos test – n - set usan la system call semop (). Esta system call es de propósito general y su función es dictada por una estructura struct sembuf que se le pasa:

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

Por supuesto, sem_num es el número de semáforos en el juego que usted quiere manipular. Luego, sem_op es lo que usted quiere hacer con ese semáforo. Esto asume significados diferentes que dependiendo de que sem_op sea positivo, negativo, o cero, como se muestra en la siguiente tabla :

sem_op	Que sucede
Positivo	El valor de sem_op se suma al valor del semáforo. Así es cómo un programa usa un semáforo para marcar un recurso como tomado..
Negativo	Si el valor absoluto de sem_op es mayor que el valor del semáforo, la llamada al proceso se bloqueará hasta que el valor del semáforo alcance al valor absoluto de sem_op. Finalmente, el valor absoluto de sem_op se restará al valor del semáforo. Así es cómo un proceso libera un recurso guardado por el semáforo.
Cero	Este proceso esperará hasta que el semáforo en cuestión alcance el 0.

Tabla 1. El valor de sem_op y sus efectos.

Básicamente, lo que está haciendo es cargar a una estructura struct sembuf con algunos valores que quiere y luego llama a semop () así:

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

El argumento de semid es el número obtenido de la llamada a semget (). El siguiente argumento es sops, que es un puntero para la estructura sembuf que usted llenó con sus comandos al semáforo.

Sin embargo, si quiere, puede hacer un array de estructuras sembufs para hacer un manojo de operaciones de semáforo al mismo tiempo. La manera en que semop () sabe que quiere hacer es mediante el argumento nsop que dice cuántas estructuras sembuf le está enviando. Si sólo tiene una ponga un 1 como argumento.

Un campo de la estructura sembuf que no he mencionado es el campo sem_flg que le permite al programa especificar flags que modifican mucho los efectos de la llamada a semop ().

Uno de estos flags es el IPC_NOWAIT que, como el nombre sugiere, hace que la llamada a semop () retorne el error EAGAIN si encuentra una situación en la que normalmente se bloquearía. Esto es bueno para las situaciones en la que necesita hacer un polling para ver si puede disponer de un recurso.

Otro flag muy útil es SEM_UNDO. Este hace que semop () grabe, en cierto modo, el cambio que le hizo al semáforo. Cuando el programa termina, el kernel deshará automáticamente todos los cambios que fueron marcados con el flag SEM_UNDO. Por supuesto, su programa deberá hacer algo mejor para des-asignar recursos marcados usando el semáforo, pero a veces esto no es posible cuando su programa obtiene un SIGKILL o le sucede algún otro percance.

Destruyendo un semáforo

Hay dos maneras de librarse de un semáforo: uno es usar el comando ipcrm de Unix. El otro es a través de una llamada a semctl () con los argumentos apropiados.

Ahora intento compilar este código tanto bajo Linux como bajo HPUX, pero encuentro que las system call´s difieren. Linux pasa un unión semun a semctl (), pero HPUX usa en su lugar una lista de argumentos variables. Intentaré proporcionar un código claro para ambos, pero daré prioridad al estilo Linux, ya que así se describe en el libro de Steven llamado Unix Network Programming.

Aquí la unión semun al estilo Linux-estilo, junto con la llamada a semctl () que destruirá el semáforo:

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

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

Simplemente note que la unión semun proporciona una manera de pasar tanto un int, como una estructura semid_ds o un puntero a ushort. Esta es la flexibilidad que la versión de HPUX de semctl () logra con una lista del argumentos variables:

```
    int semctl(int semid, int semnum, int cmd, ... /*arg*/);
```

En HPUX, en lugar de pasar una unión semun, pasa simplemente cualquier valor (entero o de otra clase). Para obtener más información sobre su sistema específico debe chequear las páginas del MAN. No obstante, el código de aquí en adelante será al estilo Linux.

¿Dónde estábamos ...? Oh sí--destruyendo un semáforo. Básicamente, debe obtener el ID que llamamos semid para el semáforo que quiere. cmd debe cargarse con IPC_RMID que le dice a semctl () que quite este juego de semáforos. Los dos parámetros semnum y arg no tienen ningún significado en el contexto de IPC_RMID y puede ponerse cualquier cosa en su lugar.

Aquí se da un ejemplo para un juego de semáforos:

```
    union semun dummy;
    int semid;
    .
    .
```

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

Advertencia

Cuando crea algunos semáforos todos ellos están inicializados en cero. Es una pena ya que significa que están todos marcados como asignados; entonces requiere de otra llamada (a semop () o a semctl () para marcarlos como libres). ¿Qué significa esto? Bien, significa que la creación de un semáforo no es atómica (en otras palabras, no es un proceso del un solo paso). Si dos procesos están intentando crear, inicializar o usar un semáforo al mismo tiempo podría darse una condición de competencia entre procesos.

Voy a referirme a este problema en el código del programa de muestra teniendo un solo proceso que crea e inicializa el semáforo. El proceso principal apenas lo accede, pero nunca lo crea o lo destruye. Simplemente esté proceso es el guardia. Stevens se refiere a esto como la "falla fatal" del semáforo.

Programas de muestra

Hay tres de ellos y todos compilarán bajo Linux (y HPUX con modificaciones). El primero, seminit.c, crea e inicializa el semáforo. El segundo, semdemo.c, pretende realizar algunos files locking usando el semáforo, en una demostración muy buena como en el documento de file locking.
Finalmente, semrm.c se usa para destruir el semáforo (de nuevo, podría usarse ipcrm para lograr esto.)
La idea es ejecutar seminit.c para crear el semáforo. Intente usar los ipcs de la línea de comandos para verificar que existan. Luego ejecutar semdemo.c en un par de ventanas y ver cómo actúan recíprocamente. Finalmente, use semrm.c para quitar el semáforo. Usted podría probar también quitando el semáforo mientras semdemo.c está corriendo sólo para ver que clase de errores se generan.

Aquí está seminit.c (¡ ejecute éste primero!):

```
#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("semdemo.c", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

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

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

```
    return 0;
}
```

Aquí está 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};  /* seteados para asignar recursos */

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

    /* toma el juego de semáforos creados por seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    printf("Presione return para lockear: ");
    getchar();
    printf("Intentando lockear...\n");

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

    printf("Lockeado.\n");
    printf("Presione return para deslockear: ");
    getchar();

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

    printf("deslockeado\n");

    return 0;
}
```

Aquí está 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("semdemo.c", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }
    /* toma el set de semáforos creado por seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

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

    return 0;
}
```

¡No es divertido! ¡ estoy seguro que quedará temblando después de jugar con todo el día con este material de semáforos.

Conclusiones

Hmmm. Pienso que he subestimado la utilidad de semáforos. Le aseguro que son muy muy muy útiles en una situación del concurrencia. A menudo son más rápidos que los file locking regulares, también. ¡También puede usarlos en otras cosas que no son archivos, como en segmentos de Memoria Compartida! De hecho, a decir verdad, a veces es difícil vivir sin ellos.

Siempre que tenga corriendo múltiples procesos a través de una sección crítica del código necesita semáforos.

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN!

- ipcrm
- ipcs
- semctl()
- semget()
- semop()

Segmentos de Memoria compartida

Lo bueno de los segmentos de memoria compartida es que son lo que parecen: un segmento de memoria que es compartido entre los procesos. ¡Quiero decir, piense en el potencial de esto! ¡Usted podría asignar un bloque de información del jugador para un juego de múltiples jugadores el cual podría ser accedido por cada proceso a voluntad! Diversión, diversión, diversión.

Hay, como de costumbre, más detalles molestos que considerar, pero a la larga son todos bastante fáciles de superar. Vea, solo se conecta al segmento de memoria compartida, y obtiene un puntero a la memoria. Puede leer y escribir a donde apunta y todos los cambios que usted hace serán visibles para todos los demás que estén conectados al segmento. No hay nada más simple.

Creando el segmento y conectando

De forma similar a otros de los 5 sistemas de IPC, un segmento de memoria compartida se crea y se conecta por vía de una llamada a una función, en este caso a shmget ():

```
int shagged(key_t key, size_t size, int shmflg);
```

Si la función shmget () resultó exitosa devuelve un identificador para el segmento de memoria compartida. El argumento key debe crearse del mismo modo que se mostró cuando en el documento de Colas de Mensaje y usando ftok (). El próximo argumento, size es el tamaño en bytes del segmento de memoria compartida.
Finalmente, a shmflg deben darse el resultado de la operación OR entre los permisos del segmento e IPC_CREAT si quiere crear el segmento, pero puede ser por otra parte 0. (No hace mella especificar IPC_CREAT cada vez que se conecte si el segmento ya existe.)

Aquí hay una llamada de ejemplo que crea un segmento de 1K con 644 como permisos (rw-r--r--):

```
key_t key;
int shmid;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

¿Pero cómo recibe usted un puntero para manejar los datos identificados por shmid? La respuesta está en la llamada a shmat (), en la siguiente sección.

Átáchame--consiguiendo un puntero al segmento

Antes de que pueda usar un segmento de memoria compartida tiene que atacharse a él por medio de una llamada a la función shmat ():

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

¿Qué significa todo esto? Bien, shmid es el identificador de la memoria compartida que recibió de la llamada a shmget (). El siguiente parámetro es shmaddr que puede utilizar para decirle a shmat () qué dirección específica quiere usar si es que quiere escoger una dirección y debe ponerlo simplemente en 0 para permitirle al SO que escoja la dirección por usted.
Finalmente, shmflg pueden ponerse a SHM_RDONL Y si sólo quiere leer de él, o 0 en otro caso.
Aquí está un ejemplo más completo de cómo obtener un puntero a un segmento de memoria compartida:

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

¡Usted tiene el puntero al segmento de memoria compartida! Note que shmat () devuelve un puntero a void, y nosotros estamos tratándolo, en este caso, como un puntero a char. Usted puede tratarlo como lo que quiera, dependiendo de que clase de datos tiene allí. Los punteros a arrays de estructuras solo aceptan que se los trate así nada más.
También, es interesante notar que shmat () retorna -1 en caso de fracaso. ¿Pero obtiene un –1 de un puntero a void? Simplemente haga un casting durante la comparación en el chequeo de errores:

```
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1))
    perror("shmat");
```

Todo lo que tiene que hacer ahora es cambiar los datos al estilo de un puntero normal. Hay algunos ejemplos de muestras en la próxima sección.

Lectura y Escritura

Permítame decirle que tiene el puntero a los datos del ejemplo anterior. Es un puntero a char, por que estaremos leyendo y escribiremos char´s con él. Además, por simplicidad, permítame suponer que el segmento de memoria compartida de 1K contiene un string nulo como terminación.
No podría ser más sencillo. Puesto que hay justo un string allí, podemos imprimir algo como esto:

```
printf("la memoria compartida contiene: %s\n", data);
```

Y podríamos guardar algo en dicha memoria tan fácilmente como hacer esto :

```
printf("Ingrese un string: ");
gets(data);
```

Por supuesto, como dije antes, puede tener otros datos allí además de sólo char´s. Yo estoy usándolos simplemente como un ejemplo. Yo asumiré que usted está bastante familiarizado con los punteros en C y que podrá tratar con cualquier tipo de datos que quiera poner allí.

Desatachando y borrando segmentos

Cuando usted se hace con un segmento de memoria compartida, su programa debe desatacharse de él usando una llamada a la función shmdt ():

```
int shmdt(void *shmaddr);
```

El único argumento, shmaddr, es la dirección que recibió del shmat (). La función devuelve -1 en caso de error y 0 en caso de éxito.
Cuando se desatacha del segmento, éste no se destruye. Ni se quita cuando todos se desatachan de él. Tiene que destruirlo utilizando una llamada específica a shmctl (), de manera similar a las llamadas de control para cualquiera de los otros Sys V de IPC:

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

La llamada anterior anula el segmento de memoria compartida y asume que nadie más se atacha a él. La función shmctl () hace mucho más que esto.

Como siempre, puede destruir el segmento de memoria compartida desde la línea de comandos usando el comando ipcrm de Unix. Asegúrese también de no dejar ningún segmento de memoria compartida sin usar ya que permanecerá derrochando recursos del sistema.
Cada uno de los Sys V de IPC puede ser objeto de los comandos ipc que usted posee.

Problemas de concurrencia

¿Cuáles son problemas de concurrencia ? Bien, desde que tiene procesos múltiples que modifican el segmento de memoria compartida, es posible que ciertos errores pudieran surgir cuando ocurre que se actualiza al segmento en forma simultánea. Este acceso coexistente casi siempre es un problema cuando tiene múltiples escritores a un objeto compartido.

La manera de tratar esto es usar semáforos para lockear el segmento de memoria compartida mientras un proceso está escribiéndolo. (A veces la cerradura abarca tanto la escritura como la lectura sobre la memoria compartida, dependiendo de lo que se esté haciendo.)

Una verdadera discusión de concurrencia está más allá del alcance de este documento, y usted podría querer consultar uno de los muchos libros que atañen a este asunto. Yo le diré apenas esto: si empieza a tener inconsistencias raras en sus datos compartidos cuando conecta dos o más procesos a él, bien puede tener un problema de concurrencia.

Código de muestra

Ahora que lo he iniciado en todos los peligros del acceso coexistente a un segmento de memoria compartida sin usar semáforos, le daré una demostración de cómo es eso. Ya que ésta no es una aplicación cuya misión sea crítica, y es improbable que usted acceda a los datos compartidos en forma simultanea con cualquier otro proceso, por simplicidad, omitiré los semáforos.

Este programa hace una de dos cosas: si usted lo ejecuta sin los parámetros de línea de comandos, imprime el contenido del segmento de memoria compartida. Si le da un parámetro de línea de comandos guarda el parámetro en el segmento de memoria compartida.

Aquí está el código para shmdemo.c:

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

#define SHM_SIZE 1024  /* segmento de memoria compartida de 1K */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "uso: shmdemo [dato_a_escribir]\n");
        exit(1);
    }

    /* hace la clave: */
    if ((key = ftok("shmdemo.c", 'R')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* se conecta al segmento(y posiblemente lo crea) : */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }

    /* se atacha al segmento para conseguir un puntero a este: */
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    /* lee o modifica el segmento, en base a la línea de comandos: */
    if (argc == 2) {
        printf("escribir para el segmento: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("el segmento contiene: \"%s\"\n", data);

    /* se desatacha del segmento: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
```

```
        }

        return 0;
    }
```

Normalmente, un proceso se atachará al segmento y correrá durante un rato mientras otros programas están cambiando y leyendo el segmento compartido. Esto se cuida para que el proceso vea una actualización del segmento y los cambios sean vistos también por los otros procesos. Nuevamente, por simplicidad, el código de muestra no lo hace, pero usted puede ver cómo los datos son compartidos entre los procesos independientes.
Tampoco hay aquí ningún código para remover el segmento, debe asegurarse de hacerlo usted.

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN!

- ftok()
- ipcrm
- ipcs
- shmat()
- shmctl()
- shmdt()
- shmget()

Archivos mapeados en memoria

Llega el momento en el que usted quiere leer y escribir los archivos para que la información sea compartida entre los procesos. Piense en ellos de esta manera: dos procesos que abren el mismo archivo, los dos lo leen y lo escriben y comparten así la información. El problema es que a veces es un sufrimiento hacer todos esos fseek ()s. ¿No sería más fácil si pudiera simplemente mapear una sección del archivo en memoria y conseguirle un puntero para ello? Entonces podría simplemente usar un puntero aritmético para acceder a los datos del archivo (y modificarlos).

Bien, esto es exactamente lo que un archivo mapeado en memoria. Y también es muy fácil de usar. Unas pocas y simples llamadas, mezcladas con unas pocas reglas simples, y usted estará en condiciones de mapear memoria como loco.

Trazado del mapa de memoria

Antes de mapear un archivo en la memoria, necesita obtener un file descriptor para ser usado por la system call open ( ):

```
    int fd;

    fd = open("mapdemofile", O_RDWR);
```

En este ejemplo, hemos abierto el archivo para acceso del lectura / escritura. Usted puede abrirlo en cualquier modo que quiera, pero tiene que coincidir con el modo especificado en el parámetro prot para la llamada a la función mmap () que se hará debajo.

Para mapear un archivo en memoria usted usa la system call mmap ( ) que está definida del siguiente modo:

```
    void *mmap(void *addr, size_t len, int prot, int flags,
               int fildes, off_t off);
```

¡Que montón de parámetros! Aquí se detallan todos ellos:

addr

Ésta es la dirección en la que queremos mapear el archivo. La mejor manera de usarlo es ponerlo a 0 (caddr_t) y permitirle al SO escogerlo por usted. Si usted le dice que use una dirección que al SO no le gusta (por ejemplo, si no es un múltiplo del tamaño de una página de memoria virtual), le dará un error.

len

Este parámetro es la longitud de los datos que queremos mapear en la memoria. Ésta puede ser cualquier longitud que usted quiera. (Aparte: si el len no un múltiplo del tamaño de una página de memoria virtual, obtendrá un tamaño de bloque que depende del redondeado de ese tamaño. Los bytes extra serán 0, y cualquier cambio que le haga a ellos no modificará el archivo.)

prot

El argumento “protección” le permite especificar qué tipo de acceso tiene este proceso para la región de memoria mapeada. Ésta puede surgir del resultado de la operación OR entre los siguientes valores: PROT_READ, PROT_WRITE, y PROT_EXEC, para permisos de lectura, escritura, y ejecución, respectivamente. El valor especificado aquí debe ser equivalente al modo especificado en la system call open ( ) que es usada para obtener el file descriptor.

flags

Aquí simplemente hay banderas misceláneas que pueden ponerse para la system call. Usted querrá ponerle el valor MAP_SHARED si planea compartir sus cambios al archivo con otros procesos, o de lo contrario MAP_PRIVATE. Si usted pusiera este último valor, su proceso conseguirá una copia de la región mapeada, para que no se reflejará cualquier cambio que usted hace en el archivo original--así, otros procesos no podrán verlos. Aquí no hablaremos en absoluto sobre MAP_PRIVATE ya que no tiene mucho que ver con IPC.

fildes

Aquí es que donde pone ese file descriptor que antes abrió.

off

Éste es el offset dentro del archivo por el que usted quiere empezar a mapear. Una restricción: éste debe ser un múltiplo del tamaño de una página de memoria virtual. Este tamaño de la página puede obtenerse con una llamada a getpagesize ().

En caso de error mmap ( ),como lo habrá supuesto, devolverá -1 y seteará la variable errno. De otro modo devuelve un puntero al comienzo de los datos mapeados.

Haremos una demostración corta que mapea la segunda “página” de un archivo en la memoria. Primero lo abriremos con open() para obtener los file descriptors, luego usaremos getpagesize () para obtener el tamaño de una página de memoria virtual y usar este valor tanto para len y como para off. De esta manera, empezaremos mapeando a la segunda página para una longitud de una página. (En mi Linux box, el tamaño de la página es de 4K.)

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>

int fd, pagesize;
char *data;

fd = fopen("foo", O_RDONLY);
pagesize = getpagesize();
```

```
data = mmap((caddr_t)0, pagesize, PROT_READ, MAP_SHARED, fd, pagesize);
```

Una vez que este código haya corrido puede acceder al primer byte de la sección del archivo mapeada que usa data[0]. Note que aquí hay mucha conversión de tipos. Por ejemplo, mmap ( ) devuelve a caddr_t, pero lo tratamos como un char *. Bien, el hecho es que normalmente a ese caddr_t se lo define para que sea un char *, que para la mayoría de los casos está bien.
También debe notar que hemos mapeado el archivo PROT_READ para tener acceso solo para lectura. Cualquier esfuerzo por escribir a los datos (data[0] = ' B', por ejemplo) causará una violación de la segmentación. Si usted quiere acceso a los datos para lectura y escritura abra el archivo como O_RDWR con prot puesto a PROT_READ | PROT_WRITE.

Desmapeando el archivo

Por supuesto que hay una función munmap () para archivos mapeados en memoria:

```
int munmap(caddr_t addr, size_t len);
```

Esta función simplemente desmapea la región apuntada por addr (devuelta por mmap ()) con una longitud len (la misma pasada a mmap ()). La función munmap () devuelve -1 en caso de error y setea la variable errno.

Una vez que tiene desmapeado al archivo, cualquier esfuerzo por acceder a los datos a través del viejo puntero producirá una falta de segmentación. ¡Usted ha sido advertido!

Una nota final: el archivo será desmapeado automáticamente si su programa termina, por supuesto.

¡Problemas de concurrencia, otra vez?!

Si tiene procesos múltiples que manejan los datos concurrentemente en el mismo archivo, podría estar en problemas. Podría tener que lockear el archivo o usar semáforos para regular el acceso al mismo mientras un proceso accede a él. Mire el documento de Memoria Compartida para obtener más información acerca de los problemas de concurrencia.

Una muestra simple

Bien, de nuevo es tiempo del código. Yo tengo aquí un programa de demostración que mapea su propia fuente en la memoria e imprime el byte que se encuentra a un offset especificado por linea de comandos.

El programa restringe los desplazamientos que usted puede especificar al rango de 0 la longitud del archivo. La longitud del archivo se obtiene a través de una llamada al stat ( ) qué seguramente jamás la habrá visto antes. Esta función devuelve una estructura llena de información del archivo, entre la que hay un campo que contiene el tamaño en bytes. Bastante fácil.
Aquí está el código fuente para mmapdemo.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    int fd, offset;
    char *data;
    struct stat sbuf;

    if (argc != 2) {
```

```
            fprintf(stderr, "uso: el offset mmapdemo \n");
            exit(1);
        }

        if ((fd = open("mmapdemo.c", O_RDONLY)) == -1) {
            perror("open");
            exit(1);
        }

        if (stat("mmapdemo.c", &sbuf) == -1) {
            perror("stat");
            exit(1);
        }

        offset = atoi(argv[1]);
        if (offset < 0 || offset > sbuf.st_size-1) {
            fprintf(stderr, "mmapdemo: el offset debe estar entre 0 y
                                        %d\n", \sbuf.st_size-1);
            exit(1);
        }

        if ((data = mmap((caddr_t)0, sbuf.st_size, PROT_READ, MAP_SHARED,
            \fd, 0)) == (caddr_t)(-1)) {
            perror("mmap");
            exit(1);
        }

        printf("el byte al offset %d es '%c'\n", offset, data[offset]);

        return 0;
    }
```

Esto es todo. Compile esto y ejecútelo con alguna línea del comando como:

```
$ mmapdemo 30
el byte al offset 30 es 'e'
```

Yo lo dejaré a usted escribir programas nuevos que usen algunas de estas system calls.

Conclusiones

Los archivos mapeados en memoria pueden ser muy útiles, sobre todo en sistemas que no soportan los segmentos de memoria compartida. De hecho, los dos son muy similares. (Los archivos mapeados en memoria son también enviados al disco, lo que puede ser una ventaja) Con los files locking, con los semáforos o con los archivos mapeados en memoria los múltiples procesos pueden compartir fácilmente los datos.

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguiente páginas locales del MAN!

- getpagesize()
- mmap()
- munmap()
- open()
- stat()

Unix Sockets

¿Recuerda a las FIFOs? ¿Recuerda que podía enviar datos en una sola dirección, como un pipe? ¿No sería grandioso poder enviar datos en ambas direcciones como con un socket?

Bien, no espere más, porque la respuesta está aquí: ¡Unix Sockets! En caso de que usted todavía esté preguntándose que es un socket le digo que es que un pipe de comunicación bidireccional que puede usarse para comunicar en una variedad de dominios. Uno de los más comunes dominios comunicados por sockets es Internet, pero nosotros no discutiremos esto aquí. Nosotros trataremos los dominios de Unix, es decir, los sockets que pueden usarse entre los procesos de un mismo sistema Unix.

Los sockets de Unix usan, en muchos casos, las mismas funciones llamadas por los sockets de Internet pero yo no voy a describir todas las funciones llamadas en detalle sino que solo trataré las que use dentro de este documento. Si la descripción de una cierta llamada es demasiado vaga (o si usted quiere aprender más sobre los sockets de Internet), por favor vea la Beej's Guide to Network Programming Using Internet Sockets para tener una información más detallada.

Reseña

Como dije antes, los sockets de Unix son como FIFOs bidireccionales. Sin embargo, todos los datos de la comunicación serán tomados de la interfase de los sockets en lugar de la interfase del archivo. Aunque los sockets de Unix son archivos especiales en file systems (como FIFOs), usted no podrá utilizar open ( ) y read ()-- usará socket ( ), bind (), recv (), etc.

Al programar con socket, normalmente creará programas servidores y programas clientes. El servidor permanecerá escuchando las conexiones entrantes de los clientes y los manejará. Esto es muy similar a la situación que existe entre los sockets de Internet, pero con algunas diferencias.

Por ejemplo, cuando dice qué socket de Unix quiere usar (es decir, el path del archivo especial que es el socket), recurrirá a una estructura sockaddr_un la cual tiene los siguientes campos

```
struct sockaddr_un {
        unsigned short sun_family;  /* AF_UNIX */
        char sun_path[108];
  }
```

Ésta es la estructura que pasará a la función bind ( ) que asocia un socket descriptor (un file descriptor) con un cierto archivo (cuyo nombre está el campo sun_path).

Que hacer para ser un Servidor

Sin entrar en demasiados detalles, yo perfilaré los pasos que normalmente da un programa para ser un servidor. Mientras estoy en él, intentaré llevar a cabo un "eco de servidor" (eco server) el que cual justamente retorna un eco de cualquier cosa que obtenga de un sockets.

Aquí están los pasos del servidor:

1. **Llama a la función *socket () :*** una llamada a socket ( ) con los argumentos apropiados crea un Unix socket

```
unsigned int s, s2;
struct sockaddr_un local, remote;
int len;
```

s = socket(AF_UNIX, SOCK_STREAM, 0);

El segundo argumento, SOCK_STREAM, le dice a socket ( ) que debe crear un stream socket. Los datagram sockets (SOCK_DGRAM) también son soportados en el dominio de Unix, pero aquí solo voy a cubrir los stream sockets.

Para los curiosos que quieran tener una buena descripción de los datagram sockets desconectados , ver la Beej's Guide to Network Programming que se aplican absolutamente bien a los sockets de Unix. Lo único que cambia es que en lugar de usar una estructura sockaddr_un debe usar una estructura sockaddr_in

Una nota más: todas estas llamadas devuelven -1 en caso de error y setean la variable global errno para reflejar que algo salió mal. Asegúrese de hacer un chequeo de errores.

2. **Llama a la función bind ( ) :** Usted recibió un socket descriptor de la llamada a socket ( ), ahora debe ligarlo a una dirección en el dominio de Unix. (Esa dirección, como dije antes, es un archivo especial en el disco.)

```
local.sun_family = AF_UNIX;  /* local es declarada antes de socket() ^ */

local.sun_path = "/home/beej/mysocket";
          unlink(local.sun_path);
          len = strlen(local.sun_path) + sizeof(local.sun_family);
          bind(s, (struct sockaddr *)&local, len);
```

Esto asocia los sockets descriptors con la dirección de la Unix "/home/beej/mysocket". Note que llamamos a la función unlink ( ) antes de bind () para quitar el socket si ya existe. Usted obtendrá un error EINVAL si el archivo ya existía.

3. **Llama a la función *listen ( )* :** Esta instruye al socket para que escuche a las conexiones entrantes de los programas clientes:

```
          listen(s, 5);
```

El segundo argumento, 5, es el número de conexión entrante que pueden estar en la cola antes de que usted ejecute la llamada a la función accept ( ). Si hay muchas conexiones que esperan ser aceptadas, los clientes adicionales generarán el error ECONNREFUSED.

4. **Llama a la función accept ( ) :** Esto aceptará una conexión de un cliente. ¡Esta función devuelve otro socket descriptor ! El descriptor viejo todavía está escuchando a las nuevas conexiones, pero el nuevo se conecta al cliente:

```
len = sizeof (struct sockaddr_un);
s2 = accept (s, &remote, &len);
```

****

Cuando retorna accept (), la estructura remote será cargada con el contenido de la estructura sockaddr_un del lado remoto, y len será cargada con su largo. El descriptor s2 se conecta al cliente, y está listo para enviar con la función send () y recibir con la función recv (), como se describe en la Network Programming Guide.

5. **Maneja la conexión y retorna hacia atrás, a la llamada a la función accept ( ):** Normalmente usted querrá comunicar al cliente (precisamente haremos retorno del eco de todas las cosas que se nos envía), cierra la conexión, luego ejecuta una nueva llamada a la función accept ().

```
    while (len = recv(s2, &buf, 100, 0), len > 0)
        send(s2, &buf, len, 0);

    /* desde aquí vuelve atrás a accept() */
```

6. **Cierra la conexión**: Usted puede cerrar la conexión llamando a close (), o por medio de la llamada shutdown ( ).

Con todo lo que se dijo, aquí está un programa fuente ,llamado echos.c, para hacer un servidor. Todo lo que hace es esperar por una conexión en un socket de Unix (llamado, en este caso, "echo_socket").

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```
#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, t, len;
    struct sockaddr_un local, remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local.sun_family = AF_UNIX;
    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);
    if (bind(s, (struct sockaddr *)&local, len) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(s, 5) == -1) {
        perror("listen");
        exit(1);
    }

    for(;;) {
        int done, n;
        printf("esperando una conexión...\n");
        t = sizeof(remote);
        if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == -1) {
            perror("accept");
            exit(1);
        }

        printf("Conectado.\n");

        done = 0;
        do {
            n = recv(s2, str, 100, 0);
            if (n <= 0) {
                if (n < 0) perror("recv");
                done = 1;
            }

            if (!done)
                if (send(s2, str, n, 0) < 0) {
                    perror("send");
                    done = 1;
                }
        } while (!done);

        close(s2);
    }

    return 0;
}
```

Como usted puede ver, todos los pasos mencionados están incluidos en este programa: llamadas a las funciones socket ( ), bind ( ), listen ( ), accept ( ) para obtener, finalmente. una red que envía con send ( ) y recibe con recv ( ).

Que hacer para ser un cliente

Ahora necesita hacer un programa para poder hablar con el servidor anterior. La diferencia del cliente, es que es mucho más fácil porque no tiene que hacer ninguna de las molestas llamadas a las funciones listen ( ), o accept ( ). Estos son los pasos que requiere:

1. Llamar a la función socket ( ) para obtener un Unix domain socket para comunicarse.
2. Preparar una estructura sockaddr_un con la dirección remota (donde el servidor está escuchando) y llamar a connect () con esta como un argumento.
3. ¡Asumiendo que no hubo errores, usted estará conectado al lado remoto ¡ ¡Use send ( ) y recv () para alegrar su corazón!

¿Cuál es el código para hablar con el echo server anterior ? No se preocupen amigos, aquí está echoc.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, t, len;
    struct sockaddr_un remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    printf("Intentando conectarse...\n");

    remote.sun_family = AF_UNIX;
    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if (connect(s, (struct sockaddr *)&remote, len) == -1) {
        perror("connect");
        exit(1);
    }

    printf("Conectado.\n");

    while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {
        if (send(s, str, strlen(str), 0) == -1) {
            perror("send");
            exit(1);
        }

        if ((t=recv(s, str, 100, 0)) > 0) {
            str[t] = '\0';
            printf("echo> %s", str);
        } else {
            if (t < 0) perror("recv");
            else printf("Servidor cerrando conexión\n");
            exit(1);
        }
    }

    close(s);
```

```
    return 0;
}
```

Por supuesto que en el código del cliente, notará que solo hay unas pocas system calls usadas para setear algunas cosas: socket ( ) y connect ( ). Dado que el cliente no va a ejecutar la llamada accept () aceptando cualquier conexión entrante, no hay necesidad de “escuchar” con la función listen ( ). Por supuesto que el cliente también usa las funciones send ( ) y recv ( ) para transferir datos.

socketpair ()—pipes bidireccionales super rápidos

¿Qué pasaría si usted quisiera un pipe (), pero quiere usar uno solo para enviar y recibir datos em ambos sentidos ? ¡Ya que los pipes son unidireccionales (con excepciones en SYSV), no puede hacerlo! Pero sin embargo hay una solución : use un Unix domain socket, ya que ellos pueden manejar datos bidireccionales.

¡ Pero esto es muy difícil ¡ ¡Poner todo ese código con listen () y connect () y todo eso sólo para pasar datos por ambas vías ¡ Pero porqué supone que no lo tiene ¡

Hay una belleza de system call conocida como socketpair ( ) que es bastante buena para devolverle un par de sockets ya conectados! No se necesita ningún trabajo extraordinario de su parte; y usted puede usar inmediatamente estos sockets descriptors para la comunicación entre procesos.

Por ejemplo, hagamos dos procesos. El primero envía un char al segundo, y el segundo cambia el carácter a mayúscula y retorna. Aquí hay un código simple para hacer justamente eso, llamado spair.c (sin chequeo de erroes para mayor claridad):

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

int main(void)
{
    int sv[2]; /* el par de socket descriptors */
    char buf; /* para intercambiar datos entre procesos */

    socketpair(AF_UNIX, SOCK_STREAM, 0, sv);

    if (!fork()) {  /* hijo */
        read(sv[1], &buf, 1);
        printf("hijo: lee '%c'\n", buf);
        buf = toupper(buf);  /* lo convierte en mayúscula */
        write(sv[1], &buf, 1);
        printf("hijo: envió '%c'\n", buf);

    } else { /* padre */
        write(sv[0], "b", 1);
        printf("padre: envió 'b'\n");
        read(sv[0], &buf, 1);
        printf("padre: lee '%c'\n", buf);
    }
    return 0;
}
```

Efectivamente, esta una manera cara de cambiar un carácter a mayúscula, pero el hecho es que tiene una comunicación simple que utiliza lo que realmente nos importa.

Una cosa más para notar es que socketpair () toma tanto al tipo domain (AF_UNIX) como al socket (SOCK_STREAM). Éstos pueden tener cualquier valor permitido para todos, dependiendo de qué rutinas quiere manejar su código en el kernel y si quiere stream sockets o datagram sockets.

Yo escogí sockets de AF_UNIX porque este es un documento de sockets Unix y son más rápidos que los AF_INET sockets.

Finalmente, si quiere conocer más acerca de por qué uso write () y read () en lugar de send () y recv (). Bien, para abreviar, por perezoso. Vea, usando éstas system calls no tengo que dar los argumentos de los flags para usar send ( ) y recv ( ) y siempre por el contrario seteo todo esto a cero. Por supuesto, los sockets descriptors son file descriptors como cualquier otro, para que respondan bien al manejo de las system calls-

Páginas HPUX del MAN

¡Si no ejecuta HPUX, verifique las siguientes páginas locales del MAN!

- accept()
- bind()
- connect()
- listen()
- socket()
- socketpair()
- send()
- recv()
- read()
- write()

Más Recursos de IPC

Libros

Aquí se da una lista de libros que describen algunos de los procedimientos que he planteado en esta guía, como también detalles específicos de Unix.

Bach, Maurice J. The Design of the UNIX Operating System. New Jersey: Prentice-Hall, 1986. ISBN 0-13-201799-7.

Stevens, Richard W. UNIX Network Programming. New Jersey: Prentice-Hall, 1990. ISBN 0-13-949876-1.

-----. Advanced Programming in the UNIX Environment. Addison-Wesley, 1992. ISBN 0-201-56317-7.

Páginas HPUX del MAN

Estas son las páginas del manual HPUX de su sistema local. Si ejecuta otro tipo de Unix, por favor mire sus propias páginas del MAN, ya que estas funciones podrían no funcionar en su sistema.

- accept()
- bind()
- connect()
- dup()
- exec()
- exit()
- fcntl()
- fileno()
- fork()

- ftok()
- getpagesize()
- ipcrm
- ipcs
- kill
- kill()
- listen()
- lockf()
- lseek()
- mknod
- mknod()
- mmap()
- msgctl()
- msgget()
- msgsnd()
- munmap()
- open()
- pipe()
- ps
- raise()
- read()
- recv()
- semctl()
- semget()
- semop()
- send()
- shmat()
- shmctl()
- shmdt()
- shmget()
- sigaction()
- signal()
- signals
- sigpending()
- sigprocmask()
- sigsetops
- sigsuspend()
- socket()
- socketpair()
- stat()
- wait()
- waitpid()
- write()