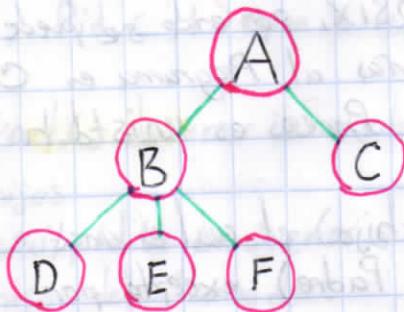


Comunicación entre Procesos: Un programa ejecuta un Proceso, cuando este se termina, hace una llamada al Sistema para terminarse a Si mismo.

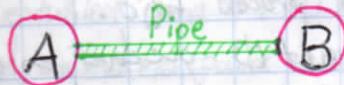
Si este Proceso puede crear más Procesos (**Procesos hijos**) y estos a su vez pueden crear más procesos, se puede desarrollar una estructura llamada **árbol de Procesos**.



Si los procesos relacionados están cooperando para realizar en conjunto alguna tarea, muy a menudo necesitan estar comunicados entre si, y sincronizar sus actividades.

Ocasionalmente, se hace necesario comunicar alguna información a un proceso en ejecución y que no está esperando recibir dicha información, en estos casos, el remitente puede solicitar al S.O. que le notifique cuando hayan pasado X segundos a fin de poder retransmitir el mensaje si todavía no ha llegado un aviso de recibido. Cuando han transcurrido esos segundos, el S.O. manda una **Señal** al proceso, esta señal hace que el proceso se ponga en modo espera, y comienza un procedimiento especial de **manejo de Señales**. Los procesos pueden cambiar de directorio de trabajo emitiendo una llamada al Sistema.

Pipes ó Conductos: es una especie de pseudoarchivo que puede servir para conectar dos procesos. Cuando un proceso A desea enviar datos al proceso B, escribe en el pipe como si fuera un archivo de entrada. Por lo tanto, la comunicación entre procesos parece más a la lectura y escritura de archivos.



Llamadas al Sistema: Ya que el mecanismo real depende mucho de la máquina y a menudo se expresa en lenguaje ensamblador, se usa una API que permite efectuar las llamadas al Sistema. (POSIX)

Como tal POSIX (**Interfaz Portátil Sist. Operativo**) es un **estándar** familia de estándares especificados por la IEEE para mantener la compatibilidad entre los Sistemas Operativos; POSIX define la API para el manejo de Procesos entre otras cosas.

Una API por lo general se relaciona con una biblioteca de Software.

La API describe y prescribe el "comportamiento esperado".

Linux implementa de forma nativa las API POSIX, por ende se puede hacer uso de ellas llamando las respectivas Librerías al Programa en C.

Una de las varias Librerías para hacer manejo de Procesos es **unistd.h**.

pid_t fork(); Crea un nuevo proceso (Proceso hijo) el cual es una copia exacta del proceso que lo invoca (Proceso Padre) excepto por algunos Puntos:

- Los PID son diferentes
- Los PPID también son diferentes
- el hijo tiene su propia copia de los flujos de directorio abiertos del Padre.
- Si se admiten los Semáforos, todos los que estén abiertos en el proceso principal lo estarán también en el hijo.
- Se anulan las posibles alarmas pendientes.
- el hijo tiene su propia copia de descriptores.
- Los bloques de archivo no son heredados por el Proceso hijo.
- el hijo no hereda ninguna operación de entrada/salida asincrónica.
- Las variables NO se comparten.

La función puede retornar estos valores:

0 → se está en el proceso hijo

>0 → se está en el proceso Padre (retorna el PID del hijo)

<0 ó -1 → error al crear el proceso hijo

pid_t wait(int *); } en ambos casos ya sea que el proceso
pid_t waitpid(pid, int*, options); } hijo termine o esté en espera, anota
 al proceso que invoca estas funciones el estado del proceso hijo.

El proceso que ejecuta la llamada al sistema **wait()** quedará en espera (quedá bloqueado) hasta que se le notifique que cualquiera de sus hijos ha acabado y continuará ejecutando lo que esté después del **wait**.

`waitpid()` funciona de la misma manera, pero este caso espera a un hijo en específico, también depende de los valores que se le asignen en el PID, es decir:

Si `Pid == -1` > espera a cualquier proceso, es decir se comporta como `wait()`

Si `Pid > 0` > espera al proceso con ese valor de PID

Si `Pid == 0` > espera a cualquier proceso que su ID de grupo sea el mismo.

Si `Pid < -1` > espera a cualquier proceso que su ID de grupo sea igual a $| -\text{Pid} |$.

`Wait()` da error si y solo si el proceso llamador (el que lo invoca) no tiene hijos

`waitpid()` dará error si el proceso o grupo especificado no existe o no es hijo del llamador.

`Void exit(int);` } el primer caso es la función de ANSI C, y el segundo `void _exit(int);` } de la librería POSIX.1. en ambos casos pone fin a la ejecución de un proceso.

Siempre es conveniente devolver un valor en `exit()`. Por convención se devuelve 0 si no hay error, y un código distinto de cero si lo hay.

Cuenta como terminación anormal si:

- Se llama la función `abort`, genera una señal **SIGABORT**.
- Cuando el proceso recibe determinadas señales ya sea por el mismo proceso, por otros procesos o por el kernel.

Cuando un programa en C ejecuta `exit()`, pasa esto:

- Se confirman los cambios de los buffers de los ficheros.
- Se cierran todos los descriptores de ficheros
- Se eliminan ficheros temporales
- Se notifica al proceso padre mediante la señal **SIGCHLD**.
- Se establece como PPID de todos los procesos huertos el valor de 1 (proceso **INIT**)
- Si el proceso era padre de un Grupo, se mandan las señales **SIGHUP** y **SIGCONT** a los huertos.
- Si era un Proceso controlador (`intro`), se envía la señal **SIGHUP** al proceso de primer plano (**foreground**) y se prohíbe el acceso al terminal controlador.

`int exec(...);` existen 6 funciones diferentes para `exec()`, pero todas funcionan de forma parecida;

Cuando un programa* hace una llamada `exec()`, su código se reemplaza por el correspondiente al del nuevo proceso que se quiere ejecutar.

De forma concreta, `exec()` reemplaza el segmento de código, datos, heap y pila del proceso que hace la llamada por el correspondiente al programa que se quiere.

NO Cambia: el PID, el ID real de usuario, ID de grupo o GID, identificadores de sesión, el ID de grupo de Procesos, directorio actual de trabajo, directorio root, máscara de creación de ficheros, señales pendientes.

Las 6 formas de llamar `exec()` son:

`exec1 (Const char *path, char *const arg[1], arg1, ..., argN, NULL);`
`exec1p (Const char *file, " ", , , ..., , NULL);`

`execv (Const char *path, Char *const argv[]);`
`execvp (Const char *file,);`

`execle (igual que exec1..., char *const envp[]);`
`execve (igual que execv..., char *const envp[]);`

No retorna nada si se ejecuta de forma normal, retorna -1 si genera error.

* 4 funciones aceptan un pathname y 2 un nombre de archivo.

→ Si el archivo NO es ejecutable, se asume que es un programa de Shell.

* Los "L" tienen paso de argumentos por lista terminada en NULL

* Los "V" tienen paso de argumentos por vector.

* Los "e" reciben un array de punteros a char Strings que terminan en NULL, estas strings constituyen el entorno para la nueva imagen.

Solo `execve` es llamada al sistema, las demás son funciones que la invocan.

Cuando creamos varios procesos, debemos identificar si nuestro diseño serán procesos independientes, o procesos cooperativos.

En el caso de los procesos independientes no es necesario un mecanismo de comunicación, en cambio, los procesos **cooperativos**, sí pueden afectar o ser afectados por los demás procesos que se ejecutan en el sistema.

Por este motivo, es casi indispensable crear un entorno que permita un correcto funcionamiento y correcta cooperación entre ellos.

Algunos motivos para fomentar la cooperación entre procesos:

Compartir Información >> Varios elementos están interesados en el mismo elemento de información (un archivo por ejemplo).

Aceleración de Los Cálculos >> Para que se acelere, debemos dividirla en subtareas, cada una se ejecutará en paralelo con las otras.

Modularidad >> Se puede querer construir un sistema siguiendo un patrón modular, dividiendo las funciones del sistema en procesos individuales.

Comodidad >> Simplemente para facilitar el procedimiento de varios trabajos.

La ejecución concurrente que precisa de la cooperación entre sus procesos requiere mecanismos de comunicación de procesos entre ellos y también sincronización entre sus acciones...

I.P.C. (Inter-Process Communication)

Hay varias formas de implementar un I.P.C.:

- > Memoria Compartida
- > Recursos Compartidos
- > Tuberías (Pipes) con nombre y Sin nombre
- > Pasaje de Mensajes
- > hilos (L.W.P., Lightweight Process)
- > Sockets.

En este momento solo se analizan los Pipes y la memoria compartida. Los Sockets son necesarios para comunicar procesos entre diferentes computadores.

Modos de IPC: Existen varios modos de comunicación entre procesos (Realmente hay 2 formas fundamentales):

Sincrónico:

- * El emisor no termina de enviar hasta que el receptor lo recibe.
- * Si el mensaje se manda sin error, significa que se recibió sin error.
- * En general involucra el bloqueo tanto del emisor como del receptor.

Asincrónico:

- * El emisor envía algo que el receptor va a recibir en algún otro momento.
- * Requiere algún mecanismo adicional para saber si el mensaje llegó.
- * Libera al emisor para realizar otras tareas, no suele haber bloqueo, aunque se puede presentar en algunos casos.

Tuberías sin nombre conocidas simplemente como pipes es un mecanismo IPC unicidireccional, se crean con la llamada a la función `pipe()` que se encuentra en la API POSIX, específicamente en la librería `unistd.h`. Su declaración es de la forma:

```
int pipe(int fileDescriptors[2]);
```

Si la llamada tiene éxito devolverá `0` y creará un pipe sin nombre; Si falla o hay un error devolverá un `-1`, y en error estará el código de error producido.

La manera de manejar el pipe es a través del array `fileDescriptors` (`fd` para comodidad). Los elementos de `fd` se comportan como dos descriptores de fichero y se usan para leer y escribir en el pipe.

Un **Descriptor de ficheros** es un número entero positivo usado por un proceso para identificar un fichero abierto. Esta traducción se hace mediante una tabla de descriptores, ubicada en la zona de datos del proceso:

- `0`: entrada normal (`stdin`)
- `1`: salida normal (`stdout`)
- `2`: salida de error (`stderr`)

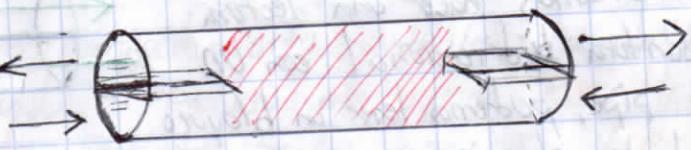
Tarea
Hernández Sánchez Juan René

2016630187

un Pipe sin nombre puede almacenar hasta 4KB en BSD o 40KB en System V.

Sólo se puede usar mediante procesos emparentados (padre e hijo o entre hijos).

~~fd[0]~~ se abre para lectura.
~~fd[1]~~ se abre para escritura.



~~fd[0]~~ se comporta como un fichero de solo lectura, y ~~fd[1]~~ se comporta como un fichero de solo escritura.

Como el núcleo del O.S. trata al pipe igual que a un fichero del sistema, le asigna un nodo-i, un par de descriptores de fichero (~~fd[0]~~ y ~~fd[1]~~) y reserva las correspondientes entradas en la tabla de ficheros del sistema.

Como los fd se heredan de padres a hijos, el Padre debe crear el pipe y así el hijo hereda esa conexión, y de esa forma ambos procesos se quedan comunicados.

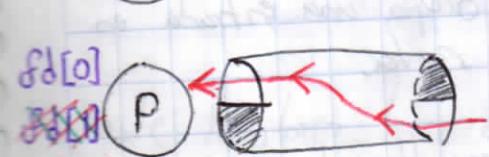
Con las llamadas **read** y **write** podremos leer y escribir en el pipe.

Cuando la tubería está llena se bloquean las llamadas write hasta que no se lean los suficientes datos para continuar.

Como tal existirán 4 fd, 2 para cada proceso (lectura y escritura respectivamente). Así que se deberá definir en qué dirección correá el pipe. esto se hace cerrando los fd que no se van a usar.

Si por ejemplo queremos pasar información del Padre al hijo:

~~fd[0]~~ se crea en el padre el fd de lectura
~~fd[1]~~ en el hijo el de escritura.



~~fd[0]~~: queremos pasar del hijo al Padre se cierra el fd de lectura en el hijo y el de escritura en el Padre.

Tareas

Hernández Sánchez Juan René

2016630187

es una buena práctica cerrar todas las conexiones del fd que no usamos ó al terminar de usar el pipe.

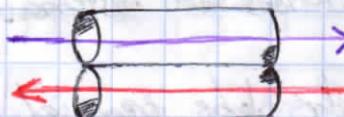
Si intentamos hacer una lectura y escritura bidireccional con 1N



Solo pipe, podemos tener un bloqueo

en donde el mismo proceso escriba y lea el dato y es segundo proceso se quedará bloqueado ya que no tendrá que leer...

Si queremos tener un IPC bidireccional necesitaremos 2 pipes, ó usar otro método de IPC.



Otra forma sería utilizar métodos de sincronización y ayudarnos de señales ó semáforos.

Tuberías en los interpretadores de órdenes»

En casi todos los interpretadores de órdenes (Shell, ó terminales)

Se pueden usar pipes para redirigir la salida de un proceso a la entrada de otro proceso. Con esto aprovechamos programas estándar.

`ls | sort -r`

Tuberías con Nombre (named pipes ó FIFOs)

Los pipes SIN nombre solo funcionan con procesos emparentados, si queremos comunicar dos ó más procesos entre sí y que no tengan ninguna relación de parentesco se usan FIFOs.

Un FIFO ó Tubería con nombre es un fichero con una semántica idéntica a la de un pipe sin nombre, pero ocupa una entrada en el directorio y se accede a él a través de una ruta.

Un proceso puede abrir una FIFO mediante una llamada a `open(...)`

de la misma forma que se abre un fichero ordinario. Así para comunicarse los procesos uno debe abrir para escribir y el otro debe abrir para leer.

La llamada **open** tiene un comportamiento particular cuando se trata de una FIFO. Cuando un proceso abre para **Escribir** en ella, se pone a **dormir hasta** que otro proceso abra para **leer** del FIFO, y esto ocurre de forma inversa, si un proceso abre el pipe para **Leer**, se dormirá hasta que un proceso abra para **escribir** en ella.

Para los FIFO el nucleo emplea solo los bloques directos de direcciones de su nodo-*i*, es decir, la cantidad total de bytes que se pueden enviar está limitada (hablando de una sola operación de escritura).

El nucleo gestiona mediante dos punteros, y los datos son manejados como una cola circular y se administra como su nombre lo dice, FIFO (First in, first out), al igual que los pipes sin nombre.

Para poder hacer uso de un named Pipe debemos crearlo primero, dos de las formas más generales de crear un pipe con nombre es mediante linea de comandos o en el mismo código del programa:

\$ mkfifo nombre-pipe

{ **#include <sys/stat.h>**

...

mkfifo(char* path, mode);

Una vez creado el named Pipe se puede hacer uso de él mediante el comando:

\$ cat nombre-pipe (Para leer)

{ **#include <fcntl.h>**

...

open (char* name, mode);

existe también la orden **mknod(...)** pero requiere privilegios de super usuario.

Tareas

Hernández Sánchez Juan René

2016630187

modo: es la máscara de modo que codifica los permisos habituales de **lectura - escritura - ejecución** del pipe.

Bit **XXXX**

S_IRUSR} Lee los bit de permiso para el propietario del archivo.
S_IREAD} Solo leido por el propietario. Bit 0400

S_IWUSR^{SR}} Escribe los bit de permiso para el propietario del archivo,
S_IWRITE} Solo escritura por el propietario. Bit 0200

S_IXUSR} Ejecución (para archivos normales) o búsqueda (para directorios),
S_IEXEC} Ejecución (búsqueda) por el propietario. Bit 0100

S_IRWXU} es equivalente a "**S_IRUSR | S_IWUSR | S_IXUSR**".
Bit 00700 Leer, escribir ó ejecutar (Búsqueda) por el propietario.

S_IRGRP} Permisos de lectura para un grupo ó su grupo propietario.
Lectura por un grupo. Bit 040.

S_IWGRP} Permisos de escritura para un grupo de Propietario.
Escritura por un grupo. Bit 020

S_IXGRP} Permisos de Búsqueda ó ejecución por un grupo propietario.
Ejecución (Búsqueda) por un grupo. Bit 010

S_IRWXG} es equivalente a "**S_IRGRP | S_IWGRP | S_IXGRP**".
Bit 00070 Leer, escribir ó ejecutar (Búsqueda) por un grupo.

Existen otros 4 modos para dar permisos a Otros usuarios:

S_IROTH, **S_IWOTH**, **S_IXOTH**, **S_IRWXO**. ~~0000X~~

S_ISUID} establece como bit de ejecución el ID del Usuario

S_ISGID} Establece como bit de ejecución el ID de Grupo de Usuario.

Tareas

Hernández Sánchez Juan René

2016630187

al momento de crear un named Pipe los errores más comunes pueden ser:

- EACCES** > no hay permisos de entrada ó ejecución (Busqueda) en el path.
- EXIST** > el pathname ya existe
- ENOENT** > algún componente del directorio del pathname no existe.
- ENOTDIR** > algún componente del pathname no es un directorio
- EROFS** > el pathname se refiere a un archivo de sólo lectura.

Todos estos errores de error se guardan en errno, así que es buena práctica revisar qué tipo de error se lanzó al ejecutarse.

En el momento que queremos hacer uso de esa FIFO, es necesario invocar la función **open()**, la cual también hace uso de modos de uso:

- O_RDONLY** > abrir para sólo Lectura
- O_WRONLY** > abrir para sólo escritura
- O_CREAT** > genera error si ya existe el archivo, sino, ejecuta igual que O_EXCL.
- O_EXCL** > La verificación y creación del archivo es atómica con respecto a otros subprocessos.

A partir de este momento un FIFO funcionará igual que un pipe sin nombre, haciendo llamadas a la función write y read, y cerrando los canales cuando ya no se quiera usar.

Comunicación DÚPLEX.

El verdadero poder del uso de FIFOs es el de que un proceso pueda leer simultáneamente de dos o más dispositivos.

Permite que un proceso pueda actuar como emisor ó receptor en cualquier instante sin tener que sujetarse al protocolo de uso de la palabra.

Para poder simular las comunicaciones duplex tenemos que asegur que un proceso pueda leer de forma simultánea datos procedentes de más de una fuente.

Hay 3 técnicas para lograr esto:

- Interrogación Periódica - **polling** -
- Lectura condicionada por eventos.
- Multiplexación mediante select.

Interrogación Periódica (Polling)

Consiste en hacer un recorrido periódico interrogando sobre el estado de las distintas fuentes. En el momento que se encuentra una actividad, se procede a leerlos y pasamos a interrogar sobre el estado de la siguiente.

Implica disponer de funciones que informen sobre el estado de la fuente, a veces esa función significa interesar con la propia función de lectura.

Para impedir que la función se quede bloqueada en el caso de que no haya datos para leer, se tiene que reprogramar el acceso al fichero para permitir una lectura **NO** bloqueante. (puede ser abriendo el fichero o dispositivo con el indicador **O_NDELAY** activo o activandolo con una llamada a **fcntl**).

La frecuencia del interrogatorio debe ser lo suficientemente lenta para no saturar la CPU y lo suficientemente rápida para que no se noten demoras en la comunicación.

Uno de sus inconvenientes es el tiempo que se pierde interrogando el estado de las diferentes fuentes.

Si el régimen de generación de datos y el periodo de muestreo son muy altos, puede darse pérdida de información.

Lectura condicionada por eventos.

Habrá varios procesos encargados de atender distintas fuentes de datos. En concreto habrá un proceso por cada fuente.

El proceso principal estará parado la mayor parte del tiempo y, solo cuando alguno de sus procesos servidores le comunique que se ha producido un evento, tomará el control y leerá del dispositivo correspondiente.

Una de las muchas formas de comunicar eventos que tienen los procesos servidores es mediante el envío de señales al proceso principal.

El uso de distintas señales sirve para identificar las distintas fuentes, pero no resulta funcional cuando el número de fuentes es muy elevado ya que solo se nos permite usar 2 señales para el desarrollo.

Para evitar esto se puede recurrir a poner una cabecera en el conjunto de datos que los procesos servidores le manden al proceso principal. Esta cabecera puede ser la procedencia del mensaje y la longitud del mismo. Y con esto tenemos implementado un IPC.

Multiplexación mediante Select.

Esta llamada permite interrogar al sistema sobre el estado de varios dispositivos, y devolver cuáles están listos para leer datos de ellos y cuáles lo están para escribir en ellos. *máscaras de Bit*.

#define <time.h>

```
int select(int nfds, int readfds, int writefds, int exceptfds, struct timeval *timeout);
```

Examina los descriptores de ficheros especificados en las máscaras de bit, y se examinan los bits comprendidos entre 0 y nfds-1. El significado de cada máscara es: (por cada Bit activo, select interroga el estado del fichero)

readfds > los fd de lectura por los que preguntamos.

writefds > los fd de escritura por los que preguntamos

exceptfds > los fd con alguna condición especial por los que preguntamos.

Estos tres argumentos son pasados por referencia, no por valor.

Significa que `select()` los va a modificar de acuerdo con el estado de los ficheros que interroga.

El parámetro `timeout` es un puntero `No nulo` que indica el tiempo máximo que se esperará desde que `select` se ejecuta hasta que devuelva el control.

Puede devolver el control bien porque alguno de los ficheros cumple con los requisitos pedidos o bien porque el tiempo de espera se agota, en ese caso `select` devolverá cero y en las máscaras aparecerán todos los bits a cero.

Si `timeout` es un puntero a `NULL`, la llamada esperará a que se dé algún cambio de estado en alguno de los ficheros interrogados.

9 Memoria Compartida: (Shared Memory)

La forma más rápida de comunicar dos procesos es hacer que comparten una zona de memoria. Para enviar datos de un proceso a otro, solo hay que escribir en memoria y automáticamente esos datos estarán disponibles para cualquiera otro proceso.

Unix System V brinda la posibilidad de crear zonas de memoria con el poder de ser direccionadas por varios procesos simultáneamente. El espacio de direcciones de esta memoria es virtual y el núcleo es el encargado de gestionar y traducir estas direcciones.

Las llamadas para manipular la memoria compartida son:

Shmget > crea una zona de memoria compartida ó habilita el acceso a una.

Shmctl > para acceder y modificar la información administrativa y de control que el núcleo le asocia a cada zona de memoria compartida.

Shmat > para unir una zona de memoria compartida a un proceso.

Shmdt > para separar una zona previamente unida.

Stok > convierte un pathname y un identificador de proyecto a una Key para System V IPC.

Petición de Shared Memory (Shmget)

Con shmget obtendremos un identificador con la que podemos realizar futuras llamadas al sistema para controlar la memoria compartida.

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

key) es una variable o constante de tipo **key_t** que se usa para acceder a los mecanismos de IPC previamente reservados ó para reservar otros nuevos. Normalmente los mecanismos utilizados en el mismo proyecto comparten la misma llave. Para generar una llave compatible usar la función **ftok()** (más adelante la describo).

Size) es el tamaño en bytes de la zona de memoria que queremos crear.

Shmflg) es una máscara de bits que describe el comportamiento:

IPC_CREAT - Crea un nuevo segmento, si no se usa, la función buscará el segmento asociado con la llave y comprobará si el usuario tiene permisos.

IPC_EXCL - Se utiliza junto con la anterior flag para garantizar que esta llamada crea el segmento. Si el segmento ya existe, falla.

SHM_HUGETLB - Asigna el segmento usando "Huge Pages" (Páginas grandes).

SHM_HUGE_2MB - Se usa junto con el flag anterior para seleccionar el tamaño de página a 2MB en sistemas que soportan múltiples hugetlb de varios tamaños

SHM_HUGE_1GB - Lo mismo que el anterior pero de 1GB.

SHM_NORESERVE - No reserva espacio de intercambio (swap) para el segmento.

Cuando se reserva espacio de intercambio, se garantiza que es modificable.

Cuando no está reservado se puede obtener el escritor **SIGSEGV**.

Completa el mismo propósito que el flag **MAP_NORESERVE** de **mmap()**.

Tareas

Hernández Sánchez dan René

2016630187

Además de los indicadores anteriores, los 9 bits menos significativos de `shmflg` especifican los permisos otorgados al propietario, grupo y otros. Funcionan exactamente igual que los bits de modo de FIFO (`open()`).

Si la llamada a `shmget` se completa con éxito, devuelve un número entero no negativo que es **el identificador asociado a la zona de memoria**. Si no, devuelve -1 y en errno se registra el tipo de error.

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

Control de una zona de memoria compartida (`shmctl`)

Con `shmctl` podemos realizar operaciones de control sobre una zona de memoria previamente creada por una llamada `shmget`.

```
#include ... // Los mismos que shmget
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

`shmid` > es un identificador válido devuelto por la llamada a `shmget`.

`cmd` > indica el tipo de operación de control a realizar:

IPC_STAT - Lee el estado de la estructura de control de la memoria y lo devuelve a través de la zona de memoria apuntada por `buf`.

IPC_SET - Inicializa algunos de los campos de la estructura de control de la memoria compartida. El valor se toma de la estructura del `buf`.

IPC_RMID - Borra del sistema la zona de memoria compartida identificada por `shmid`. Si el segmento está unido a varios procesos, no se borra hasta que todos los procesos liberen memoria.

SHM_LOCK - Bloquea en memoria el segmento identificado por `shmid`; significa que no le afectan las acciones del intercambiador (swapper). Solo los procesos cuyo **USER_ID (EUID)** sea igual al superusuario podrán realizar esta operación.

Tareas

Hernández Sánchez dan René

20/06/2018

SHM_UNLOCK - Desbloquea el segmento de memoria compartida, con lo que los mecanismos de intercambio podrán trasladarlo de la memoria principal a la secundaria y viceversa. Solo los procesos cuyo EUID sea igual al del superusuario pueden realizar esta operación.

La estructura Shmid_ds está definida como:

struct shmid_ds {

 struct ipc_perm shm_perm; → estructura de permisos

 int sh_segsz; → Tamaño del área de memoria compartida.

 int pad1; → Usado por el sistema.

 ushort shm_lpid; → PID del último proceso que hizo la última operación con este segmento de memoria.

 ushort shm_cpid; → PID del proceso creador del segmento de memoria.

 ushort shm_nattach; → Número de procesos unidos al segmento de memoria.

 short pad2; → Usado por el sistema.

 time_t shm_atime; → Fecha de la última unión al segmento de memoria.

 time_t shm_dtime; → Fecha de la última separación del segmento de memoria.

 time_t shm_ctime; → Fecha del último cambio en el segmento de memoria.

}

Operaciones con la memoria Compartida (shmat y shmdt)

Antes de usar una zona de memoria compartida se le tiene que asignar un espacio de direcciones virtuales de nuestro proceso.

A esta acción se conoce como unirse o atarse al segmento de memoria compartida. Una vez que se deja de usar un segmento de memoria, se tiene que desatar, el segmento deja de estar accesible para el proceso.

#include // las mismas que las anteriores funciones

char *shmat(int shmid, char *shmaddr, int shmflg) // atar
int shmdt(char *shmaddr); // desatar.

Shmid > identificador de una zona de memoria creada mediante una llamada previa a **shmget**.

Shmaddr > dirección virtual donde queremos que empiece la zona de memoria compartida.

Si se efectúa correctamente la llamada **shmfdt** se devolverá **un puntero a la dirección virtual**. Esta dirección puede o NO coincidir con **shmaddr**, dependiendo del Núcleo.

Normalmente vale Ø con lo cual se deja la elección de la dirección al núcleo. Caso contrario el núcleo intentará satisfacer la petición pero es poco probable.

(algunos manuales indican que **shmaddr** es de tipo const void*)
En el caso de **shmfdt** indica la dirección virtual del segmento de memoria que queremos separar del proceso.

Shmflg > es una máscara de bits que indica la forma de acceso a la memoria:

SHM_EXEC - Permite que se ejecute el contenido del segmento. El que hace el llamado debe tener los permisos de ejecución en el segmento.

SHM_RDONLY - Adjunta el segmento de memoria para solo lectura. Si no se especifica este indicador, el segmento que se adjunta tendrá el acceso de Lectura y escritura.

SHM_REMAP - Especifica que el mapeo del segmento debe reemplazar cualquier mapeo existente en el rango comenzado en **shmaddr** y continúa para el tamaño del segmento.

Produce un error **EINVAL** si ya existe una asignación en este rango de direcciones. **shmaddr** NO debe ser nulo.

Si las llamadas funcionan **shmfdt** devolverá la **Dirección** a la que está unido el segmento de memoria, **shmfdt** devolverá Ø.

Si algo falla, ambas devuelven -1.

Formación de llaves (ftok)

Hay múltiples formas de crear llaves y es necesario que todo sistema defina algún procedimiento estándar para realizar esta función.

La biblioteca de C aparta la función `ftok()` para crear llaves de una manera estándar.

```
#include <types.h>
#include <sys/IPC.h>
```

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

Devuelve una llave basada en `path` y en `id`. Esta llave será muy útil a la hora de gestionar y usar mecanismos de **IPC**.

`path` - es un puntero a la ruta de un fichero que debe existir dentro de nuestro sistema de ficheros.

`id` - es un carácter que identifica al proyecto.

La función devuelve la misma llave para rutas enlazadas a un mismo fichero, siempre que se utilice el mismo `id`.

Genera error si el `path` no es accesible o no existe el fichero y devuelve ~~1~~ (`key_t`) (-1).