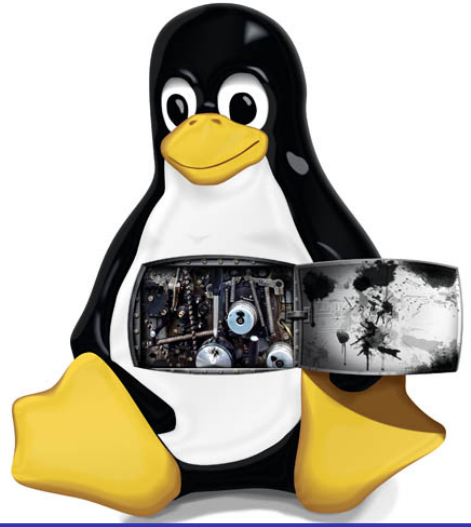


Informática I

Programación Avanzada - Comunicación entre Procesos - Señales, Pipes, Named FIFO's

Alejandro Furfaro

Departamento de Electrónica - UTN.BA

8 de octubre de 2018

Temario

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

Motivación de intercomunicar procesos

¿Que aprendimos hasta aquí?

Ya aprendimos a crear procesos utilizando `fork ()`. También comprendimos (muchas veces a costa de sufrir algunos inconvenientes) los riesgos que implica tener esta “llave maestra”.
Asumiendo que estos conceptos están ciertamente comprendidos, nos proponemos ahora intercomunicar esos procesos con los recursos que dispone el Sistema Operativo para tales fines.
Linux ofrece los mecanismos definidos en el estaándar POSIX.
Trataremos de ir desde los mas sencillos a los mas elaborados de modo de abordar el tema incrementalmente desde el punto de vista de su complejidad.

Contenido

1 Introducción

2 Señales
- Generalidades
- Señales mas relevantes
- System Calls para manejo de señales
- Señales Hands On.

3 pipes
- introducción
- pipe
- Named FIFOs

Contenido

1 Introducción

2 Señales
- Generalidades
- Señales mas relevantes
- System Calls para manejo de señales
- Señales Hands On.

3 pipes
- introducción
- pipe
- Named FIFOs

Defnición

- Son tal vez (además de uno de los primeros) el mas simple de los mecanismos de intercomunicación de Procesos que podamos encontrar.
- No transmiten datos en sí desde un proceso a otro, sino que lo que hacen en realidad es enviar a través de un servicio del kernel una suerte de aviso (es decir, ¡una señal!) a un proceso determinado.
- En respuesta a esa señal, el poceso destinatario realizara una acción determinada

Defnición

- Son tal vez (además de uno de los primeros) el mas simple de los mecanismos de intercomunicación de Procesos que podamos encontrar.
- No transmiten datos en sí desde un proceso a otro, sino que lo que hacen en realidad es enviar a través de un servicio del kernel una suerte de aviso (es decir, ¡una señal!) a un proceso determinado.
- En respuesta a esa señal, el poceso destinatario realizara una acción determinada

Defnición

- Son tal vez (además de uno de los primeros) el mas simple de los mecanismos de intercomunicación de Procesos que podamos encontrar.
- No transmiten datos en sí desde un proceso a otro, sino que lo que hacen en realidad es enviar a través de un servicio del kernel una suerte de aviso (es decir, ¡una señal!) a un proceso determinado.
- En respuesta a esa señal, el poceso destinatario realizara una acción determinada

POSIX

- El estándar POSIX define 32 señales para enviar a cada uno de los procesos junto con su comportamiento predeterminado, es decir, que es lo que se espera que haga el proceso receptor de la señal.
- El Kernel provee servicios para que un proceso pueda modificar el comportamiento predefinido para una o mas señales reemplazando la función que le asignó por defecto el Sistema Operativo por una función propia que le resultará mas conveniente.
- Veremos que hay una señal determinada que queda fuera de esta última posibilidad.
- Finalmente POSIX define también una system call para enviar señales a un proceso determinado.

POSIX

- El estándar POSIX define 32 señales para enviar a cada uno de los procesos junto con su comportamiento predeterminado, es decir, que es lo que se espera que haga el proceso receptor de la señal.
- El Kernel provee servicios para que un proceso pueda modificar el comportamiento predefinido para una o mas señales reemplazando la función que le asignó por defecto el Sistema Operativo por una función propia que le resultará mas conveniente.
- Veremos que hay una señal determinada que queda fuera de esta última posibilidad.
- Finalmente POSIX define también una system call para enviar señales a un proceso determinado.

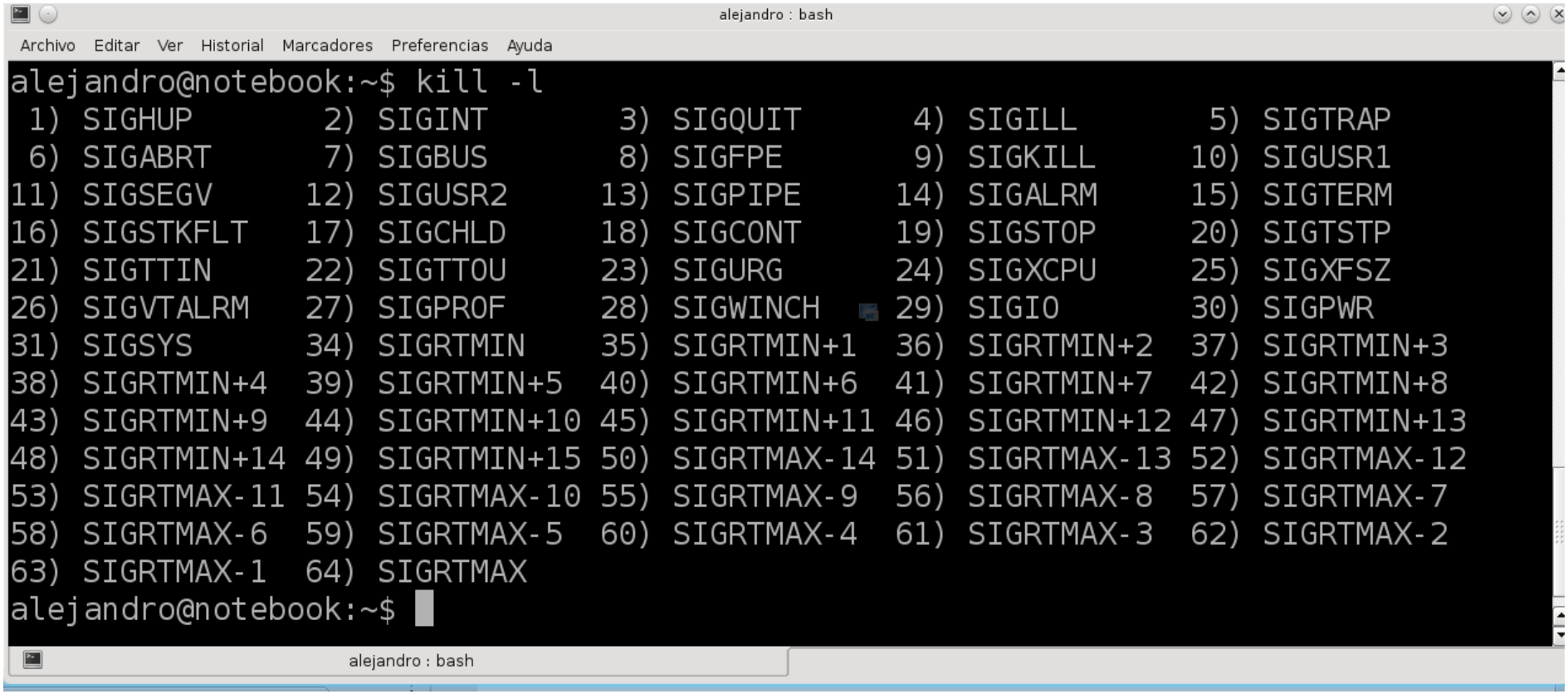
POSIX

- El estándar POSIX define 32 señales para enviar a cada uno de los procesos junto con su comportamiento predeterminado, es decir, que es lo que se espera que haga el proceso receptor de la señal.
- El Kernel provee servicios para que un proceso pueda modificar el comportamiento predefinido para una o mas señales reemplazando la función que le asignó por defecto el Sistema Operativo por una función propia que le resultará mas conveniente.
- Veremos que hay una señal determinada que queda fuera de esta última posibilidad.
- Finalmente POSIX define también una system call para enviar señales a un proceso determinado.

POSIX

- El estándar POSIX define 32 señales para enviar a cada uno de los procesos junto con su comportamiento predeterminado, es decir, que es lo que se espera que haga el proceso receptor de la señal.
- El Kernel provee servicios para que un proceso pueda modificar el comportamiento predefinido para una o mas señales reemplazando la función que le asignó por defecto el Sistema Operativo por una función propia que le resultará mas conveniente.
- Veremos que hay una señal determinada que queda fuera de esta última posibilidad.
- Finalmente POSIX define también una system call para enviar señales a un proceso determinado.

¿Como saber cuales son las señales disponibles?



El comando ***kill*** es el que utilizaremos desde el shell para enviar una señal a un proceso cualquiera. De hecho, estarán familiarizados a esta altura con kill -9 (llave de escape inapelable, que tantas veces nos salvó de situaciones indeseadas).

Nº	Nombre	Acción (default)	Descripción	POSIX
1	SIGHUP	Terminar	Suspende (Hang up) el control de una terminal o proceso	Si
2	SIGINT	Terminar	Interrupción desde teclado	Si
3	SIGQUIT	Volcar	Quit desde el teclado	Si
4	SIGILL	Volcar	Instrucción llegal	Si
5	SIGTRAP	Volcar	Breakpoint for debugging	No
6	SIGABRT	Volcar	Terminación Anormal	Si
6	SIGIOT	Volcar	Equivalente a SIGABRT	No
7	SIGBUS	Volcar	Error de Bus	No
8	SIGFPE	Volcar	Excepción de Floating-point	Si
9	**SIGKILL**	**Terminar**	**Terminación Forzada**	**Si**
10	SIGUSR1	Terminar	User Defined. Disponible	Si
11	SIGSEGV	Volcar	referencia inválida de memoria.	Si
12	SIGUSR2	Terminar	User Defined. Disponible	Si
13	SIGPIPE	Terminar	Escritura en un pipe sin lectores	Si
14	SIGALRM	Terminar	Real-timerclock	Si
15	SIGTERM	Terminar	Process termination	Si
16	SIGSTKFLT	Terminar	Coprocessor stack error	No
17	SIGCHLD	Ignorar	Proceso Hijo terminado o detenido	Si
18	SIGCONT	Continuar	Reasume la ejecución si estaba Detenido	Si
19	SIGSTOP	Detener	Detiene ejecución del proceso	Si
20	SIGTSTP	Detener	Detención del proceso enviada desde tty	Si
21	SIGTTIN	Detener	Proceso en Background requiere entrada	Si
22	SIGTTOU	Detener	Proceso en Background requiere salida	Si
23	SIGURG	Ignorar	Condición Urgente en socket	No
24	SIGXCPU	Volcar	Límite de tiempo de CPU excedido	No
25	SIGXFSZ	Volcar	Límte de tamaño de archivo superado	No
26	SIGVTALRM	Terminar	Virtual timer clock	No
27	SIGPROF	Terminar	Perfil del timer clock	No
28	SIGWINCH	Ignorar	Window resizing	No
29	SIGIO	Terminar	I/O imposible	No
29	SIGPOLL	Terminar	Equivalente a SIGIO	No
30	SIGPWR	Terminar	Falla en fuente de alimentación	No
31	SIGSYS	Volcar	System call errónea	No
32	SIGUNUSED	Volcar	Equivalente a SIGSYS	No

Acciones

- **Terminar**: Finaliza la ejecución del proceso. Realiza las mismas actividades de la system call ***exit ()***.
- **Volcar**: (Dump), finaliza la ejecución del proceso pero con el agregado de la generación de un archivo imagen core. Esta imagen tiene fines de proveer información a un debugger como gdb, el que puede obtener de ella la información del estado del procesador y demás cuestiones inherentes al error que se produjo en el código del proceso y que motivó el envío de la señal desde el kernel al proceso para su terminación.
- **Detener**: (Stop), suspende la ejecución del proceso (no lo termina) simplemente lo coloca en estado TASK_STOPPED. Puede reasumirse mediante el envío de SIGCONT
- **Continuar**: Reasume la ejecución de un proceso detenido
- **Ignorar**: La señal no tiene efecto, el proceso receptor por defecto la descarta.

Acciones

- **Terminar**: Finaliza la ejecución del proceso. Realiza las mismas actividades de la system call `exit ()`.
- **Volcar**: (Dump), finaliza la ejecución del proceso pero con el agregado de la generación de un archivo imagen core. Esta imagen tiene fines de proveer información a un debugger como gdb, el que puede obtener de ella la información del estado del procesador y demás cuestiones inherentes al error que se produjo en el código del proceso y que motivó el envío de la señal desde el kernel al proceso para su terminación.
- **Detener**: (Stop), suspende la ejecución del proceso (no lo termina) simplemente lo coloca en estado TASK_STOPPED. Puede reasumirse mediante el envío de SIGCONT
- **Continuar**: Reasume la ejecución de un proceso detenido
- **Ignorar**: La señal no tiene efecto, el proceso receptor por defecto la descarta.

Acciones

- **Terminar**: Finaliza la ejecución del proceso. Realiza las mismas actividades de la system call `exit ()`.
- **Volcar**: (Dump), finaliza la ejecución del proceso pero con el agregado de la generación de un archivo imagen core. Esta imagen tiene fines de proveer información a un debugger como gdb, el que puede obtener de ella la información del estado del procesador y demás cuestiones inherentes al error que se produjo en el código del proceso y que motivó el envío de la señal desde el kernel al proceso para su terminación.
- **Detener**: (Stop), suspende la ejecución del proceso (no lo termina) simplemente lo coloca en estado TASK_STOPPED. Puede reasumirse mediante el envío de SIGCONT
- **Continuar**: Reasume la ejecución de un proceso detenido
- **Ignorar**: La señal no tiene efecto, el proceso receptor por defecto la descarta.

Acciones

- **Terminar**: Finaliza la ejecución del proceso. Realiza las mismas actividades de la system call `exit ()`.
- **Volcar**: (Dump), finaliza la ejecución del proceso pero con el agregado de la generación de un archivo imagen core. Esta imagen tiene fines de proveer información a un debugger como gdb, el que puede obtener de ella la información del estado del procesador y demás cuestiones inherentes al error que se produjo en el código del proceso y que motivó el envío de la señal desde el kernel al proceso para su terminación.
- **Detener**: (Stop), suspende la ejecución del proceso (no lo termina) simplemente lo coloca en estado TASK_STOPPED. Puede reasumirse mediante el envío de SIGCONT
- **Continuar**: Reasume la ejecución de un proceso detenido
- **Ignorar**: La señal no tiene efecto, el proceso receptor por defecto la descarta.

Acciones

- **Terminar**: Finaliza la ejecución del proceso. Realiza las mismas actividades de la system call ***exit ()***.
- **Volcar**: (Dump), finaliza la ejecución del proceso pero con el agregado de la generación de un archivo imagen core. Esta imagen tiene fines de proveer información a un debugger como gdb, el que puede obtener de ella la información del estado del procesador y demás cuestiones inherentes al error que se produjo en el código del proceso y que motivó el envío de la señal desde el kernel al proceso para su terminación.
- **Detener**: (Stop), suspende la ejecución del proceso (no lo termina) simplemente lo coloca en estado TASK_STOPPED. Puede reasumirse mediante el envío de SIGCONT
- **Continuar**: Reasume la ejecución de un proceso detenido
- **Ignorar**: La señal no tiene efecto, el proceso receptor por defecto la descarta.

Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

SIGKILL

- Es la que enviamos a un proceso cuya finalización queremos asegurar mediante el comando ***kill -9*** seguido del process ID de dicho proceso.
- Desde el shell podemos enviar las señales especificando su número o su nombre, de modo que es equivalente a enviar ***kill -SIGKILL***.
- **Particularidad**: No puede ser modificado su comportamiento mediante el reemplazo de su función establecida por el kernel.
- Esto significa que ningún proceso puede convertirse por si mismo en No Interrumpible. De otro modo no tendríamos forma de detener a un proceso que está por una falla en su código, consumiendo CPU y/o memoria en exceso, o en un dead lock.
- Esta señal es generalmente inapelable. Sin embargo si el proceso destino está en estado `TASK_UNINTERRUPTIBLE` o `EXIT_ZOMBIE`, es decir aquellos que visualizamos con la D o la Z respectivamente en `top`, `htop`, o `ps`, `SIGKILL` no lo afecta.

SIGKILL

- Es la que enviamos a un proceso cuya finalización queremos asegurar mediante el comando ***kill -9*** seguido del process ID de dicho proceso.
- Desde el shell podemos enviar las señales especificando su número o su nombre, de modo que es equivalente a enviar ***kill -SIGKILL***.
- **Particularidad**: No puede ser modificado su comportamiento mediante el reemplazo de su función establecida por el kernel.
- Esto significa que ningún proceso puede convertirse por si mismo en No Interrumpible. De otro modo no tendríamos forma de detener a un proceso que está por una falla en su código, consumiendo CPU y/o memoria en exceso, o en un dead lock.
- Esta señal es generalmente inapelable. Sin embargo si el proceso destino está en estado `TASK_UNINTERRUPTIBLE` o `EXIT_ZOMBIE`, es decir aquellos que visualizamos con la D o la Z respectivamente en `top`, `htop`, o `ps`, `SIGKILL` no lo afecta.

SIGKILL

- Es la que enviamos a un proceso cuya finalización queremos asegurar mediante el comando ***kill -9*** seguido del process ID de dicho proceso.
- Desde el shell podemos enviar las señales especificando su número o su nombre, de modo que es equivalente a enviar ***kill -SIGKILL***.
- **Particularidad**: No puede ser modificado su comportamiento mediante el reemplazo de su función establecida por el kernel.
- Esto significa que ningún proceso puede convertirse por si mismo en No Interrumpible. De otro modo no tendríamos forma de detener a un proceso que está por una falla en su código, consumiendo CPU y/o memoria en exceso, o en un dead lock.
- Esta señal es generalmente inapelable. Sin embargo si el proceso destino está en estado `TASK_UNINTERRUPTIBLE` o `EXIT_ZOMBIE`, es decir aquellos que visualizamos con la D o la Z respectivamente en `top`, `htop`, o `ps`, `SIGKILL` no lo afecta.

SIGKILL

- Es la que enviamos a un proceso cuya finalización queremos asegurar mediante el comando ***kill -9*** seguido del process ID de dicho proceso.
- Desde el shell podemos enviar las señales especificando su número o su nombre, de modo que es equivalente a enviar ***kill -SIGKILL***.
- **Particularidad**: No puede ser modificado su comportamiento mediante el reemplazo de su función establecida por el kernel.
- Esto significa que ningún proceso puede convertirse por si mismo en No Interrumpible. De otro modo no tendríamos forma de detener a un proceso que está por una falla en su código, consumiendo CPU y/o memoria en exceso, o en un dead lock.
- Esta señal es generalmente inapelable. Sin embargo si el proceso destino está en estado `TASK_UNINTERRUPTIBLE` o `EXIT_ZOMBIE`, es decir aquellos que visualizamos con la D o la Z respectivamente en `top`, `htop`, o `ps`, `SIGKILL` no lo afecta.

SIGKILL

- Es la que enviamos a un proceso cuya finalización queremos asegurar mediante el comando ***kill -9*** seguido del process ID de dicho proceso.
- Desde el shell podemos enviar las señales especificando su número o su nombre, de modo que es equivalente a enviar ***kill -SIGKILL***.
- **Particularidad**: No puede ser modificado su comportamiento mediante el reemplazo de su función establecida por el kernel.
- Esto significa que ningún proceso puede convertirse por si mismo en No Interrumpible. De otro modo no tendríamos forma de detener a un proceso que está por una falla en su código, consumiendo CPU y/o memoria en exceso, o en un dead lock.
- Esta señal es generalmente inapelable. Sin embargo si el proceso destino está en estado **TASK_UNINTERRUPTIBLE** o **EXIT_ZOMBIE**, es decir aquellos que visualizamos con la D o la Z respectivamente en **top**, **htop**, o **ps**, **SIGKILL** no lo afecta.

SIGTERM

- Es la que enviamos a un proceso cuya finalización queremos indicar mediante el comando ***kill*** seguido del process ID de dicho proceso.
- Si bien la misma acción default, que **SIGKILL** no actúa si el proceso está bloqueado esperando un evento o actualizando un archivo, por ejemplo.
- **SIGKILL** en cambio , es implacable. Lo termina de cualquier forma, y puede hacernos perder información si el archivo está ocupado actualizando un log o lo que fuere que lo tuviese ocupado.
- Lo recomendable es tratar con **SIGTERM**, y en caso de no tener resultados y ser imprescindible terminar el proceso sin emportar el costo, optar por **SIGKILL**.

SIGTERM

- Es la que enviamos a un proceso cuya finalización queremos indicar mediante el comando ***kill*** seguido del process ID de dicho proceso.
- Si bien la misma acción default, que **SIGKILL** no actúa si el proceso está bloqueado esperando un evento o actualizando un archivo, por ejemplo.
- **SIGKILL** en cambio , es implacable. Lo termina de cualquier forma, y puede hacernos perder información si el archivo está ocupado actualizando un log o lo que fuere que lo tuviese ocupado.
- Lo recomendable es tratar con **SIGTERM**, y en caso de no tener resultados y ser imprescindible terminar el proceso sin emportar el costo, optar por **SIGKILL**.

SIGTERM

- Es la que enviamos a un proceso cuya finalización queremos indicar mediante el comando ***kill*** seguido del process ID de dicho proceso.
- Si bien la misma acción default, que **SIGKILL** no actúa si el proceso está bloqueado esperando un evento o actualizando un archivo, por ejemplo.
- **SIGKILL** en cambio , es implacable. Lo termina de cualquier forma, y puede hacernos perder información si el archivo está ocupado actualizando un log o lo que fuere que lo tuviese ocupado.
- Lo recomendable es tratar con **SIGTERM**, y en caso de no tener resultados y ser imprescindible terminar el proceso sin emportar el costo, optar por **SIGKILL**.

SIGTERM

- Es la que enviamos a un proceso cuya finalización queremos indicar mediante el comando ***kill*** seguido del process ID de dicho proceso.
- Si bien la misma acción default, que **SIGKILL** no actúa si el proceso está bloqueado esperando un evento o actualizando un archivo, por ejemplo.
- **SIGKILL** en cambio , es implacable. Lo termina de cualquier forma, y puede hacernos perder información si el archivo está ocupado actualizando un log o lo que fuere que lo tuviese ocupado.
- Lo recomendable es tratar con **SIGTERM**, y en caso de no tener resultados y ser imprescindible terminar el proceso sin emportar el costo, optar por **SIGKILL**.

SIGCHLD

- Cada vez que un proceso termina, el kernel envía al proceso padre una señal **SIGCHLD**. Esta señal advierte al proceso padre acerca de la terminación o detención de un proceso creado por él.
- Para entender la utilidad de esta señal es necesario comprender que necesita hacer el kernel para terminar un proceso.
  - Cuando un proceso ejecuta la función `fork()`, no se duplican los espacios de memoria de código ni de datos.
  - Cuando uno de los procesos modifica una variable, el sistema operativo duplica la página de memoria que contiene la variable para ese proceso. (Mecanismo Copy-On-Write)

SIGCHLD

- Cada vez que un proceso termina, el kernel envía al proceso padre una señal **SIGCHLD**. Esta señal advierte al proceso padre acerca de la terminación o detención de un proceso creado por él.
- Para entender la utilidad de esta señal es necesario comprender que necesita hacer el kernel para terminar un proceso.
  - Cuando un proceso ejecuta la función `fork()`, no se duplican los espacios de memoria de código ni de datos.
  - Cuando uno de lo procesos modifica una variable, el sistema operativo duplica la página de memoria que contiene el dato solo para ese proceso. (Mecanismo ***Copy-On-Write***).

SIGCHLD

- Cada vez que un proceso termina, el kernel envía al proceso padre una señal **SIGCHLD**. Esta señal advierte al proceso padre acerca de la terminación o detención de un proceso creado por él.
- Para entender la utilidad de esta señal es necesario comprender que necesita hacer el kernel para terminar un proceso.
  - Cuando un proceso ejecuta la función ***fork ()***, no se duplican los espacios de memoria de código ni de datos.
  - Cuando uno de lo procesos modifica una variable, el sistema operativo duplica la página de memoria que contiene el dato solo para ese proceso. (Mecanismo ***Copy-On-Write***).

SIGCHLD

- Cada vez que un proceso termina, el kernel envía al proceso padre una señal **SIGCHLD**. Esta señal advierte al proceso padre acerca de la terminación o detención de un proceso creado por él.
- Para entender la utilidad de esta señal es necesario comprender que necesita hacer el kernel para terminar un proceso.
  - Cuando un proceso ejecuta la función ***`fork ()`***, no se duplican los espacios de memoria de código ni de datos.
  - Cuando uno de lo procesos modifica una variable, el sistema operativo duplica la página de memoria que contiene el dato solo para ese proceso. (Mecanismo ***Copy-On-Write***).

SIGCHLD

- Cuando un proceso termina, el kernel eventualmente debe actualizar información en el área de control del proceso padre.
- Pero no puede hacerlo sin cerciorarse que el proceso padre, al ser interrumpido por el scheduler la última vez, no haya quedado en medio de una operación que también requiera actualizar información en su área de control.
- En tal caso actualizarla para cerrar un proceso hijo resultaría riesgoso, ya que podría sobreescribir información en uso por parte del proceso padre y que éste requeriría al ser reasumido.
- El kernel solo actualizará el área de control del proceso padre cuando éste esté ejecutando una función que lo mantenga bloqueado y en espera para tal fin.

SIGCHLD

- Cuando un proceso termina, el kernel eventualmente debe actualizar información en el área de control del proceso padre.
- Pero no puede hacerlo sin cerciorarse que el proceso padre, al ser interrumpido por el scheduler la última vez, no haya quedado en medio de una operación que también requiera actualizar información en su área de control.
- En tal caso actualizarla para cerrar un proceso hijo resultaría riesgoso, ya que podría sobreescribir información en uso por parte del proceso padre y que éste requeriría al ser reasumido.
- El kernel solo actualizará el área de control del proceso padre cuando éste esté ejecutando una función que lo mantenga bloqueado y en espera para tal fin.

SIGCHLD

- Cuando un proceso termina, el kernel eventualmente debe actualizar información en el área de control del proceso padre.
- Pero no puede hacerlo sin cerciorarse que el proceso padre, al ser interrumpido por el scheduler la última vez, no haya quedado en medio de una operación que también requiera actualizar información en su área de control.
- En tal caso actualizarla para cerrar un proceso hijo resultaría riesgoso, ya que podría sobreescribir información en uso por parte del proceso padre y que éste requeriría al ser reasumido.
- El kernel solo actualizará el área de control del proceso padre cuando éste esté ejecutando una función que lo mantenga bloqueado y en espera para tal fin.

SIGCHLD

- Cuando un proceso termina, el kernel eventualmente debe actualizar información en el área de control del proceso padre.
- Pero no puede hacerlo sin cerciorarse que el proceso padre, al ser interrumpido por el scheduler la última vez, no haya quedado en medio de una operación que también requiera actualizar información en su área de control.
- En tal caso actualizarla para cerrar un proceso hijo resultaría riesgoso, ya que podría sobreescribir información en uso por parte del proceso padre y que éste requeriría al ser reasumido.
- El kernel solo actualizará el área de control del proceso padre cuando éste esté ejecutando una función que lo mantenga bloqueado y en espera para tal fin.

SIGCHLD

- Esa función es ***wait()*** o ***waitpid()***.
- Ambas bloquean al proceso que las invoca de modo que no resultaría práctica su inserción en el flujo principal del programa, ya que el proceso padre no podría hacer nada mas hasta tanto no termine la instancia child que termina de generar, perdiéndose las ventajas de paralelización de código que muchas veces se busca con esta técnica.
- Podemos reemplazar la función predefinida por el kernel para la Señal **SIGCHLD** por una función propia del proceso en donde se ejecuten ***wait()*** o ***waitpid()***.
- De este modo el tiempo de bloqueo del proceso será mínimo ya que se invoca a la función cuando el proceso child ha concluido (prueba de ello es que estamos dentro de esta función a la que solo se ingresa si el proceso ha recibido la señal **SIGCHLD**).

SIGCHLD

- Esa función es ***wait()*** o ***waitpid()***.
- Ambas bloquean al proceso que las invoca de modo que no resultaría práctica su inserción en el flujo principal del programa, ya que el proceso padre no podría hacer nada mas hasta tanto no termine la instancia child que termina de generar, perdiéndose las ventajas de paralelización de código que muchas veces se busca con esta técnica.
- Podemos reemplazar la función predefinida por el kernel para la Señal **SIGCHLD** por una función propia del proceso en donde se ejecuten ***wait()*** o ***waitpid()***.
- De este modo el tiempo de bloqueo del proceso será mínimo ya que se invoca a la función cuando el proceso child ha concluido (prueba de ello es que estamos dentro de esta función a la que solo se ingresa si el proceso ha recibido la señal **SIGCHLD**).

SIGCHLD

- Esa función es ***wait()*** o ***waitpid()***.
- Ambas bloquean al proceso que las invoca de modo que no resultaría práctica su inserción en el flujo principal del programa, ya que el proceso padre no podría hacer nada mas hasta tanto no termine la instancia child que termina de generar, perdiéndose las ventajas de paralelización de código que muchas veces se busca con esta técnica.
- Podemos reemplazar la función predefinida por el kernel para la Señal **SIGCHLD** por una función propia del proceso en donde se ejecuten ***wait()*** o ***waitpid()***.
- De este modo el tiempo de bloqueo del proceso será mínimo ya que se invoca a la función cuando el proceso child ha concluido (prueba de ello es que estamos dentro de esta función a la que solo se ingresa si el proceso ha recibido la señal **SIGCHLD**).

SIGCHLD

- Esa función es ***wait()*** o ***waitpid()***.
- Ambas bloquean al proceso que las invoca de modo que no resultaría práctica su inserción en el flujo principal del programa, ya que el proceso padre no podría hacer nada mas hasta tanto no termine la instancia child que termina de generar, perdiéndose las ventajas de paralelización de código que muchas veces se busca con esta técnica.
- Podemos reemplazar la función predefinida por el kernel para la Señal **SIGCHLD** por una función propia del proceso en donde se ejecuten ***wait()*** o ***waitpid()***.
- De este modo el tiempo de bloqueo del proceso será mínimo ya que se invoca a la función cuando el proceso child ha concluido (prueba de ello es que estamos dentro de esta función a la que solo se ingresa si el proceso ha recibido la señal **SIGCHLD**).

SIGCHLD

Y sino? Que pasaría?

Si el proceso padre no espera la finalización del proceso hijo, éste no podrá concluir ya que el kernel no ingresará al área de control del proceso padre que le permitirá terminar de actualizar toda la información que se requiere para cerrar adecuadamente al proceso hijo.

El proceso hijo en esta condición quedará inconcluso y en estado Zombie, (o *defunct* como se lo suele denominar en la jerga UNIX).

En este estado consume memoria y una entrada en la tabla de procesos pero no responde ni hace nada útil. Es claramente una situación que debemos evitar.

Señales desde teclado

- **SIGINT** (2), es la señal que recibe un proceso cuando pulsamos CTRL-C desde del teclado,
- **SIGSTOP** (19) cuando se pulsa CTRL-Z.
- Ésta última, al igual que **SIGKILL** no acepta modificación a su handler, por parte de un proceso, de modo que un proceso no puede decidir por si mismo no ser Detenible por parte de otro proceso o mediante una señal enviada desde el shell de modo teto.

Señales desde teclado

- **`SIGINT`** (2), es la señal que recibe un proceso cuando pulsamos CTRL-C desde del teclado,
- **`SIGSTOP`** (19) cuando se pulsa CTRL-Z.
- Ésta última, al igual que **`SIGKILL`** no acepta modificación a su handler, por parte de un proceso, de modo que un proceso no puede decidir por si mismo no ser Detenible por parte de otro proceso o mediante una señal enviada desde el shell de modo teto.

Señales desde teclado

- **`SIGINT`** (2), es la señal que recibe un proceso cuando pulsamos CTRL-C desde del teclado,
- **`SIGSTOP`** (19) cuando se pulsa CTRL-Z.
- Ésta última, al igual que **`SIGKILL`** no acepta modificación a su handler, por parte de un proceso, de modo que un proceso no puede decidir por si mismo no ser Detenible por parte de otro proceso o mediante una señal enviada desde el shell de modo teto.

Señales asociadas con comandos

- **`SIGCONT`** (18), es la señal que reanuda la ejecución de un proceso con los comandos **fg** o **bg**, colocándolo respectivamente en, primer plano o en segundo plano liberando la terminal al usuario.

Temporizaciones mediante señales

- **SIGALRM** (14), es una señal que recibe el proceso luego de los ***n*** segundos establecidos mediante la syscall ***alarm (n)***.
- Es una buena forma de generar eventos periódicos.
- La primer vez ***alarm (n)*** se invoca desde el punto del programa mas adecuado para el caso. Luego dentro del handler de **SIGALRM** (reemplazado mediante la syscall ***signal ()*** al inicio del programa.
- Luego se establece como handler una función del tipo:

```
void sigalarm_handler (int sig)
{
  alarm (delay);
  /*accion a realizar a intervalos regulares*/
  return;
}
```

Temporizaciones mediante señales

- **SIGALRM** (14), es una señal que recibe el proceso luego de los ***n*** segundos establecidos mediante la syscall ***alarm (n)***.
- Es una buena forma de generar eventos periódicos.
- La primer vez ***alarm (n)*** se invoca desde el punto del programa mas adecuado para el caso. Luego dentro del handler de **SIGALRM** (reemplazado mediante la syscall ***signal ()*** al inicio del programa.
- Luego se establece como handler una función del tipo:

```
void sigalarm_handler (int sig)
{
  alarm (delay);
  /*accion a realizar a intervalos regulares*/
  return;
}
```

Temporizaciones mediante señales

- **SIGALRM** (14), es una señal que recibe el proceso luego de los ***n*** segundos establecidos mediante la syscall ***alarm (n)***.
- Es una buena forma de generar eventos periódicos.
- La primer vez ***alarm (n)*** se invoca desde el punto del programa mas adecuado para el caso. Luego dentro del handler de **SIGALRM** (reemplazado mediante la syscall ***signal ()*** al inicio del programa.
- Luego se establece como handler una función del tipo:

```
void sigalarm_handler (int sig)
{
  alarm (delay);
  /*accion a realizar a intervalos regulares*/
  return;
}
```

Temporizaciones mediante señales

- **SIGALRM** (14), es una señal que recibe el proceso luego de los ***n*** segundos establecidos mediante la syscall ***alarm (n)***.
- Es una buena forma de generar eventos periódicos.
- La primer vez ***alarm (n)*** se invoca desde el punto del programa mas adecuado para el caso. Luego dentro del handler de **SIGALRM** (reemplazado mediante la syscall ***signal ()*** al inicio del programa.
- Luego se establece como handler una función del tipo:

```
void sigalarm_handler (int sig)
{
  alarm (delay);
  /*accion a realizar a intervalos regulares*/
  return;
}
```

Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

kill()

Prototipo:

```
int kill(pid_t pid, int sig);
```

Headers:

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

Envía la señal que recibe en el argumento ***sig*** a un proceso o grupo de procesos, definidos en el argumento ***pid***.
Si ***pid*** > 0, la señal ***sig*** es enviada a ***pid***. En este caso, se devuelve 0 si hay éxito, o un valor negativo si hubo algún error.
Si ***pid*** = 0, ***sig*** se envía a cada proceso que tenga el mismo GroupID (grupo de procesos) del proceso que envía la señal.
Si ***pid*** = -1, ***sig*** se envía a cada proceso, excepto al proceso 1 (init).
Si ***pid*** < -1, ***sig*** se envía a cada proceso en el grupo de procesos ***-pid***.
Si ***sig*** = 0, no se envía ninguna señal pero se realiza la comprobación de errores.

signal()

Prototipo:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Headers:

```
#include <signal.h>
```

Es utilizada por un proceso para reemplazar su handler de señal (normalmentte predefinido por el sistema operativo) por otro que especifica en su propio código. De este modo colocando en el segundo argumento un puntero a la función deseada el proceso puede reemplazar el handler que recibió al ser creado por otro afín a sus propósitos. También pueden utilizarse macros para ignorar la señal (**SIG_IGN**) o para utilizar el handler default en caso de querer recuperarlo (**SIG_DFL**).

Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

Modo simple de bloqueo de una señal

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char * argv[])
{
  if (argc < 2 ) {
    printf ("Error de argumentos\nUso: noint [nro-señal]\n");
    exit (1);
  }
  /* Modificamos el handler de la señal recibida como argumento
     , por otro al que ignore.*/
  if (signal(atoi(argv[1]),SIG_IGN) == SIG_ERR)  {
    printf ("Error al trapear la señal % d\n", atoi(argv[1]));
    exit (1);
  }
  while (1)
    sleep(1);
  return;
}
```

Comprobaciones

- Para compilar:

```
$ gcc  -o noint noint.c
```

- Verificamos que no es posible cambiar el handler que el Sistema Operativo le instala al proceso en el momento de crearlo, para la señal **SIGKILL**.

```
$ ./noint 9
No se puede trapear la señal 9
$
```

Comprobaciones

- Para compilar:

```
$ gcc  -o noint noint.c
```

- Verificamos que no es posible cambiar el handler que el Sistema Operativo le instala al proceso en el momento de crearlo, para la señal **SIGKILL**.

```
$ ./noint 9
No se puede trapear la señal 9
$
```

Comprobaciones

- Verificamos que no es posible cambiar el handler que el Sistema Operativo le instala al proceso en el momento de crearlo, para la señal **SIGSTOP**.

```
$ ./noint 19
No se puede trapear la señal 19
$
```

- Y en el caso de no querer que nuestro proceso sea cancelado con la combinación CTRL-C desde el teclado, se logra ignorando **SIGQUIT**. En este caso solo podemos cancelarlo con el comando *kill* desde una consola de texto.

```
$ ./noint 2
^C^C^C
```

Comprobaciones

- Verificamos que no es posible cambiar el handler que el Sistema Operativo le instala al proceso en el momento de crearlo, para la señal **SIGSTOP**.

```
$ ./noint 19
No se puede trapear la señal 19
$
```

- Y en el caso de no querer que nuestro proceso sea cancelado con la combinación CTRL-C desde el teclado, se logra ignorando **SIGQUIT**. En este caso solo podemos cancelarlo con el comando ***kill*** desde una consola de texto.

```
$ ./noint 2
^C^C^C
```

Segundo Programa

handler multiseñal

En base a lo visto, vamos a escribir un programa simple que con un único handler de señal cambie el comportamiento de todas las señales que aceptan cambiar su handler.
Aprovechar el número de señal recibido en forma de argumento por la función que obra como handler de la señal para loguear en un archivo un mensaje que indique “La señal ***n*** ha reemplazado su handler”.

Tercer Programa

init

Vamos a escribir un programa que simplemente cree una instancia child con la syscall **`fork ()`**.
Luego de **`fork ()`**, la instancia padre duerme durante 20 segundos y luego termina. La instancia hijo duerme 40 segundos y luego termina.

Para compilar y probar trabajar con dos instancias de consolas, una en cada mitad horizontal de la pantalla. En una consola lanzar el programa y en la otra visualizar el estado de los procesos (usar **`ps -el | grep [proceso]`** y observar el estado de cada instancia.
¿Que pasa con el proceso hijo cuando termina el proceso padre?
¿Terminan normalmente?

Cuarto Programa

programa zombie

Editar el código del programa anterior e invertir los tiempos utilizados en `sleep ()` entre las instancias padre e hijo. Compilar y ejecutar. ¿Como es el comportamiento en este caso del proceso hijo?¿Termina normalmente? ¿Que ocurre cuando termina su ejecución el proceso padre?

Quinto Programa

Solución

Editar el código del programa anterior e incluir en el proceso padre un handler para la señal **`SIGCHLD`** que permita solucionar el problema utilizando la syscall **`wait ()`**.
¿Como es el comportamiento en este caso del proceso hijo?¿Termina normalmente?

Contenido

1 Introducción

2 Señales
- Generalidades
- Señales mas relevantes
- System Calls para manejo de señales
- Señales Hands On.

3 pipes
- introducción
- pipe
- Named FIFOs

Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

Generalidades

- Las señales son un IPC tan rudimentario que no proveen intercambio de datos entre dos procesos. Solo sirven para señalar eventos.
- Para intercambiar información entre procesos los primeros mecanismos contemplados en el estándar POSIX son los *pipes*.
- Un pipe es un nodo en el file system que se comporta entonces como un archivo pero con algunas diferencias conceptuales:
  - Se leen de manera secuencial, comenzando por el primer byte escrito y finalizando en el último byte escrito. Es decir, son dispositivos FIFO (First In First Out).
  - Los datos leídos son extraídos del pipe (lectura destructiva).

Generalidades

- Las señales son un IPC tan rudimentario que no proveen intercambio de datos entre dos procesos. Solo sirven para señalar eventos.
- Para intercambiar información entre procesos los primeros mecanismos contemplados en el estándar POSIX son los *pipes*.
- Un pipe es un nodo en el file system que se comporta entonces como un archivo pero con algunas diferencias conceptuales:
  - Se leen de manera secuencial comenzando por el primer byte escrito y finalizando en el último byte escrito. Es decir son dispositivos **FIFO** (Fisrt In First Out)
  - Los datos leídos son extraídos del pipe (lectura destructiva)

Generalidades

- Las señales son un IPC tan rudimentario que no proveen intercambio de datos entre dos procesos. Solo sirven para señalar eventos.
- Para intercambiar información entre procesos los primeros mecanismos contemplados en el estándar POSIX son los *pipes*.
- Un pipe es un nodo en el file system que se comporta entonces como un archivo pero con algunas diferencias conceptuales:
  - Se leen de manera secuencial comenzando por el primer byte escrito y finalizando en el último byte escrito. Es decir son dispositivos **FIFO** (**F**isrt **I**n **F**irst **O**ut)
  - Los datos leídos son extraídos del pipe (lectura destructiva)

Generalidades

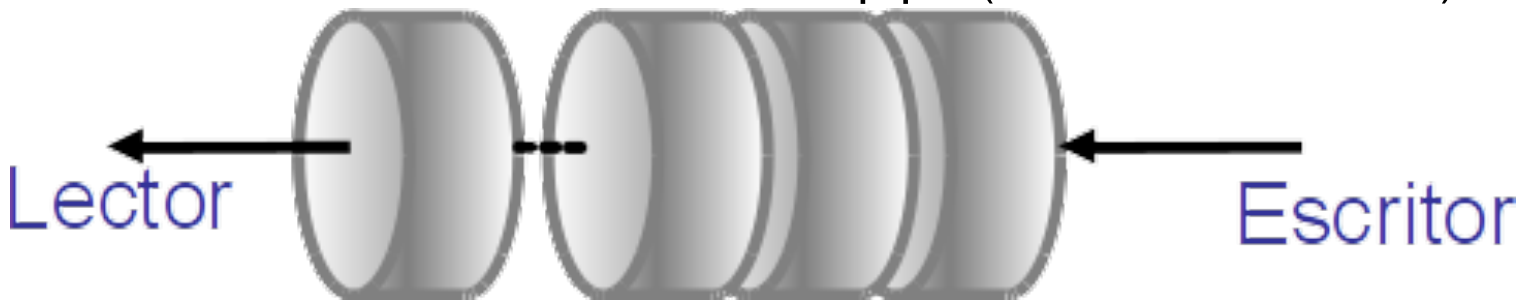
- Las señales son un IPC tan rudimentario que no proveen intercambio de datos entre dos procesos. Solo sirven para señalar eventos.
- Para intercambiar información entre procesos los primeros mecanismos contemplados en el estándar POSIX son los *pipes*.
- Un pipe es un nodo en el file system que se comporta entonces como un archivo pero con algunas diferencias conceptuales:
  - Se leen de manera secuencial comenzando por el primer byte escrito y finalizando en el último byte escrito. Es decir son dispositivos **FIFO** (**F**isrt **I**n **F**irst **O**ut)
  - Los datos leídos son extraídos del pipe (lectura destructiva)

Generalidades

- Las señales son un IPC tan rudimentario que no proveen intercambio de datos entre dos procesos. Solo sirven para señalar eventos.
- Para intercambiar información entre procesos los primeros mecanismos contemplados en el estándar POSIX son los *pipes*.
- Un pipe es un nodo en el file system que se comporta entonces como un archivo pero con algunas diferencias conceptuales:
  - Se leen de manera secuencial comenzando por el primer byte escrito y finalizando en el último byte escrito. Es decir son dispositivos **FIFO** (**F**isrt **I**n **F**irst **O**ut)
  - Los datos leídos son extraídos del pipe (lectura destructiva)

Generalidades

- Las señales son un IPC tan rudimentario que no proveen intercambio de datos entre dos procesos. Solo sirven para señalar eventos.
- Para intercambiar información entre procesos los primeros mecanismos contemplados en el estándar POSIX son los *pipes*.
- Un pipe es un nodo en el file system que se comporta entonces como un archivo pero con algunas diferencias conceptuales:
  - Se leen de manera secuencial comenzando por el primer byte escrito y finalizando en el último byte escrito. Es decir son dispositivos **FIFO** (**F**isrt **I**n **F**irst **O**ut)
  - Los datos leídos son extraídos del pipe (lectura destructiva)



Clasificación

En principio hay dos tipos de pipes:

1. **pipes**
2. **named fifos**

Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

Características

- Se crean mediante la system call `pipe ()`
- Como resultado se obtiene un array de dos file descriptors. El primero de los dos (el elemento 0 del array) se empleará para lectura y el segundo (el elemento 1 del array) se empleará para escritura.

Características

- Se crean mediante la system call `pipe ()`
- Como resultado se obtiene un array de dos file descriptors. El primero de los dos (el elemento 0 del array) se empleará para lectura y el segundo (el elemento 1 del array) se empleará para escritura.

Características

- Se crean mediante la system call `pipe ()`
- Como resultado se obtiene un array de dos file descriptors. El primero de los dos (el elemento 0 del array) se empleará para lectura y el segundo (el elemento 1 del array) se empleará para escritura.

Usamos pipes todo el tiempo!

Los utilizamos hace poco tiempo para probar el comportamiento de los ejemplos de señales.

Cuando queremos ver el estado de un proceso particular (por ejemplo firefox) recurrimos a esta orden:

```
$ ps -elf | grep firefox
```

Este es el efecto

```
alejandro@DarkSideOfTheMoon:~$ ps -elf | grep firefox
0 S alejand+  3786  3565 11  80   0 - 548596 -     feb25 ?        04:44:34 /opt/firefox/firefox --sm-co
nfig-prefix /firefox-itVpdY/ --sm-client-id 101b917d1991a70001456400923000000036010041 --screen 0
0 S alejand+ 32097 32078  0  80   0 -  3187 -     10:51 pts/5    00:00:00 grep firefox
alejandro@DarkSideOfTheMoon:~$
```

El caracter '|' casualmente se llama *pipe*...

Conectando procesos

- En un sistema POSIX cuando se crea un proceso se le asignan los siguientes file descriptors predeterminados
  - 0 FILENO_STDIN; File descriptor de ***stdin***, Standard input (teclado)
  - 1 FILENO_STDOUT; File Descriptor de ***stdout***, Standard Output (pantalla)
  - 2 FILENO_STDERR; File Descriptor de ***stderr*** Standard Error (generalmente pantalla)
- Operador '|': Conecta la salida de un comando con la entrada del siguiente. En otras palabras aplica una redirección del stdout del primer proceso, al stdin del segundo

Conectando procesos

- En un sistema POSIX cuando se crea un proceso se le asignan los siguientes file descriptors predeterminados
  - 0 FILENO_STDIN; File descriptor de ***stdin***, Standard input (teclado)
  - 1 FILENO_STDOUT; File Descriptor de ***stdout***, Standard Output (pantalla)
  - 2 FILENO_STDERR; File Descriptor de ***stderr*** Standard Error (generalmente pantalla)
- Operador '|': Conecta la salida de un comando con la entrada del siguiente. En otras palabras aplica una redirección del stdout del primer proceso, al stdin del segundo

Conectando procesos

- En un sistema POSIX cuando se crea un proceso se le asignan los siguientes file descriptors predeterminados
  - 0 FILENO_STDIN; File descriptor de ***stdin***, Standard input (teclado)
  - 1 FILENO_STDOUT; File Descriptor de ***stdout***, Standard Output (pantalla)
  - 2 FILENO_STDERR; File Descriptor de ***stderr*** Standard Error (generalmente pantalla)
- Operador '|': Conecta la salida de un comando con la entrada del siguiente. En otras palabras aplica una redirección del stdout del primer proceso, al stdin del segundo

Conectando procesos

- En un sistema POSIX cuando se crea un proceso se le asignan los siguientes file descriptors predeterminados
  - 0 FILENO_STDIN; File descriptor de ***stdin***, Standard input (teclado)
  - 1 FILENO_STDOUT; File Descriptor de ***stdout***, Standard Output (pantalla)
  - 2 FILENO_STDERR; File Descriptor de ***stderr*** Standard Error (generalmente pantalla)
- Operador '|': Conecta la salida de un comando con la entrada del siguiente. En otras palabras aplica una redirección del stdout del primer proceso, al stdin del segundo

Conectando procesos

- En un sistema POSIX cuando se crea un proceso se le asignan los siguientes file descriptors predeterminados
  - 0 FILENO_STDIN; File descriptor de ***stdin***, Standard input (teclado)
  - 1 FILENO_STDOUT; File Descriptor de ***stdout***, Standard Output (pantalla)
  - 2 FILENO_STDERR; File Descriptor de ***stderr*** Standard Error (generalmente pantalla)
- Operador '|': Conecta la salida de un comando con la entrada del siguiente. En otras palabras aplica una redirección del stdout del primer proceso, al stdin del segundo

Ejemplo

Visor de proceso

Nos proponemos escribir un programa llamado *processviewer* que reciba como argumento el nombre de un proceso en particular y permita ver el estado de ese proceso como lo arroja ps -elf.
Nos proponemos utilizar pipes dentro de nuestro código para poder lograrlo

Ejemplo

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int pfds[2];
    if (argc!=2)    {
        fprintf(stderr,"Cantidad de argumentos insuficientes.\n");
        fprintf(stderr,"Uso %s NombreDelProceso.\n",argv[0]);
        exit (1);
    }
    pipe(pfds);
    if (!fork()) {
        dup2(pfds[1],fileno(stdout));/*duplicamos stdout como pfds[1]*/
        close(pfds[0]); /*No necesitamos pfds[0] en este proceso*/
        execlp("ps", "ps", "-elf", NULL);
    } else {
        dup2(pfds[0],fileno(stdin));/*duplicamos stdin como pfds[0]*/
        close(pfds[1]); /* No necesitamos pfds[1] en este proceso*/
        execlp("grep", "grep", argv[1], NULL);
    }
}
```

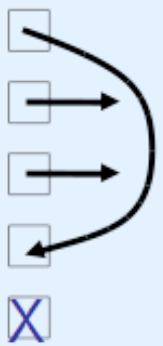
Que paso?

Child (ps - elf)

stream	N°	Estado
stdin	0	→
stdout	1	
stderr	2	→
fd[0]	3	X
fd[1]	4	→

Parent (grep argv[1])

stream	N°	Estado
stdin	0	
stdout	1	→
stderr	2	→
fd[0]	3	
fd[1]	4	X



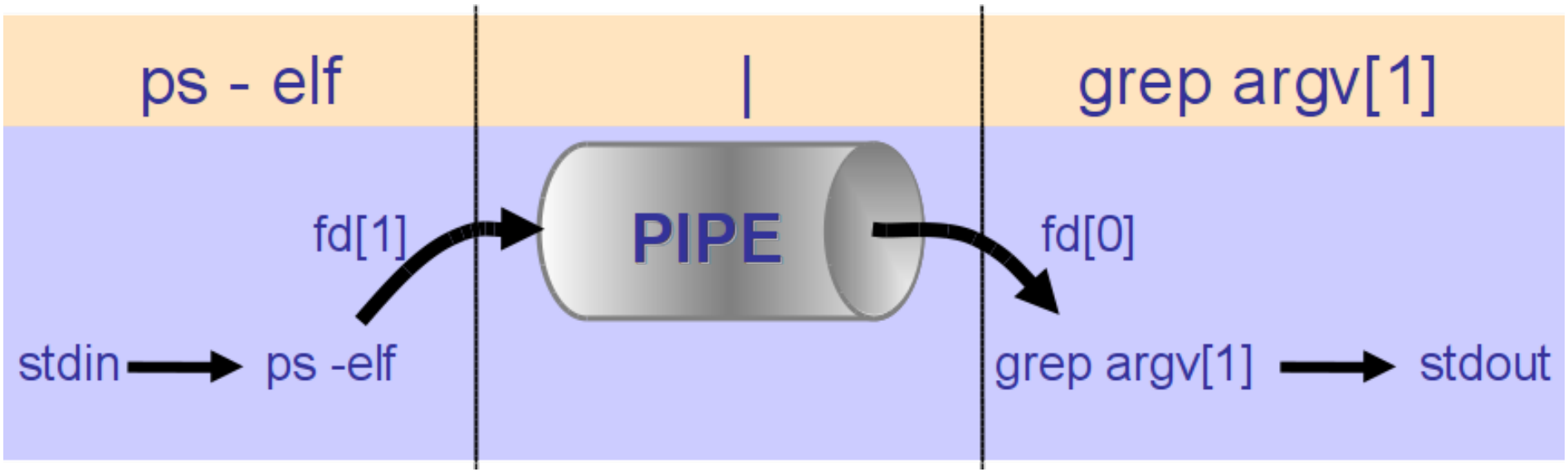
Que paso?

Child (ps - elf)

stream	N°	Estado
stdin	0	→
stdout	1	→
stderr	2	→
fd[0]	3	X
fd[1]	4	→

Parent (grep argv[1])

stream	N°	Estado
stdin	0	→
stdout	1	→
stderr	2	→
fd[0]	3	→
fd[1]	4	X



Contenido

1. Introducción
2. Señales
   - Generalidades
   - Señales mas relevantes
   - System Calls para manejo de señales
   - Señales Hands On.
3. pipes
   - introducción
   - pipe
   - Named FIFOs

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
  - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod -m 660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).
- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open ()**;

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
    - Solo se pueden crear desde la instancia padre de los procesos involucrados
    - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod -m 660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).
- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open ()**;

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
    - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod  m  660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).
- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open ()**;

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
  - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod  m  660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).
- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open ()**;

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
  - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod   m   660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).

- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open** ();

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
  - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod   m   660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).
- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open ()**;

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
  - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod   m   660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).

- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call `open()`;

Generalidades

- Los pipes tienen limitaciones:
  - Solo pueden utilizarse entre procesos padres e hijos
  - Solo se pueden crear desde la instancia padre de los procesos involucrados
  - Se identifican mediante un par de file descriptors.
- Desde el shell podemos crear un Nodo del tipo **named pipe** o **named FIFO**, mediante la siguiente línea:

```
$ mknod   m   660 myfifo p
```

- Los permisos se setean igual que en chmod. En este caso es lectura y escritura para el dueño y su grupo.
- Desde un programa con la system call homónima

```
int mkfifo ( const char *camino, modo_t modo );
```

camino es el path en donde se creará el FIFO, y modo es un OR con los permisos (en sys/stat.h están las macros que definen los tipos).

- El tratamiento es mediante un solo file descriptor. Si el FIFO ya existe se lo puede abrir mediante system call **open ()**;

Hands On!

Práctica Nº1

Escribir un programa que, intente abrir (**`open ()`**) para escritura un Named FIFO en el directorio de trabajo del propio programa, llamado **`Info1_pipe`**.
Si el pipe no está creado no podrá abrirlo, de modo que la condición de error de (**`open ()`**), intentará crear el FIFO (**`mkfifo ()`**).
Si ésta segunda chance no tiene éxito devolverá un mensaje de error que explique lo mas claramente posible el problema. Este mensaje de error debe enviarse por **`stderr`**.
Pistas: Explorar syscall **`strerror()`**

Hands On!

Práctica Nº2

En base al programa de la Práctica Nº1, crear dos instancias: una que abra **`Info1_pipe`** para escritura y otra para lectura solamente. Ambas intentando abrir **`Info1_pipe`** y si no existe crearlo, y manejando los errores de la forma establecida en la Práctica Nº1.
El programa que gana acceso para escritura debe presentar un mensaje por **`stdout`** que invite a escribir una línea de texto. Una vez finalizado el ingreso escribirá en **`Info1_pipe`**. Compilar para obtener un ejecutable llamado **`Info1-FIFO-wr`**
Volverá a buscar otra cadena de texto iterativamente hasta que la cadena ingresada sea una cadena vacía.

Hands On!

Práctica Nº2

El programa que ganó acceso a **`Info1_pipe`** en modo Read Only, presentará un mensaje indicando que empieza a esperar cadenas de texto, e inmediatamente después leerá (**`read ()`**)las cadenas de texto escritas en él, y las presentará en pantalla, precedidas de la hora de lectura de la cadena. Compilar para obtener un ejecutable llamado **`Info1-FIFO-rd`**

Hands On!

Jugando con la Práctica Nº2

1. Conviene tener tres consolas abiertas a lo ancho de la pantalla ocupando un tercio cada una. Ejecutar **`nfo1-FIFO-rd`** y **`Info1-FIFO-wr`** las dos primeras y en la tercera el siguiente comando:

```
while cat /dev/null; do clear ; ps -elf | grep Info1-FIFO; sleep
    1;done
```

   Así tendremos un monitor personalizado del estado de los procesos involucrados que refrescará su estado cada 1 segundo. .

2. Ejecutar en primer lugar **`Info1-FIFO-wr`**.
3. Observar su estado en la consola del monitor. Verificar en la concola de este proceso si presentó el mensaje de invitación a ingresar texto. ¿Lo hizo?.

Hands On!

Jugando con la Práctica Nº2

1. Conviene tener tres consolas abiertas a lo ancho de la pantalla ocupando un tercio cada una. Ejecutar **`nfo1-FIFO-rd`** y **`Info1-FIFO-wr`** las dos primeras y en la tercera el siguiente comando:

```
while cat /dev/null; do clear ; ps -elf | grep Info1-FIFO; sleep
    1;done
```

   Así tendremos un monitor personalizado del estado de los procesos involucrados que refrescará su estado cada 1 segundo. .

2. Ejecutar en primer lugar **`Info1-FIFO-wr`**.
3. Observar su estado en la consola del monitor. Verificar en la concola de este proceso si presentó el mensaje de invitación a ingresar texto. ¿Lo hizo?.

Hands On!

Jugando con la Práctica Nº2

1. Conviene tener tres consolas abiertas a lo ancho de la pantalla ocupando un tercio cada una. Ejecutar **`nfo1-FIFO-rd`** y **`Info1-FIFO-wr`** las dos primeras y en la tercera el siguiente comando:

```
while cat /dev/null; do clear ; ps -elf | grep Info1-FIFO; sleep
    1;done
```

   Así tendremos un monitor personalizado del estado de los procesos involucrados que refrescará su estado cada 1 segundo. .
2. Ejecutar en primer lugar **`Info1-FIFO-wr`**.
3. Observar su estado en la consola del monitor. Verificar en la concola de este proceso si presentó el mensaje de invitación a ingresar texto. ¿Lo hizo?.

Hands On!

Jugando con la Práctica Nº2

4. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿En que punto del código está ocurriendo lo que observamos en el monitor? ¿Porque está en el estado que indica el monitor de procesos?
5. Terminar el proceso **`Info1-FIFO-wr`**
6. Ahora ejecutar en primer lugar **`Info1-FIFO-rd`**.
7. Observar su estado en la consola del monitor. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
8. Ejecutar ahora **`Info1-FIFO-wr`**.

Hands On!

Jugando con la Práctica Nº2

4. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿En que punto del código está ocurriendo lo que observamos en el monitor? ¿Porque está en el estado que indica el monitor de procesos?
5. Terminar el proceso **`Info1-FIFO-wr`**
6. Ahora ejecutar en primer lugar **`Info1-FIFO-rd`**.
7. Observar su estado en la consola del monitor. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
8. Ejecutar ahora **`Info1-FIFO-wr`**.

Hands On!

Jugando con la Práctica Nº2

4. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿En que punto del código está ocurriendo lo que observamos en el monitor? ¿Porque está en el estado que indica el monitor de procesos?
5. Terminar el proceso **`Info1-FIFO-wr`**
6. Ahora ejecutar en primer lugar **`Info1-FIFO-rd`**.
7. Observar su estado en la consola del monitor. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
8. Ejecutar ahora **`Info1-FIFO-wr`**.

Hands On!

Jugando con la Práctica Nº2

4. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿En que punto del código está ocurriendo lo que observamos en el monitor? ¿Porque está en el estado que indica el monitor de procesos?
5. Terminar el proceso **`Info1-FIFO-wr`**
6. Ahora ejecutar en primer lugar **`Info1-FIFO-rd`**.
7. Observar su estado en la consola del monitor. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
8. Ejecutar ahora `Info1-FIFO-wr`.

Hands On!

Jugando con la Práctica Nº2

4. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿En que punto del código está ocurriendo lo que observamos en el monitor? ¿Porque está en el estado que indica el monitor de procesos?
5. Terminar el proceso **`Info1-FIFO-wr`**
6. Ahora ejecutar en primer lugar **`Info1-FIFO-rd`**.
7. Observar su estado en la consola del monitor. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
8. Ejecutar ahora **`Info1-FIFO-wr`**.

Hands On!

Jugando con la Práctica Nº2

9. Observar su estado en la consola del monitor. Verificar en la concola de este proceso si presentó el mensaje de invitación a ingresar texto. ¿Lo hizo?.
10. Observar el código de `Info1-FIFO-wr` en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
11. Usar, probar y experimentar

Hands On!

Jugando con la Práctica Nº2

9. Observar su estado en la consola del monitor. Verificar en la concola de este proceso si presentó el mensaje de invitación a ingresar texto. ¿Lo hizo?.
10. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
11. Usar, probar y experimentar

Hands On!

Jugando con la Práctica Nº2

9. Observar su estado en la consola del monitor. Verificar en la concola de este proceso si presentó el mensaje de invitación a ingresar texto. ¿Lo hizo?.
10. Observar el código de **`Info1-FIFO-wr`** en el editor. ¿Porque está en el estado que indica el monitor de procesos? ¿En que punto del código está ocurriendo lo que observamos en el monitor?
11. Usar, probar y experimentar

Hands On!

Práctica Nº3

Utilizando un par de Named FIFOs escribir un programa que permita efectuar un chat entre dos terminales diferentes conectadas al sistema.