Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Ingeniería en Informática

Sistemas Operativos

Trabajo Práctico N° 11

"Comunicación entre Procesos IPC"

Entrega

	Nicanor Casas
Equipo	Graciela De Luca
	Waldo Valiente
Docente	Gerardo Puyo
	Sergio Martín

GRUPO 5

	Alumnos	
Apellido	**Nombre**	**DNI**
Arévalo	Juan Pablo	28.023.828
Cofone	Fernando	33.206.786
Pérez	Aldo	29.697.303
Prado	Juan Pablo	29.752.494
Revetria	Paula	29.975.288

Acreditación:

Instancia	Fecha	Calificación
PRE-ENTREGA	/ /2011	
ENTREGA	/ /2011	
FINAL	/ /2011	

-- 2011 --

Observaciones:

CONTENIDO

1 Introducción

1.1 Propósito

El objetivo de este documento es de realizar una entrega del TP 11 - Comunicación entre Procesos, diferenciando dos partes, una en formato WICC y las respuestas del enunciado en el anexo.

1.2 Alcance

Este documento posee una parte teórica en formato WICC y además las respuestas a las preguntas realizadas en el enunciado del TP.

1.3 Abreviaciones y nomenclaturas

Abreviación	Descripción

1.4 Referencias

Referencia	Descripción
TP 11 - Comunicación entre Procesos	Enunciado y alcance del TP elegido
TP 11 – Formato WICC 2009	Tp en formato WICC 2009 según normas de la cátedra

2 Historial de Cambios

Fecha	Versión	Descripción	Autor
08/05/2011	1.0	Creación del Documento	Todos
15/05/2011	1.0	Mayor detalle de la solución propuesta	Todos
21/05/2011	1.1	Se mejoró la teoría y la solución planteada para Signal.	Todos
27/06/2001	1.2	Documento WICC y anexo	Todos
14/11/2011	1.3	Actualización de anexos	Todos
19/11/2011	1.4	Actualizar ejemplos IPC / Enviroment	Todos

3 Documento en formato WICC 2009

Ver documento de referencia TP 11 – Formato WICC 2009

4 Anexos

Cuestionario

Comparaciones entre los mecanismos de comunicación entre procesos que hoy utiliza SODIUM y los que se utilizan en LINUX y WINDOWS. Análisis de ventajas y desventajas.

Basándonos en las investigaciones realizadas sobre los mecanismos de ipc que existen actualmente en Windows y Linux, observamos que Sodium tiene implementado solamente Memoria Compartida y Señales

A continuación vamos a detallar brevemente la forma en que Sodium tiene implementado estos dos mecanismos.

En Sodium la memoria compartida está simulada por software, es decir, los procesos, antes de culminar su ejecución copian los datos, de su segmento de datos, a los segmentos de datos de los demás procesos, con los cuales comparte memoria.

Por su parte Linux no simula memoria compartida, sino que para compartir memoria es posible hacer que dos procesos (dos programas) distintos sean capaces de compartir una zona de memoria común y, de esta manera, compartir datos.

Se necesita una clave que sea común para todos los programas que quieren compartir la memoria, que identifica unívocamente a cada objeto de memoria compartida. También se ayudan de mecanismos de sincronización, como por ejemplo semáforos, para administrar el acceso a esta área de datos compartida.

Windows, en cambio, para compartir datos, permite que múltiples procesos puedan usar archivos mapeados en memoria que el sistema almacena. El primer proceso crea el objeto de asignación de archivo llamando a la función CreateFileMapping con INVALID_HANDLE_VALUE y un nombre para el objeto. Anteponiendo los nombres de asignación de archivos objeto con "Global \" permite a los procesos se comunican entre sí, incluso si se encuentran en diferentes sesiones de terminal server. Esto requiere que el proceso debe tener el privilegio SeCreateGlobalPrivilege. Cuando el proceso ya no necesita acceso al objeto de asignación de archivos, debe llamar a la función CloseHandle. Cuando todos los manejadores están cerrados, el sistema puede liberar la sección de paginación.

Con respecto al manejo de señales, en Sodium solamente están implementadas 7 de las 32 señales que deberían implementarse. Todas estas señales son manejadas por el Kernel, a través de la System Call “Kill” y no brinda un soporte para que un proceso usuario pueda configurar como debe tratar cada unas de las señales.

Por su parte Linux, además de permitir que el Kernel maneje la señal con su tratamiento por defecto, permite que el mismo, pueda cederle la ejecución al proceso que recibió la señal, para que trate él mismo la señal. Para poder realizar lo anterior, ofrece una Sistem Call “Signal” que permite que un proceso pueda configurar como tratar cada una de las señales.

Resumiendo, un proceso puede optar por ignorar la señal (hay algunas señales que no se pueden ignorar), o bien indicar que el kernel debe darle el tratamiento por defecto, o también puede tratarla el mismo con alguna rutina en particular.

Windows solo implementa señales en las bibliotecas de C-runtime de C y actualmente no poveen soporte para su uso como mecanismos de IPC.

En Linux, las señales y los pipes son dos de los mecanismos de IPC soportados, aunque Linux también soporta el llamado mecanismo System V por la versión Unix en la que apareció por primera vez. Estos mecanismos System V son: memoria compartida, cola de mensajes y semáforos.

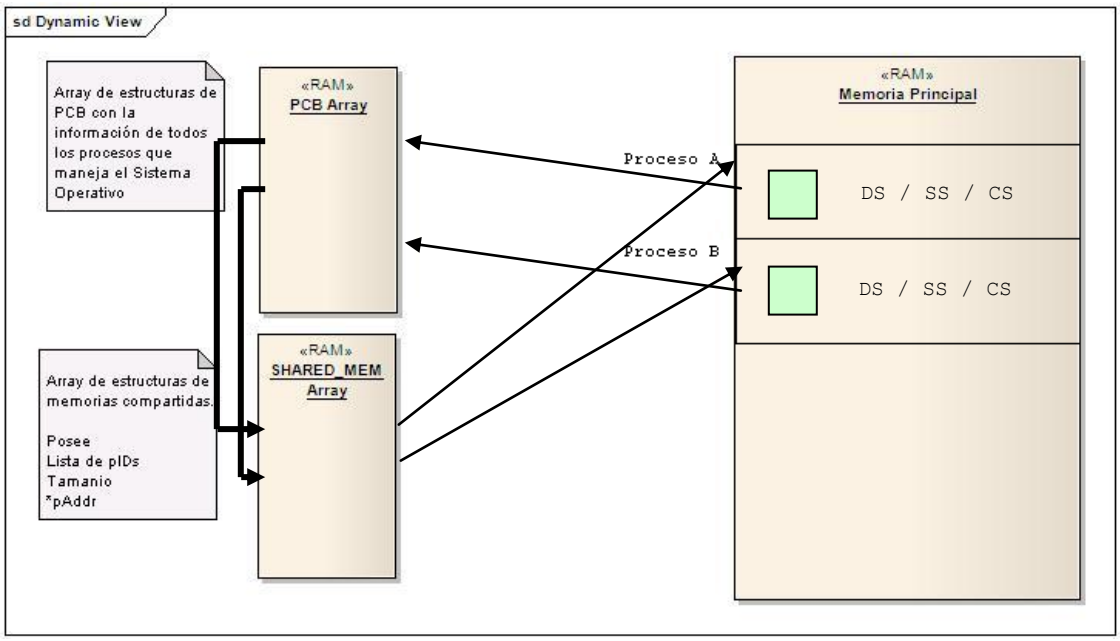
Windows posee mecanismos de IPC comunes con Linux y otros propios de su entorno: Clipboard, COM, DataCopy, DDE, File Mapping, Mailslots, Pipes, RCP, Windows Sockets.

Diseño de cómo estaba y cómo se implementara el TP en S.O.D.I.U.M. (Memoria Compartida, Cola de Mensajes y tratamiento de señales)

Memoria Compartida

En Sodium se encontraban las llamadas al sistema de distintas funciones para hacer una *simulación* del uso de memoria compartida, la misma consistía en copiar variables en todos los DS (Segmentos de datos) de los procesos que se encontraban ejecutando, luego de que finalizara el planificador.

En el siguiente esquema se visualiza la solución que tenía el Sodium.



Dato compartido

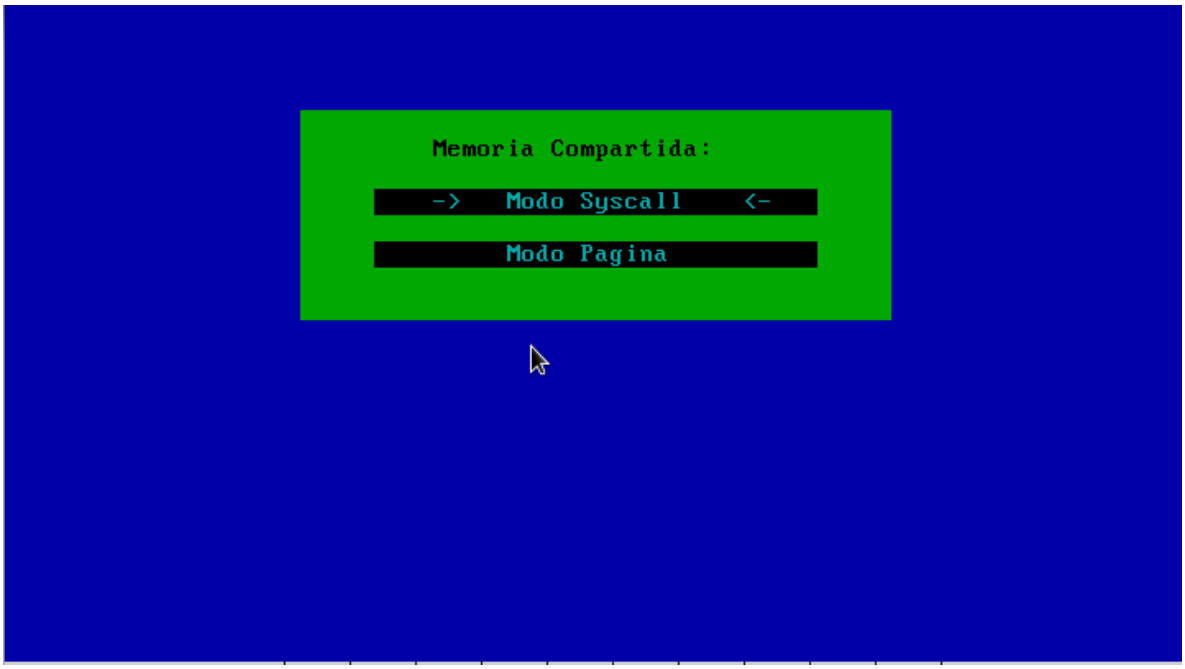
Sodium poseía estructuras de datos y funciones para la manipulación de memoria compartida en Segmentación. Algo de este código fue reutilizado para implementarlo en Paginación haciendo los ajustes correspondientes.

En cuanto a su funcionamiento, a grandes rasgos, utilizaba 2 tablas, una a nivel global denominada “memoriasCompartidas”, la misma guarda el key (Harcodeada), tamaño y todos los procesos que usan la misma (pid). La otra tabla esta dentro de los datos del PCB de cada uno de los procesos, y se denomina “memoriasAtachadas” y almacena la posición del área compartida dentro de la tabla de memorias compartidas. Esta limitada la cantidad de áreas a compartir globalmente y, además, está limitada a nivel de proceso.

Para memoria compartida se realizaron 2 formas.

- Modo Syscall.
- Modo Página.

Para poder elegir entre un modo u otro, se agrego el siguiente menú al SODIUM.



Memoria Compartida Modo SysCall.

Nuestra solución se basa en distintas llamadas al sistema que generan y comparten porciones de memoria teniendo al Kernel como interlocutor entre los procesos.

A continuación vamos a describir el funcionamiento de cada una de ellas.

Para la utilización de una zona de memoria compartida se tienen que llevar a cabo los siguientes pasos:

- Creación de un nuevo segmento de memoria compartida o acceder a uno existente.

  Llamada al sistema: **shmGet**

- El proceso debe Attacharse al segmento de memoria compartida creado

  Llamada al sistema: **shmAt**

  - Escribir dentro del área de memoria compartida generado

    Llamada al sistema: **shmWrite**

  - Leer el área de memoria compartida generado

Llamada al sistema: **shmRead**

- Desenlace del segmento de memoria compartida por parte del proceso.

  Llamada al sistema: **shmdt**

Memoria Compartida Modo Página

Pasamos a explicar como se implementó memoria compartido en modo página en SODIUM, la idea es que un proceso reserve una página, que compartida con otro proceso, permita que cuando uno escriba en la misma, el otro automáticamente pueda acceder a esos datos.
En este caso a la hora de compartir memoria, se realizan los siguientes pasos:

- Se crea el directorio de páginas.
- Se crea la tabla de páginas.
- Se crea la página de 4k. Para esta tarea se utiliza la función lFnCrearPagina4k, la cual devuelve el puntero de la página.
- Se realiza el malloc.

Claves IPC

Antes de comenzar a estudiar las llamadas al sistema para la utilización de memoria compartida es necesario explicar que son las claves IPC (InterProcess Communication). Cada objeto IPC tiene un identificador único, que se usa dentro del Kernel para identificar de forma única un objeto IPC. Una zona de memoria compartida es un objeto IPC, ya que es un objeto que sirve para la comunicación entre procesos.

Para obtener el identificador único correspondiente a un objeto IPC se debe utilizar una *clave*. Esta clave debe ser conocida por los procesos que utilizan el objeto. La clave puede ser estática, incluyendo su código en la propia aplicación. Pero en este caso hay que tener cuidado ya que la clave podría estar en uso. Otra forma es generar la clave utilizando la función ftok cuyo prototipo es el siguiente:

```
int ftok(char * cArch, int iCod);
```

Ftok devuelve una clave basada en nombre y código que puede ser utilizada para la obtención de un identificador único de objeto IPC. El argumento nombre debe ser el nombre del path de un archivo. Esta función devolverá la misma clave para todos los path que nombren el mismo archivo, cuando se llama con el mismo valor para iCod devolverá la misma Key. Se generan valores diferentes para la clave cuando es llamada con path que apunten a distintos archivos o con valores diferentes para el parámetro iCod.

Ejemplo:

```
key_t key;
/* Genero la KEY para el para el cual voy a necesitar compartir memoria */
key=ftok("/resource", 1);
```

Creación de un segmento de memoria compartida

Para la creación de un nuevo segmento de memoria compartida o para acceder a uno existente se utiliza la llamada al sistema shmget. Permite obtener el identificador del objeto de memoria compartida a partir de una clave dada. El prototipo es el siguiente:

```
int shmget( key_t ktKey, size_t stSize, int iFlags );
```

Los parámetros son los siguientes:
key: clave asociada al objeto de memoria compartida que se quiere crear o acceder.
size: tamaño del área de memoria compartida.
iFlags: flag indicando los permisos de acceso y algunas condiciones

- IPC_CREAT: crea un segmento si no existe ya en el Kernel.
- IPC_EXCL: al usarlo con IPC_CREAT, falla si el segmento ya existe.

Devuelve el identificador del segmento de memoria compartida si éxito o –1 en caso de error.

Attachado del segmento de memoria compartida

Una vez que se obtiene el identificador de la zona de memoria compartida, se debe adjuntar el proceso al espacio de memoria compartido creado. Para ello se utiliza el servicio shmat. El prototipo es el siguiente:

```
int shmat( int iShmid, void * pvShmAddr, int iFlags )
```

Los parámetros son los siguientes:

shmid: identificador del área de memoria compartida
shmaddr: dirección donde se mapea el área de memoria compartida. Si es NULL, el núcleo intenta encontrar una zona no mapeada.
iFlags: se indica el flag SHM_RDONLY si es solo para lectura.

Devuelve 1 si el proceso se pudo adjuntar o –1 en caso de error.

Escribir dentro del área de memoria compartida generada

Una vez que el proceso se encuentra adjuntado se puede escribir dentro del área generada, para ellos se utiliza el servicio shmwrite. El prototipo es el siguiente:

```
int shmwrite( int iShmid, void * pto, size_t stSize)
```

Los parámetros son los siguientes:

shmid: identificador del área de memoria compartida
pto: Dirección de memoria de donde se va a tomar los datos a poner en el área de memoria compartida.
stSize: Cantidad de bytes a escribir.

Leer del área de memoria compartida generada

Una vez que el proceso se encuentra adjuntado se puede leer del área generada, para ellos se utiliza el servicio shmread. El prototipo es el siguiente:

```
int shmread( int iShmid, void * pto, size_t stSize)
```

Los parámetros son los siguientes:

shmid: identificador del área de memoria compartida
pto: Dirección de memoria donde se va a poner los datos leídos (generalmente una estructura conocida por los procesos que forman parte del área compartida).
stSize: Cantidad de bytes a leer.

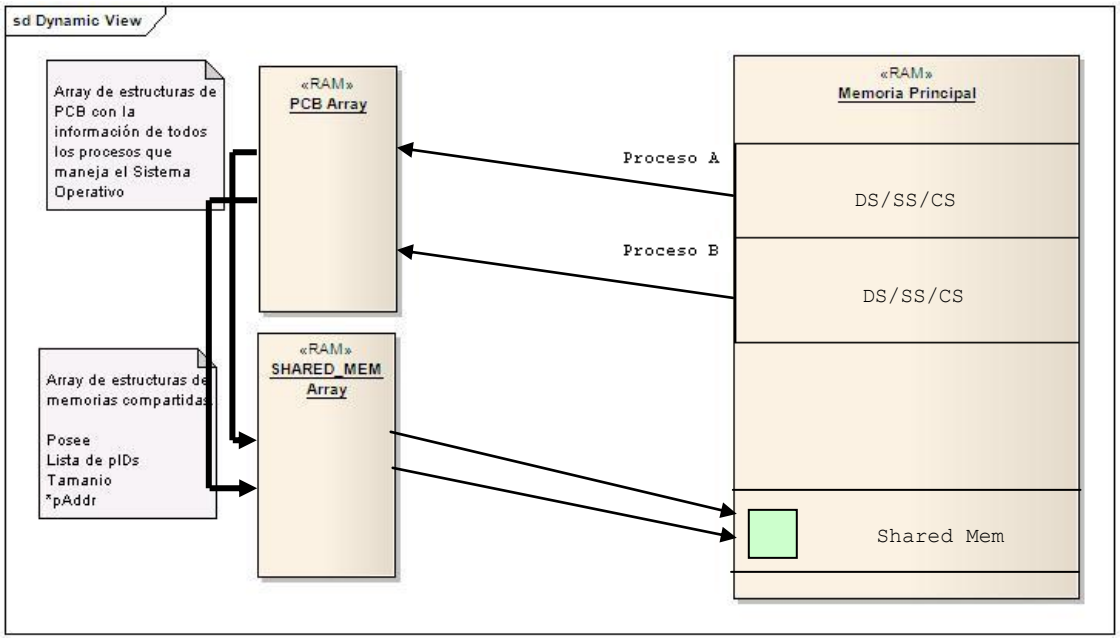
Desenlace del segmento de memoria compartida por parte del proceso

Una vez que ya no se necesita más acceder al segmento de memoria compartida, el proceso debe realizar el desenlace. El desenlace no es lo mismo que la eliminación del segmento desde el núcleo. El segmento de memoria compartida se elimina sólo en el caso de que no queden procesos enlazados. Para el desenlace se utiliza el servicio shmdt. El prototipo es el siguiente:

```
int shmdt( int iShmid )
```

Donde **shmid** es el identificador del área de memoria compartida. Devuelve 0 si éxito y –1 en caso de error.

Grafico que representa la solución realizada



Dato compartido

Cola de Mensajes

A diferencia de lo anterior sobre Cola de Mensajes no había nada realizado en el Sodium. Por lo tanto solo nos remitimos a explicar la solución que llevamos a cabo.

Para la utilización de cola de mensajes se tienen que llevar a cabo los siguientes pasos:

- Creación de una nueva área de memoria compartida o acceder a una existente.

  Llamada al sistema: **msgGet**

- Enviar un dato al área de memoria compartida generada

  Llamada al sistema: **msgSnd**

- Obtener el primer dato del área de memoria compartida generada

  Llamada al sistema: **msgRcv**

Creación de un segmento de memoria compartida para utilizar con Cola de Mensajes

Para la creación de un nuevo segmento de memoria compartida o para acceder a uno existente se utiliza la llamada al sistema msgGet. Permite obtener el identificador del objeto de memoria compartida a partir de una clave dada. El prototipo es el siguiente:

```
int msgGet( key_t ktKey, size_t stSize);
```

Los parámetros son los siguientes:
key: clave asociada al objeto de memoria compartida que se quiere crear o acceder.
size: tamaño del área de memoria compartida para la utilización de Cola de Mensajes.

Devuelve el identificador del segmento de memoria compartida si éxito o –1 en caso de error.

Enviar un dato al área de memoria compartida generada

Para el envío de datos al área compartida se utiliza el servicio msgSnd. El prototipo es el siguiente:

```
int msgSnd( int iMsgid, void * pto, size_t stSize)
```

Los parámetros son los siguientes:

msgid: identificador del área de memoria compartida
pto: Dirección de memoria de donde se va a tomar los datos a poner en el área de memoria compartida.
stSize: Cantidad de bytes a escribir.

Obtener el primer dato del área de memoria compartida generada

Una vez que el proceso obtiene a través de la Key el Id del área de memoria compartida que tiene la cola de mensajes, utiliza el servicio msgRcv para obtener el primer dato de la misma. El prototipo es el siguiente:

```
int msgRcv( int iMsgid, void * pto, size_t stSize)
```

Los parámetros son los siguientes:

msgid: identificador del área de memoria compartida
pto: Dirección de memoria donde se va a poner el dato leído.
stSize: Cantidad de bytes del dato a leer.

Diseño de cómo estaba y cómo se implementara el TP en S.O.D.I.U.M. (SIGNAL)

En Sodium se estaba utilizando la función lFnSyskill para el funcionamiento de kill y para el envío de señales, pero no estaba implementada la función Signal para el tratamiento de señales por parte del usuario. Además, solo había implementadas 7 de las 32 señales que se encuentran en el archivo Signal.h.

Nuestra solución se basó en como lo realiza LINUX y lo implementamos de la siguiente manera:

Para gestionar las señales del sistema implementaremos las siguientes modificaciones:

1) Desde el lado del kernel vamos a:

- Crear un array de estructuras dentro de la PCB llamada **sigaction** que tendrá 32 posiciones una por cada señal (como la tiene Linux).

```
stuSigaction
{
	int iPendiente; //indica si una señal esta pendiente o no
	int iSenial; //nro que representa a la señal
	// indica como debe tratarse la señal.
	//1-SIG_IGN 2-SIG_DFL Otro valor-SIG_RUT
	unsigned long ulMascara;

 };
```

- Una inicialización de la estructura mencionada anteriormente, cuando se crea la PCB del proceso.

  Esto ocurre dentro de la función *iFnCrearPCB* dentro del archivo *gdt.c* de la siguiente manera

```
//inicializo el vector de seniales
for (iJ = 0; iJ < CANTMAXSEN; iJ ++)
{
	pstuPCB[iPosicion].valorSeniales[iJ].iSenial = iJ;
	// Se inicializa siempre por default con el valor 2-SIG_DFL
	pstuPCB[iPosicion].valorSeniales[iJ].ulMascara = 2;
}
```

- Cambiamos el tratamiento de la función LFnSyskill para que el Kernel, pueda consultar el valor de la estructura para ese Pid y esa señal, y saber que acción debe tomar, si la trata con la rutina por defecto que tiene definida dentro del kernel o si debe hacer un salto a la rutina definida por el programador para el tratamiento de esa señal. En caso de que la

señal posea un tratamiento definido por el usuario, esta función lo único que hace es dejar la señal como pendiente, para que el proceso luego chequee la misma.

2) Desde el lado del proceso usuario:

- En libsodium.c definimos una systemCall cuyo prototipo será

  int signal(int iSenial, unsigned long iMask)

  Los parámetros son los siguientes:

  iSenial: Nro de señal que se desea configurar
  ulMascara: Puede admitir 3 tipos de valores bien diferenciados

  - SIG_IGN = "1" - Ignora la señal
  - SIG_DFL = "2" - Tratamiento de la señal por defecto, como lo realiza el Kernel
  - Dirección de memoria del comienzo de la rutina de tratamiento del proceso usuario.

3) Con el tratamiento de señales, estuvimos dedicados la mayor parte del tiempo, tratando de resolver que un determinado proceso, al recibir un SysKill por parte de otro de proceso, pueda atender esa señal con al algún tratamiento propio del proceso.

A continuación explicamos muy brevemente cada una de las soluciones que intentamos plasmar en el sistema operativo.

- La primer solución que intentamos implementar, sin éxito, fue tratar de que el Kernel al procesar la señal, se de cuenta que tiene que ejecutar una rutina propia del proceso y busque la dirección de la rutina y realice un call a la misma. El problema que nos encontramos fue poder apuntar al segmento de código del proceso. Ya que nos encontrábamos del lado del kernel.

- La segunda solución, fue que la función SysKill, modifique el eip en la TSS del proceso que había recibido un kill. Y que este cuanto le toque ejecutarse nuevamente, comience su ejecución desde la rutina de atención. Pero luego de consultarlo con los docentes de la cátedra, dicha solución no era viable.

Por ultimo pasamos a explicar brevemente la solución planteada.

- En libsodium.c definimos una systemCall cuyo prototipo será

  int signalR()

  Devuelve –1 en caso de error.

  La misma no recibe parámetros y su propósito es permitirle a un proceso conocer si tiene alguna señal pendiente de atención.

  Esta llamada al sistema ejecuta la función iFnSignalR del lado del kernel, la cual chequea en la estructura sigaction del proceso, si existe alguna señal pendiente, si existe devuelve la dirección de la rutina que debe ejecutar para que se realice un call a esa dirección y se atienda la señal. El Salto lo realizamos del lado de usuario.

Como crítica a la solución planteada es que el proceso antes de terminar su ejecución, debe acordarse de hacer esta llamada al sistema para chequear si existe alguna señal pendiente de atención, si no realiza esta llamada, nunca se van a atender las mismas.

Diseño cómo se implementó el TP en S.O.D.I.U.M. (PIPES)

Son flujos unidireccionales de bytes que conectan la salida estándar de un proceso con la entrada estándar de otro proceso. Ningún proceso es consciente de esta redirección y actúa como lo haría normalmente. Este puede ser usado solo entre procesos emparentados (pipes sin nombre), es por ellos que también existe el llamado pipe con nombre o FIFO, provee una interfaz simple para transmitir datos entre dos o más procesos, que residan o no en el mismo equipo. Para este último caso es necesario que se conozca el nombre en ambos extremos.

Para el correcto uso de esta funcionalidad el sistema operativo debe brindar 2 llamadas al sistema puntuales, que vamos a describir a continuación.

Creación de un canal de comunicación entre procesos emparentados

Para la creación de un canal unidireccional para procesos emparentados se debe utilizar la llamada al sistema pipe.

```
int pipe (int *iDescriptores);
```

El parámetro es el siguiente:

iDescriptores: Arreglo que recibirá los descriptores de entrada y de salida del pipe

iDescriptores[0] es el descriptor para el extremo del pipe que será utilizado para lectura y i**Descriptores[1]** es el descriptor para el extremo del pipe que será utilizado para escritura.
La operación de lectura sobre el pipe utilizando i*Descriptores*[0] accede a los datos escritos en el pipe por medio del descriptor i*Descriptores*[1] como en una cola FIFO (Primero en llegar, primero en salir).

Devuelve 0, si se ha completado correctamente; -1, en caso de error.

Creación de un canal FIFO de comunicaciones entre procesos que no necesitan estar emparentados.

En POSIX, los pipes con nombre se conocen como FIFO. Los FIFO tienen un nombre local que lo identifican dentro de una misma máquina. El nombre que se utiliza corresponde con el de un archivo. Esta característica permite que los FIFO puedan utilizarse para comunicar y sincronizar procesos de la misma máquina, sin necesidad de que lo hereden por medio de la llamada fork.

El prototipo de la llamada al sistema es:

```
int mkfifo (const char *sPath, int iPermisos)
```

Los parámetros son los siguientes:

sPath: Ruta completa y nombre del archivo FIFO.

iPermisos: Permisos de acceso para el usuario, el grupo y todos los demás, solo están disponibles los permisos de lectura (4) y escritura (2).

Devuelve 0, si se ha completado correctamente, -1 en caso de error.

Implementación en Sodium.

En sodium, no estaba implementado PIPES en su totalidad, así que pasamos a explicarles brevemente su implementación.

Se crearon las siguientes estructuras

- stuPipe: tiene datos de los pipes del sistema. Entre algunos de los datos que contiene:
  - tipo: indica si es un pipe con nombre o un pipe sin nombre
  - permisos: contiene permisos del pipe
  - nombre del descriptor de escritura
  - nombre del descriptor de lectura.

- stuPipesatachados: es una estructura que forma parte del PCB e indica los pipes en los cuales participa el proceso.

Se crearon las siguientes llamados al sistema.

- Int pipe (Descriptor[])

  El parámetro es el siguiente:

  Descriptor**:** Arreglo de dos posiciones, que recibirá los descriptores de entrada y de salida del pipe.

  Esta system call realiza el llamado a la función pipe del lado kernel y crea el canal de comunicaciones unidireccional entre procesos emparentados (pipes sin nombre), básicamente lo que realiza es abrir dos archivos uno en modo lectura y uno en modo escritura. Recordamos que la cantidad de pipes que se puede crear en el sistema esta limitada por la variable CANTMAXPIPE.
  Por otro lado adjunta el pipe al proceso en el PCB.

- int ObtenerPipeLectura(int iPipeId)

  El parámetro es el siguiente:

  PipeId**:** Es un entero que representa en Id del Pipe.

  Esta system call realiza un llamado a la funcion iFnGetPipeRead del lado del kernel y lo que hace es devolver el descriptor de lectura del pipe que recibe por parámetro.

- int ObtenerPipeEscritura(int iPipeId)

  El parámetro es el siguiente:

  PipeId**:** Es un entero que representa en Id del Pipe.

  Esta system call realiza un llamado a la función iFnGetPipeWrite del lado del kernel y lo que hace es devolver el descriptor de escritura del pipe que recibe por parámetro.

Para la implementación de Pipes con nombre (MkFifo) en Sodium, se reutilizaron las estructuras ya creadas para Pipe sin nombre.

Se crearon las siguientes llamados al sistema.

- int MkFifo (const char *path, int permisos)

  Los parámetros son los siguientes:

  Path: Path o ruta del archivo a utilizar como comunicación para el PIPE con nombre.

  Permisos: representan los permisos sobre el archivo a utilizar

  Esta system call realiza el llamado a la función *iFnMkfifo* del lado kernel que reserva en file system un archivo para poder comunicar a dos procesos no emparentados. La diferencia con los pipes sin nombre, es que en este caso los procesos que se comunican deben conocer el nombre del archivo que utilizan para comunicarse. Recordamos que la cantidad de pipes que se puede crear en el sistema esta limitada por la variable CANTMAXPIPE.

  Si bien cuando hablamos de pipes con nombre o pipes sin nombre, mencionamos la palabra archivo, en Sodium, tuvimos el inconveniente de que el mismo, aun no cuenta con las bibliotecas para el manejo de archivos.
  Por lo que para poder sortear este obstáculo y poder entregar una versión de pipes funcionando, lo que realizamos fue una simulación del manejo de archivos, que paso a detallar brevemente, básicamente se pensó al archivo como una porción de memoria:

- En primer lugar, se anexo en la estructura stuFileInfo los siguientes datos:
  - usado: indica si la entrada del file system esta ocupada o no.
  - modoApertura: puede ser de lectura, de escritura o de lectura/escritura.
  - path[MAXPATHFILE]: indica el path del archivo, esto es ficticio porque en realidad el archivo no existe.

- En segundo lugar se creo la estructura fileData, que posee los siguientes datos:
  - path[MAXPATHFILE]: indica el path del archivo, esto es ficticio porque en realidad el archivo no existe.
  - data[MAXCANTFILEDATA]: representa los datos que posee el archivo.

Se crearon las siguientes llamadas al sistema.

- **int fdOpen (char *path, char* modo)**

  Los parámetros son los siguientes:

  Path: path o ruta del archivo a abrir.

  Modo: modo de apertura del archivo (escritura/lectura)

  Esta system call realiza el llamado a la función *iFnfdOpen* del lado kernel y realiza la apertura (simulada) de un archivo.

- **int fdClose (int fd)**

  Los parámetros son los siguientes:

  Fd: descriptor del archivo a cerrar

  Esta system call realiza el llamado a la función iFnfdClose del lado kernel y realiza el cierre (simulado) de un archivo.

- **int fdRead (int fd, char *bufferSalida, int cantidad)**

  Los parámetros son los siguientes:

  fd: descriptor del archivo del que se va a realizar la lectura.

  bufferSalida: representa el buffer de salida. Donde se van a almacenar los datos leidos.

  Cantidad: cantidad de caracteres a leer.

  Esta system call realiza el llamado a la función iFnfdRead del lado kernel y realiza la lectura (simulada) de un archivo.

- **int fdWrite (int fd, char *bufferEntrada, int cantidad)**

  Los parámetros son los siguientes:

  fd: descriptor del archivo del que se va a realizar la escritura.

  bufferEntrada: representa el buffer de entrada.

  Cantidad: cantidad de caracteres a escribir en el archivo.

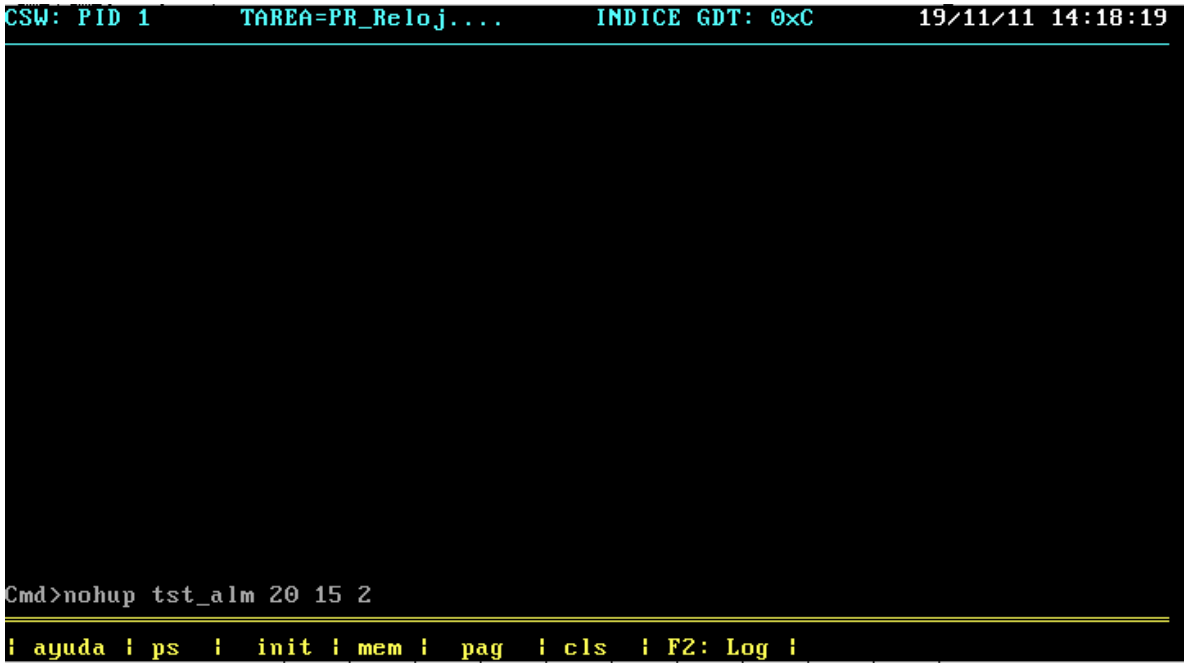
  Esta system call realiza el llamado a la función iFnfdWrite del lado kernel y realiza la escritura (simulada) de un archivo.

La nota del segundo cuatrimestre estuvo compuesta por realizar diversos test para demostrar el buen funcionamiento de las distintas funcionalidades implementadas en el primer cuatrimestre.
De todos esos ejemplos pasamos a explicar aquellos que consideramos más relevantes.

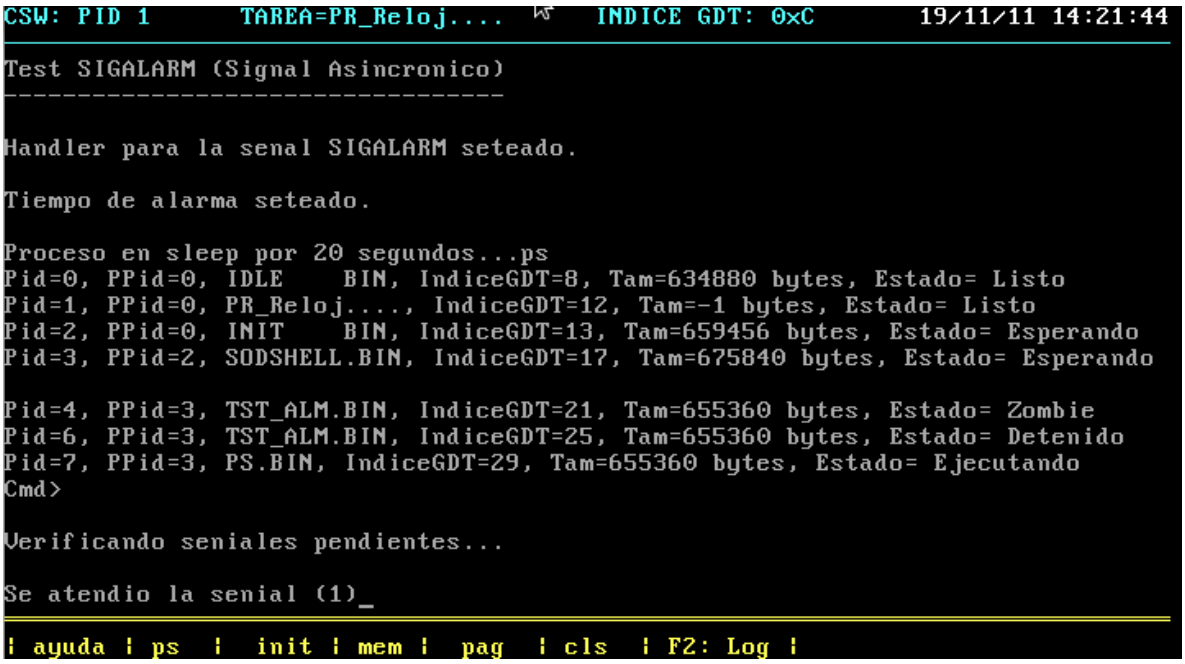
Detalle de cómo se implementó en S.O.D.I.U.M. (COMANDO NOHUP)

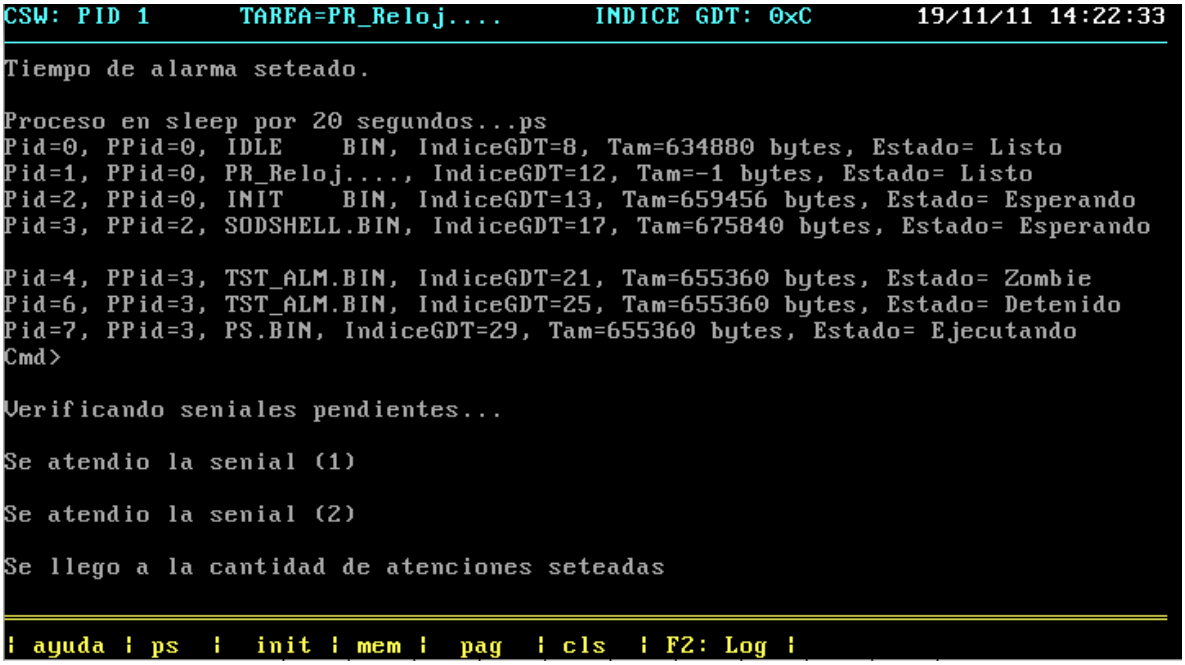
En el archivo sodshell.c se agregó un nuevo comando, el cual llama a la función *int vFnMenuNoHup().* Esta función en primer lugar va a verificar y setear el modo de ejecución, en caso de que la variable de entorno "ModoShell", se encuentre con el valor "PASIVO", se procede a cambiarlo a "ACTIVO".
Luego de realizado dicho cambio se quita del buffer el primer comando (el NOHUP), para que en el mismo, solo quede el binario que se quiere ejecutar en backgound con sus correspondientes parámetros.
A continuación, se llama a la función *iFnEjecutarBinario(stCmd, 0)* que va a ejecutar de manera POSIX el archivo ejecutable.
Una vez que se ejecutó el binario, se vuelve atrás la configuración del modo de ejecución, es decir, se pasa nuevamente a modo "ACTIVO".

Entrada



Salida





En estas 2 screenshoot se puede ver como el tst_alm lo ejecuta en segundo plano, ya que en el medio de su ejecución, el Shell nos vuelve a dar el control para que podamos ejecutar otro comando, en el ejemplo ejecutamos el PS, quedando ejecutando el tst_alm en segundo plano y terminando OK, tal como se puede ver en la segunda pantalla.

Detalle de cómo se implementó en S.O.D.I.U.M. (SIG_ALARM)

1) Desde el lado de usuario:

Se agregó una llamada al sistema *alarm* cuyo parámetro es el tiempo en el que el Sistema Operativo va a enviar al proceso la señal SIGALARM.

2) Desde el lado del kernel:

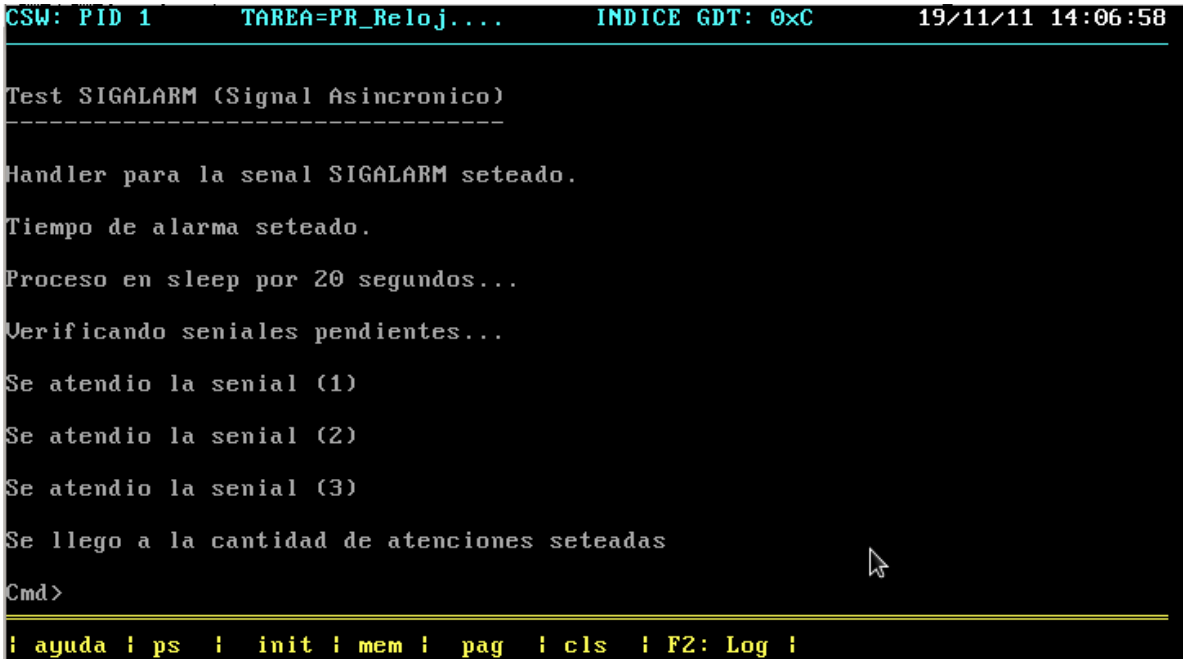
Se creó la función *int alarm(int iSegundos)* la cual va a setear el timer del proceso en la PCB. Esto lo realiza actualizando la estructura timeval (declarada en el archivo tiempo.h).

Luego de setear lo anterior, en la función *void vFnHandlerTimer (), por medio de la funcion vFnDecrementarTimer* va a decrementar el timer real.
Cuando el valor del timer sea menor o igual a cero, va a realizar un Kill al proceso y va a setear la señal SIGALARM como pendiente en la estructura stuSigAction del mismo.

El proceso, antes de morir, verifica s tiene señales pendientes mediante la llamada a la función *signalR()* la cual va a ejecutar la función seteada para la señal SIGALARM del proceso.

Entrada

Salida


CSW: PID 1 TAREA=PR_Reloj.... INDICE GDT: 0xC 19/11/11 14:06:58
Test SIGALARM (Signal Asincronico)
Handler para la senal SIGALARM seteado.
Tiempo de alarma seteado.
Proceso en sleep por 20 segundos...
Verificando seniales pendientes...
Se atendio la senial (1)
Se atendio la senial (2)
Se atendio la senial (3)
Se llego a la cantidad de atenciones seteadas
Cmd>
| ayuda | ps | init | mem | pag | cls | F2: Log |

Detalle de cómo se implementó en S.O.D.I.U.M. (Comando PIPE en el shell)

En este caso, a pesar de que solo nos pidieron hacer una investigación sobre cómo debería implementarse en SODIUM, optamos directamente por realizar la implementación del comando.

Funciona desde el Shell ejecutándolo de la siguiente manera:

cmd>binario1 | binario2 //se ejecuta el binario1 y el binario2

Para realizarlo, en el archivo sodshell.c, se verifica si se trata de una operación de PIPE. En caso de ser así se hace una llamada a la función *vFnTratamientoPIPE()***.** Esta función lo que hace es:

- Obtener los dos nombres de los binarios y validarlos que no sean vacíos. Para lograr esto se utilizó una función llamada *iFnTomaNParametro* la cual va a buscar el segundo binario dentro del buffer.
- Se setea la forma en que deben ejecutarse los binarios, para esto se verifica si la variable de entorno “ModoShell” se encuentra “ACTIVA” en cuyo caso se setea en “PASIVO”.
- Se convierten ambos nombres de los binarios a mayúsculas.
- Se procede a ejecutar cada binario, llamando a la función *iFnEjecutarBinario*
- Se vuelve hacia atrás la configuración del modo de ejecución a modo “ACTIVO”.

Entrada

Salida


CSW: PID 1 TAREA=PR_Reloj.... INDICE GDT: 0xC 19/11/11 13:57:03
Consumidor> Cadena Nro. 3
Productor> Cadena Nro. 4
Consumidor> Cadena Nro. 4
Productor> Cadena Nro. 5
Consumidor> Cadena Nro. 5
Productor> Cadena Nro. 6
Consumidor> Cadena Nro. 6
Productor> Cadena Nro. 7
Consumidor> Cadena Nro. 7
Productor> Cadena Nro. 8
Consumidor> Cadena Nro. 8
Productor> Cadena Nro. 9
Consumidor> Cadena Nro. 9
Productor> Cadena Nro. 10
Consumidor> Cadena Nro. 10
Productor> Liberando PIPE...
Productor> Finalizado
Consumidor> Finalizado
¦ ayuda ¦ ps ¦ init ¦ mem ¦ pag ¦ cls ¦ F2: Log ¦

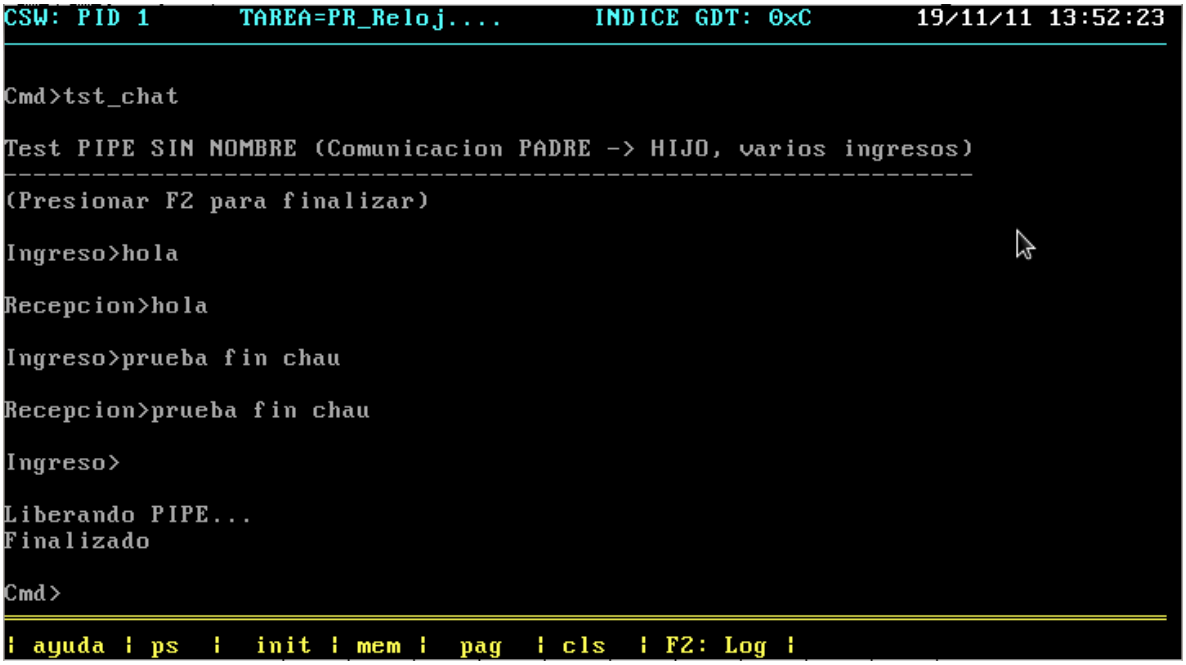
Detalle de cómo se implementó en S.O.D.I.U.M. (Chat entre procesos utilizando PIPE sin nombre)

Básicamente el test simula un chat entre un proceso padre e hijo.

El proceso padre realiza la apertura de un pipe, a través de la llamada al sistema pipe() y luego permite el ingreso de mensajes por teclado. Los mensajes los escribe por el descriptor de escritura en el PIPE.
A medida que el padre escribe mensajes el hijo va leyendo los mismos y los imprime por pantalla.

Entrada

Salida



Tests IPC´s (Casos genéricos)

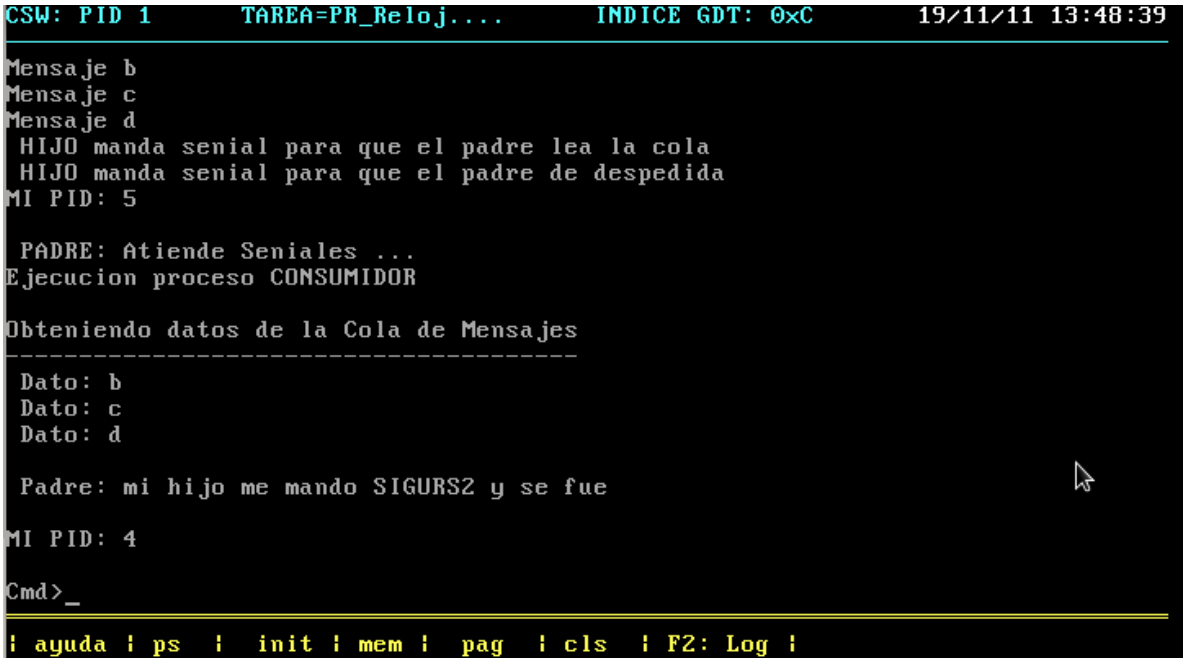
Estos test, trata de combinar el manejo de memoria compartida, a través de una cola de mensajes, y el manejo de señales. Tienen como objetivo, podar mostrar el buen funcionamientos de los distintos mecanismos ipc's realizados en el primer cuatrimestre. Pasamos a explicar uno por uno.

- **TST_IPC3**

En este tests intervienen 2 procesos emparentados. El padre setea la señal SIGUSR1 para que se atienda con una rutina propia, esta se encarga de leer una cola de mensajes vFnLeerCola. También setea la señal SIGUSR2 con otra función propia *vFnSignalService*.
El hijo, por su parte, reserva un espacio compartido de cola de mensajes y graba 5 mensajes en la misma. Inmediatamente envía un kill al padre, tanto para la señal SIGUSR1 como para SIGUSR2. Luego el padre antes de culminar su ejecución realiza un *SignalR* para verificar si posee señales pendientes y atenderlas. En este caso, las dos señales pendientes (SIGUSR1 y SIGUSR2).

Entrada

Salida



- **TST_IPC4**

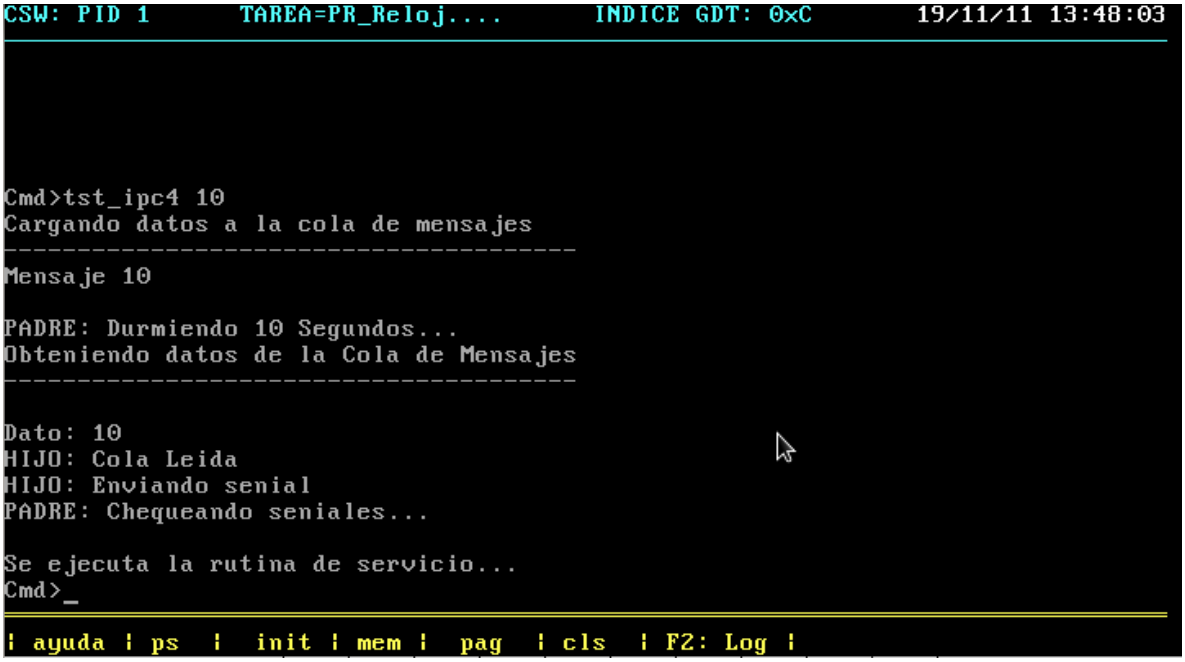
Este test recibe como parámetro: el nro de la señal que se desea setear en este test:

10 -> SIGUSR1
12 -> SIGUSR2
15 -> SIGTERM

En este tests intervienen 2 procesos emparentados. El padre setea la señal pasada por parámetro con la función *vFnSignalService*, luego le pide al kernel que reserve un espacio de memoria compartida y escribe un mensaje con el nro de señal seteado.
El proceso hijo por su parte, lee el mensaje de la cola de mensajes compartida con el padre, y le envía un kill al padre de la señal pasada por parámetro, en este test.
Por ultimo el padre se duerme por 10´ y al despertarse y antes de culminar su ejecución realiza un *SignalR* para chequear si tiene señales pendientes y atenderlas. En este caso ejecuta la función *vFnSignalService*

Entrada

Salida

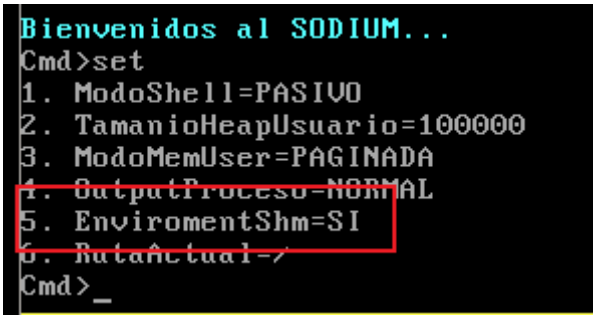


Detalle de cómo se implementó en S.O.D.I.U.M. (Memoria Compartida entre el Shell y los procesos usuarios)

Una deficiencia del S.O.D.I.U.M. era que no podía compartir Variables de Entorno del Shell con los procesos usuario, para solucionar esto se utilizó un mecanismo de comunicación entre procesos, la memoria compartida.

El desarrollo consiste en generar una memoria compartida cuando se inicializa el Shell, el array local que posee el dicho proceso se seguirá usando como hasta ahora para poder mantener la compatibilidad de lo que está hecho, la diferencia está en que cada vez que se agregue, cambie o elimine una variable de entorno, se va a sincronizar el array local del Shell con un array que se encuentra en la memoria compartida antes mencionada. De esta forma podemos hacer que todos los procesos del S.O.D.I.U.M. (incluyendo el Kernel) puedan manejar las mismas variables de entorno.

Cabe destacar que esta funcionalidad puede utilizarse si la variable de entorno *EnviromentShm* tiene el valor ***SI.***



En caso de que la variable de entorno no se encuentre o no tenga el valor ***SI*** S.O.D.I.U.M. se comportará como lo venía haciendo antes.

Para probar esta funcionalidad se crearon 2 binarios, son ejemplos muy sencillos que se describen a continuación

- tst_env

  El mismo se encarga de modificar una variable de entorno existente y agregar una nueva.

- tst_env2

  Este ejemplo solo lista las variables de entorno.

La forma de probar la funcionalidad sería la siguiente

- Realizar un set en la línea de comandos del Shell y verificar que la variable de entorno *EnviromentShm* tenga el valor ***SI,*** además visualizamos el estado de las demás variables.
- Ejecutar el binario tst_env, para que modifique y cree una variable de entorno nueva

```
void main()
{

   int i;
   i = iFnEnvSet("OutputProceso=NORMAL");
   i = iFnEnvSet("TestEnv=VALORAGREGADO");

   iFnEnvListarEnviroment();

   return;
}
```

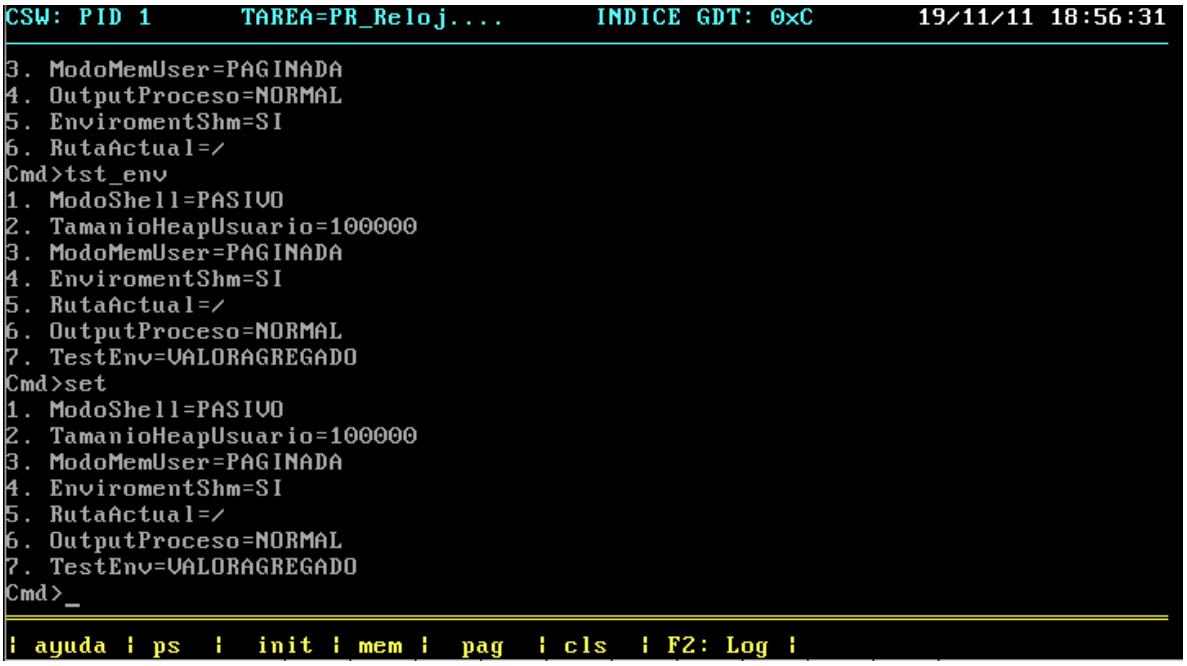
- Realizar nuevamente un set en la línea de comandos del shell y visualizar los cambios antes descriptos en las variables de entorno listadas.
- Ejecutar el binario tst_env2, el mismo listará las variables de entorno que el ve y así vamos a poder verificar que los cambios que realizó el binario tst_env, impactaron tanto en el Shell como en el proceso usuario tst_env2.

A continuación los printScreen de los pasos antes comentados

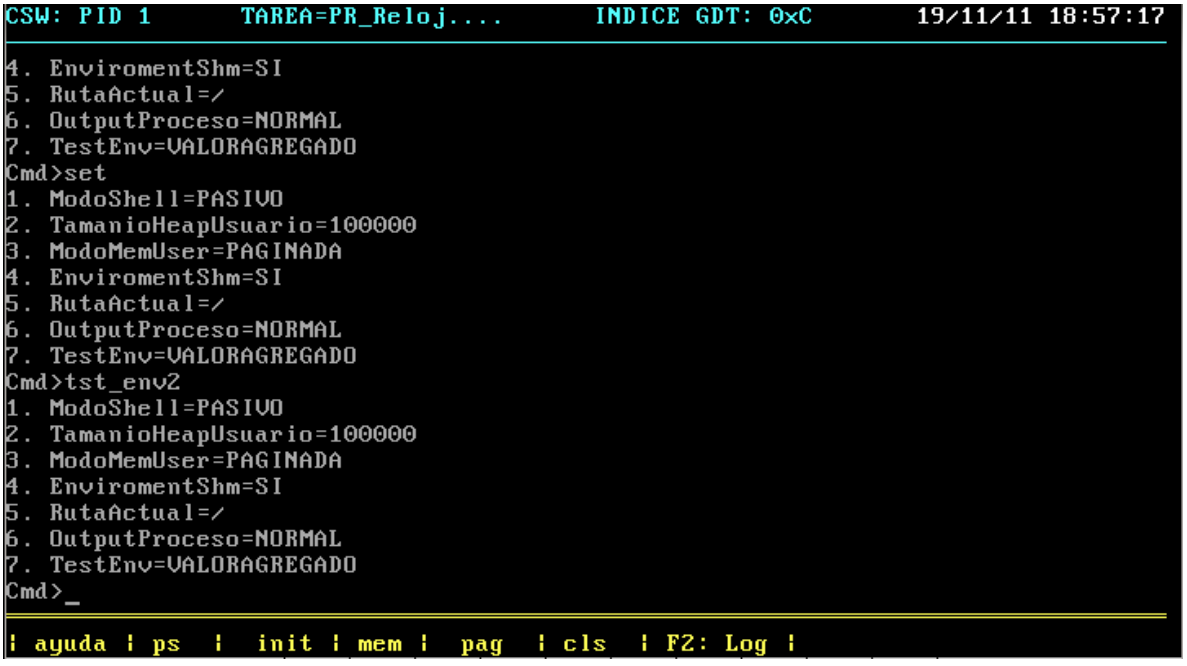
Entrada

```
CSW: PID 1      TAREA=PR_Reloj....      INDICE GDT: 0xC       19/11/11 18:39:26
Inicializar Memoria Compartida...                          [ HECHO ]
Inicializar Colas de Mensajes...                           [ HECHO ]
Inicializar Pipes...                                       [ HECHO ]
Inicializar Funciones APM...                               [ HECHO ]
Iniciando reloj del sistema ...                            [ HECHO ]
Iniciando Proceso Reloj ...                                [ HECHO ]
Iniciando Task Register ...                                [ HECHO ]
Memoria total del sistema: 31 MB                           [ HECHO ]
Iniciando el proceso Init ...                              [ HECHO ]
Se cargo la distribucion de teclado "Ingles UK"            [ HECHO ]
Habilitando Interrupciones ...                             [ HECHO ]

Bienvenidos al SODIUM...
Cmd>set
1. ModoShell=PASIVO
2. TamanioHeapUsuario=100000
3. ModoMemUser=PAGINADA
4. OutputProceso=NORMAL
5. EnviromentShm=SI
6. RutaActual=/
Cmd>
¦ ayuda ¦ ps  ¦  init ¦ mem ¦  pag  ¦ cls  ¦ F2: Log ¦
```

```
CSW: PID 1      TAREA=PR_Reloj....      INDICE GDT: 0xC       19/11/11 18:53:18
Iniciando el proceso Init ...                              [ HECHO ]
Se cargo la distribucion de teclado "Ingles UK"            [ HECHO ]
Habilitando Interrupciones ...                             [ HECHO ]

Bienvenidos al SODIUM...
Cmd>set
1. ModoShell=PASIVO
2. TamanioHeapUsuario=100000
3. ModoMemUser=PAGINADA
4. OutputProceso=NORMAL
5. EnviromentShm=SI
6. RutaActual=/
Cmd>tst_env
1. ModoShell=PASIVO
2. TamanioHeapUsuario=100000
3. ModoMemUser=PAGINADA
4. EnviromentShm=SI
5. RutaActual=/
6. OutputProceso=NORMAL
7. TestEnv=VALORAGREGADO
Cmd>_
¦ ayuda ¦ ps  ¦  init ¦ mem ¦  pag  ¦ cls  ¦ F2: Log ¦
```



Salida



Nota: Debido a la imposibilidad de poder colocar todos los test en la carpeta usr/bin, casi todos se encuentran en la carpeta usr/tst. Por lo tanto, para poder probar los distintos ejemplos, es necesario, antes de levantar el SODIUM copiarlos a la carpeta usr/bin.

Realizar un manual de operaciones que documente los pasos necesarios para usar estas bibliotecas.

Pasos para poder compartir memoria entre procesos:

a) El programador deberá incluir las siguientes bibliotecas:

```
#include <usr/lib/libsodium.h>
```

Contiene todas las funciones básicas para el manejo de memoria compartida

```
#include <usr/bin/tst_def.h>
```

Contiene definiciones relacionadas con memoria compartida.

b) El segundo paso es obtener una clave univoca para el manejo de objetos IPC. Para esto se utiliza la llamada al sistema Ftok

int ftok(char * cArch, int iCod);

Los parámetros son los siguientes:

cArch: representa el path del archivo que vamos a utilizar para la obtención de la clave
iCod: Código numérico que vamos a utilizar para la obtención de la clave.

Recordamos que para un mismo archivo y un mismo Código devuelven siempre la misma clave.

Devuelve el identificador o key si hubo éxito o –1 en caso de error.

c) Para la creación de un nuevo segmento de memoria compartida o acceder a uno se debe utilizar la Llamada al sistema: **shmGet**

int shmget(key_t ktKey, size_t stSize, int iFlags);

Los parámetros son los siguientes:
key: clave asociada al objeto de memoria compartida que se quiere crear o acceder.
size: tamaño del área de memoria compartida.
iFlags: flag indicando los permisos de acceso y algunas condiciones

- IPC_CREAT: crea un segmento si no existe ya en el Kernel.
- IPC_EXCL: al usarlo con IPC_CREAT, falla si el segmento ya existe.

Devuelve el identificador del segmento de memoria compartida si hubo éxito o –1 en caso de error.

d) Una vez creado el segmento de memoria, el proceso debe Attacharse (asociarse) al segmento de memoria compartida creado. Para esto se utiliza la llamada al sistema: **shmAt**

int shmat(int iShmid, void * pvShmAddr, int iFlags)

Los parámetros son los siguientes:

shmid: identificador del área de memoria compartida
shmaddr: dirección donde se mapea el área de memoria compartida. Si es NULL, el núcleo intenta encontrar una zona no mapeada.
iFlags: se indica el flag SHM_RDONLY si es solo para lectura.

Devuelve 1 si el proceso se pudo adjuntar o –1 en caso de error.

e) Para escribir datos dentro del área de memoria compartida generado se utiliza la llamada al sistema: **shmWrite**

int shmwrite(int iShmid, void * pto, size_t stSize)

Los parámetros son los siguientes:

shmid: identificador del área de memoria compartida
pto: Dirección de memoria de donde se va a tomar los datos a poner en el área de memoria compartida.
stSize: Cantidad de bytes a escribir.
Devuelve 0 si el proceso se pudo adjuntar o –1 en caso de error.

f) Para leer el área de memoria compartida generado se utiliza la llamada al sistema: **shmRead**

int shmread(int iShmid, void * pto, size_t stSize)

Los parámetros son los siguientes:

shmid: identificador del área de memoria compartida
pto: Dirección de memoria donde se va a poner los datos leídos (generalmente una estructura conocida por los procesos que forman parte del área compartida).
stSize: Cantidad de bytes a leer.

Devuelve 0 si el proceso se pudo adjuntar o –1 en caso de error.

g) Un proceso puede solicitar el desenlace del segmento de memoria compartida (desvinculación). En caso de querer volver a utilizar ese segmento de memoria compartida va a necesitar attacharse nuevamente. Para la desvinculación se utiliza la llamada al sistema: **shmdt**

int shmdt(int iShmid)

Donde **shmid** es el identificador del área de memoria compartida.

Devuelve 0 si éxito y –1 en caso de error.

Pasos para poder compartir memoria entre procesos (Cola de Mensajes):

a) El programador deberá incluir las siguientes bibliotecas:

```
#include <usr/lib/libsodium.h>
```

Contiene todas las funciones básicas para el manejo de cola de mensajes.

b) Lo primero que se debe hacer es obtener una clave univoca para el manejo de objetos IPC. Para esto se utiliza la llamada al sistema **Ftok**

int ftok(char * cArch, int iCod);

Los parámetros son los siguientes:
cArch: representa el path del archivo que vamos a utilizar para la obtención de la clave

iCod: Código numérico que vamos a utilizar para la obtención de la clave.

Recordamos que mismo archivo y mismo Código devuelven siempre la misma clave.

Devuelve el identificador la key si hubo éxito o –1 en caso de error.

c) Para la creación de una nueva área de memoria compartida o acceder a una existente y obtener el identificador del objeto de memoria compartida a partir de una clave dada existe utilizamos la llamada al sistema: **msgGet**

int msgGet(key_t ktKey, size_t stSize);

Los parámetros son los siguientes:
key: clave asociada al objeto de memoria compartida que se quiere crear o acceder.
size: tamaño del área de memoria compartida para la utilización de Cola de Mensajes.

Devuelve el identificador del segmento de memoria compartida si éxito o –1 en caso de error.

d) Enviar un dato al área de memoria compartida generada. Se podría decir que esto lo que hace es encolar un mensaje. Se utiliza la llamada al sistema: **msgSnd**

int msgSnd(int iMsgid, void * pto, size_t stSize)

Los parámetros son los siguientes:

msgid: identificador del área de memoria compartida
pto: Dirección de memoria de donde se va a tomar los datos a poner en el área de memoria compartida.
stSize: Cantidad de bytes a escribir.

Devuelve 0 si hubo éxito y –1 en caso de error.

e) Obtener el primer dato del área de memoria compartida generada. Se podría decir que lo que hace es desencolar un dato de la cola. Se utiliza la llamada al sistema: **msgRcv**

int msgRcv(int iMsgid, void * pto, size_t stSize)

Los parámetros son los siguientes:

msgid: identificador del área de memoria compartida
pto: Dirección de memoria donde se va a poner el dato leído.
stSize: Cantidad de bytes del dato a leer.

Devuelve 0 si hubo éxito y –1 en caso de error.

Pasos para poder realizar el manejo de señales entre procesos:

a) El programador deberá incluir las siguientes bibliotecas:

```
#include <usr/lib/libsodium.h>
```

Contiene todas las funciones básicas para el manejo de señales.

b) Un proceso puede configurar que acción debe realizar el Kernel, cuando se reciba alguna señal, en donde el propio proceso sea la victima. Entre las acciones se puede optar por:

- El proceso puede indicar que el Kernel ignore la señal, siempre y cuando no se trate de las señales SIGSTOP y SIGKILL, las cuales no pueden ser ignoradas.
- El proceso puede indicar, que cuando reciba la señal, se realicen las acciones definidas por el Kernel. Dicho de otra manera, que se le de a la señal el tratamiento por defecto.
- O bien, el proceso puede configurar que él mismo va a tratar la señal, con una rutina definida a nivel de usuario. Para esto debe indicarle al Kernel la dirección de donde se encuentra la rutina de atención de la señal (esta se encuentra dentro del segmento de código del proceso).

Para esto se utiliza la llamada al sistema **Signal**

int signal(int iSenial, unsigned long iMask)

Los parámetros son los siguientes:

iSenial: Nro de señal que se desea configurar
iMask: Puede admitir 3 tipos de valores bien diferenciados

- SIG_IGN = “1” - Ignora la señal
- SIG_DFL = “2” - Tratamiento de la señal por defecto, como lo realiza el Kernel
- Dirección de memoria del comienzo de la rutina de tratamiento del proceso usuario.

c) Si un proceso desea enviarle una señal a otro proceso, sea emparentado o no, debe utilizar la llamada al sistema **Kill.**

int kill(int iPid, int iSenial)

Los parámetros son los siguientes:

iPid: Pid del proceso al que se le desea enviar una señal.
iSenial: Nro de señal que se desea enviar.

d) Un proceso antes de finalizar su ejecución debe chequear si tiene alguna señal pendiente por atender, y en caso de existir una o más atenderlas. para esto debe utilizad la llamada al sistema **SignalR.**

int signalR()

La llamada al sistema no recibe parámetros.

5 Bibliografía

1. MSDN Microsoft , http://msdn.microsoft.com/en-us/library/aa365574%28v=vs.85%29.aspx

2. http://web.usal.es/~hernando/segi2010/colas.pdf

3. http://www.udb.edu.sv/Academia/Laboratorios/informatica/SistemasOperativos/guia7SO.pdf

4. W Richard Stevens , “Advance Programming in the Unix Enviornment” , ISBN: 0-201-56317-7,

Addison-Wesley

5. David A Rusling, El núcleo Linux (1998).

6. Manual de INTEL sección 3A - 3B.

7. William Stalings , Operating System – Internals And Design Pinciples, 5ta Edicion, ISBN: 0-13-147954-7, Editorial Prentice Hall.

8. http://www.chuidiang.com/clinux/ipcs/mem_comp.php