

Sistemas Operativos I

Inter-Process Communication (IPC) - El comienzo

Hay varios mecanismos de comunicación entre procesos

Señales

Podemos indicarle al SO que le envíe una señal a un proceso

Pipes

tuberías que nos sirve para comunicarnos entre los procesos

Sockets

una generalización de la estructura local para poder comunicarnos entre procesos en por la red

Message Queues

una generalización de Pipes donde los mensajes tienen estructura

Shared Memory

memoria compartida entre procesos

Remote Procedure Calls (RPCs)

hacer llamadas a procesos de forma remota

Hay varios mecanismos de comunicación entre procesos que todavía no vimos.

- + Señales: Podemos indicarle al SO que le envíe una señal a un proceso
- + Pipes: tuberías que nos sirve para comunicarnos entre los procesos.
- + Sockets: una generalización de la estructura local para poder comunicarnos entre procesos en por la red.
- + Message Queues : una generalización de Pipes donde los mensajes tiene más estructura (de hecho los mensajes tienen estructura)
- + Shared Memory : memoria compartida entre procesos (no solo entre hilos, librería shm shmget o también mmap para mapear IO/Devices a memoria)
- + Remote Procedure Calls (RPCs): hacer llamadas a procesos de forma remota, hoy en día se ve mucho con servicios en la internet.

Nosotros por el momento nos concentraremos en Señales/Pipes/Sockets.

- + Message Queues son una generalización de Pipes, en particular tienen : Mensajes con estructura, son bidireccionales, los mensajes se pueden recorrer y leer.
- + Shared Memory es memoria compartida entre procesos, ya vimos como era entre hilos y los problemas que traía.

Ambos involucran mayor manejo de C, pero que deberían ser capaces de usarlos por su cuenta

RPCs sobre el final de la materia



Señales

- Las señales son **interrupciones** producidas por un proceso o por el Sistema Operativo (Kernel) y sirven para identificar que algo ha ocurrido.
- Algunas combinaciones de tecla producen señales a los procesos que se están ejecutando en una terminal. Por ejemplo, **CTRL+C** y **CTRL+Z** producen las señales **SIGINT** y **SIGQUIT**, respectivamente.
- Aunque también se pueden producir por **fallos de hardware** (objetivo por el cual fueron creadas).
- Se dice que son **generadas** en el momento que se producen y son consideradas **entregadas** cuando el proceso que la recibe actúa sobre ellas.

Hay 5 tipos de comportamientos por defecto que puede generar una señal.

Term

La acción por defecto es
terminar el proceso

Ign

La acción por defecto es de
ignorar la señal

Core

La acción por defecto es
terminar el proceso y generar
un **core dump**

Stop

La acción por defecto es la de
frenar el proceso

Cont

La acción por defecto es la de
continuar un proceso frenado

Más información en `man 7 signal`



Visitamos `man 7 signal` y programamos un manejador de señales `man 2 signal`

Ejemplo: División por Cero (I)

Hagamos un programa que haga una división por 0.

Cuando intentamos realizar una operación aritmética, el sistema ha generado una excepción de punto flotante con core dump, que es la acción por defecto de la señal.

Un core dump es un archivo que contiene el espacio de direcciones (memoria) de un proceso cuando el proceso finaliza inesperadamente. Los core dump se pueden producir a pedido (por ejemplo, mediante un debugger) o automáticamente al finalizar. Los core dump son generados por el kernel en respuesta al crasheo de programas y pueden pasarse a un programa auxiliar (como systemd-coredump) para su posterior procesamiento.

Utilizar gdb con el core dump:

```
gdb <executable> -c <core-file>
```

Ejemplo: División por Cero (II)

Ahora, modifiquemos el código para manejar esta señal en particular usando la llamada al sistema **signal()**

Ejemplo: generar una señal mediante programación (I)

Utilizar la función **raise()** para generar una señal **SIGSTOP**. Luego, reanudamos el proceso detenido con el comando **fg**.

La señal SIGSTOP también puede ser generada por la combinación de teclas CTRL + Z (Control + Z). Después de emitir esta señal, el programa dejará de ejecutarse. Envía la señal (SIGCONT) para continuar con la ejecución.

Reanudamos el proceso detenido con el comando fg.

Output:
Testing SIGSTOP
[1]+ Stopped ./a.out
./a.out

Ejemplo: generar una señal mediante programación (II)

Mejorar el programa anterior para continuar con la ejecución del proceso detenido emitiendo **SIGCONT** desde otra terminal con el comando **kill -CONT <PID>**

Output:

Testing SIGSTOP

Open Another Terminal and issue following command

kill -SIGCONT <PID> or kill -CONT <PID> or kill -18 <PID>

[1]+ Stopped ./a.out

Received signal SIGCONT

[1]+ Done ./a.out

Ejemplo: Manejar una señal (I)

Genera la señal **SIGTSTP** (terminal stop) cuya acción por defecto es detener la ejecución. Luego, crear un handler que tome la señal, imprima un mensaje y termine.

Ejemplo: Ignorar una señal

Primero registramos la señal **SIGTSTP** para ignorarla a través de SIG_IGN, y luego generamos la señal **SIGTSTP** (terminal stop). Cuando generemos la señal SIGTSTP, está será ignorada.

Hemos visto los casos de realizar una acción predeterminada o manejar la señal.
Ahora, es hora de ignorar la señal.

Output:
Testing SIGTSTP
Signal SIGTSTP is ignored

Ejemplo: Trabajando con múltiples señales

Veamos un programa que:

1. Registra un handler (`handleSignals`) para capturar o manejar las señales `SIGINT` (`CTRL + C`) o `SIGQUIT` (`CTRL + \`)
2. Si el usuario genera la señal `SIGQUIT` (utilizando el comando `kill` o con `CTRL + \`), el handler simplemente imprime el mensaje.
3. Si el usuario genera la señal `SIGINT` (utilizando el comando `kill` o con `CTRL + C`) por primera vez, se captura la señal mostrando un mensaje en pantalla y se restablece la acción de la señal con el comportamiento por defecto (con `SIG_DFL`).
4. Si el usuario genera la señal `SIGINT` por segunda vez, realiza la finalización del programa.

To terminate this program, perform the following:

1. Open another terminal
2. Issue command: `kill 74` or issue `CTRL+C` 2 times (second time it terminates)
^C

You pressed `CTRL+C`

Now reverting `SIGINT` signal to default action

To terminate this program, perform the following:

1. Open another terminal
2. Issue command: `kill 74` or issue `CTRL+C` 2 times (second time it terminates)
^\\You pressed `CTRL+\`

To terminate this program, perform the following:

1. Open another terminal
2. Issue command: `kill 120`
Terminated

Ejemplo: Comunicación entre 2 procesos utilizando señales

El Parent debe enviarle una señal al Child y esperar a que el Child termine. El Child al recibir la señal, debe imprimir en pantalla que la señal fue recibida y terminar.

Probar primero con la señal `SIGHUP`, esto probablemente genere un race condition porque si el Parent ejecuta primero y envía la señal antes que el Child se suscriba entonces se ejecutara el comportamiento por defecto de la señal que es terminal el proceso.

Para solucionar esto puede utilizar la señal `SIGCHLD` (que el comportamiento por defecto es que sea ignorada) o poner un `sleep()` en el parent antes de enviar la señal.

Observar que la comunicación a través de señales es muy limitada ya que las señales NO permiten enviar datos.

sigaction()

La llamada al sistema **sigaction()** es una versión avanzada de **signal()**, permite especificar señales que deben bloquearse mientras está ejecutando el código del handler. También, permite obtener más información de la señal.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Si el argumento **act** no es **NULL**, la nueva acción para la señal **signum** es instalada desde el argumento **act**. Si **oldact** no es **NULL**, la acción anterior es guardada en **oldact**.

```
struct sigaction {
    void (*sa_handler)(int);           // handler por defecto
    void (*sa_sigaction)(int, siginfo_t *, void *); // handler más avanzado, se activa si en sa_flag se especifica SA_SIGINFO
    sigset_t sa_mask;                  // máscara para especificar qué señales deben ser bloqueadas durante la ejecución del handler
    int sa_flags;                       // banderas que modifican el comportamiento de la señal
    ...
};
```

NOTA: Ver el argumento **siginfo_t** de la función miembro **sa_sigaction()**

A faint, stylized illustration of two hands shaking, symbolizing agreement or communication, is centered in the background of the slide.

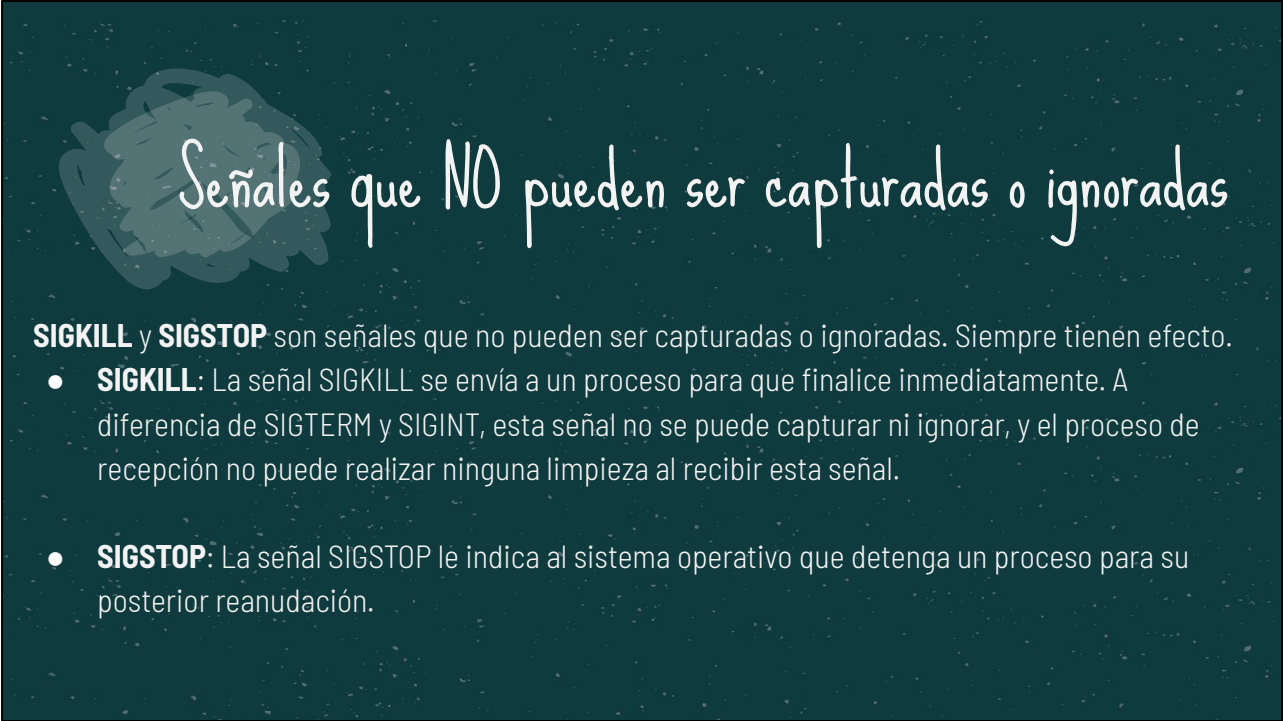
Ejemplo: Comunicación entre 2 procesos utilizando señales (sigaction)

El Parent debe enviarle una señal al Child y esperar a que el Child termine. El Child al recibir la señal, debe imprimir en pantalla que la señal fue recibida y terminar.

Probar primero con la señal `SIGHUP`, esto probablemente genere un race condition porque si el Parent ejecuta primero y envía la señal antes que el Child se suscriba entonces se ejecutara el comportamiento por defecto de la señal que es terminal el proceso.

Para solucionar esto puede utilizar la señal `SIGCHLD` (que el comportamiento por defecto es que sea ignorada) o poner un `sleep()` en el parent antes de enviar la señal.

Observar que la comunicación a través de señales es muy limitada ya que las señales NO permiten enviar datos.



Señales que NO pueden ser capturadas o ignoradas

SIGKILL y **SIGSTOP** son señales que no pueden ser capturadas o ignoradas. Siempre tienen efecto.

- **SIGKILL**: La señal SIGKILL se envía a un proceso para que finalice inmediatamente. A diferencia de SIGTERM y SIGINT, esta señal no se puede capturar ni ignorar, y el proceso de recepción no puede realizar ninguna limpieza al recibir esta señal.
- **SIGSTOP**: La señal SIGSTOP le indica al sistema operativo que detenga un proceso para su posterior reanudación.



Los pipes como indica su nombre introducen un medio de comunicación similar a lo que sería una tubería.

Es un canal *unidireccional* (en una dirección), que permite escribir de un extremo, y leer del otro.

Es básicamente como tener un archivo ya que la creación del pipe nos retornará dos descriptores de archivos, uno para realizar la escritura y otro para realizar la lectura. Veamos un ejemplo.

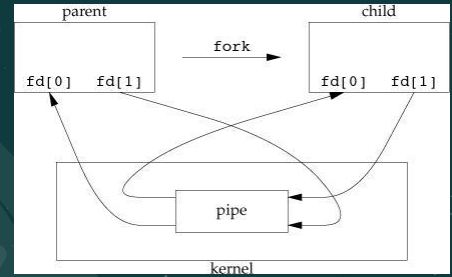
Visitamos el manual de pipe: `man 3 pipe`

Creamos un ejemplo para que se comunique el padre.

Pipes

```
#include <unistd.h>

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



- Un pipe es un canal **unidireccional** (en una dirección), que permite escribir de un extremo, y leer del otro. Aunque se puede acceder a un pipe como a un archivo ordinario, el sistema en realidad lo gestiona como una **cola FIFO**.
- Es importante que el proceso que escribe, cierre su extremo de lectura del pipe y el proceso que lee, cierre su extremo de escritura del pipe.
- Cuando se crea un pipe, se le asigna un tamaño fijo en bytes.
- Cuando un proceso intenta escribir en el pipe, la solicitud de escritura se ejecuta inmediatamente si el pipe no está lleno.
- Si el pipe está lleno, el proceso se bloquea hasta que cambia el estado del pipe.
- Si el pipe está vacío, un proceso de lectura se bloquea; de lo contrario, se ejecuta el proceso de lectura. Solo un proceso puede acceder al pipe a la vez.

Internamente al crear un Pipe, el extremo de lectura se abre usando `O_RDONLY` bandera; el extremo de escritura se abre usando el indicador `O_WRONLY`.

El canal de comunicación proporcionado por un pipe es un stream de bytes: no existe el concepto de límites del mensaje.

If you connect two processes - parent and child - using a pipe, you create the pipe before the fork.

The fork makes the both processes have access to both ends of the pipe. This is not desirable.

The reading side is supposed to learn that the writer has finished if it notices an EOF condition. This can only happen if all writing sides are closed. So it is best if it closes its writing FD ASAP.

The parent closes the end of the pipe it did not use. These closes are necessary to make end-of-file tests work correctly. For example, if a child process intended to read the pipe does not close the write end of the pipe, it will never see the end of file condition on the pipe, because there is one write process potentially active

<https://stackoverflow.com/questions/19265191/why-should-you-close-a-pipe-in-linux>

Ejemplo: Comunicación entre Procesos usando Pipes

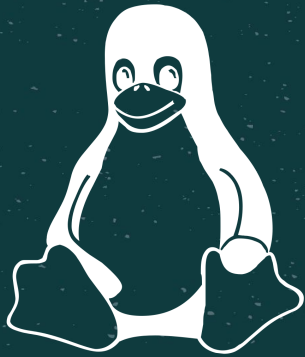
Crear dos procesos que se comuniquen a través de un pipe. Después del **fork(2)** cada proceso cierra los File Descriptors que no necesita. El Child hereda un conjunto duplicado de descriptores de archivo que se refieren a la misma canalización.

Luego, el Parent escribe una cadena de caracteres en el pipe, y el Child lee esta cadena de a un byte a la vez e imprime lo que leyó en la salida estándar.



Limitaciones de los Pipes:

- Como canal de comunicación, un pipe opera en **una sola dirección**.
- Los pipes **no soportan broadcast**, es decir, no puede enviar mensajes a múltiples procesos al mismo tiempo.
- Se requiere algo de **plomería** (cierre de extremos del pipe) para crear un pipe correctamente dirigido.



Shared-Memory



Shared-Memory

La memoria compartida en POSIX se organiza mediante archivos mapeados en memoria, que asocian la región de la memoria compartida con un archivo. Un proceso puede crear un objeto de memoria compartida usando la llamada al sistema `shm_open()`.

```
#include <sys/mman.h>
#include <sys/stat.h>      /* For mode constants */
#include <fcntl.h>         /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

La llamada al sistema `shm_open()` crea y abre un objeto de memoria compartida POSIX nuevo o abre uno existente. Un objeto de memoria compartida POSIX es, de hecho, un identificador que puede ser utilizado por procesos no relacionados que mapeen (`mmap(2)`) a una misma Región de la memoria compartida. La función `shm_unlink()` realiza la operación inversa, eliminando un objeto creado previamente por `shm_open()`. La operación de `shm_open()` es análoga a la de `open(2)`.

Los parámetros de `shm_open()` son:

- `name` especifica el objeto de memoria compartida
- `oflag` es una bit mask que puede tomar los flags: `O_RDONLY`, `O_RDWR`, `O_CREAT`, entre otras.
- `mode_t` especifica los permisos para el objeto memoria (Ejemplo: 0666)

`shm_open()` retorna un file descriptor.

Descripción general de la memoria compartida POSIX se encuentra en `shm_overview(7)`.

Shared-Memory

La llamada al sistema **mmap()** crea una nueva asignación en el espacio de direcciones virtuales del proceso que la llama. La dirección inicial para el nuevo mapeo se especifica en el argumento **addr**. El argumento **length** especifica la longitud del mapeo (que debe ser mayor que 0).

En caso de éxito, `mmap()` devuelve un puntero al área mapeada.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
off_t offset);

int munmap(void *addr, size_t length); //unmap memory
```

La llamada al sistema `mmap()` crea una nueva asignación en el espacio de direcciones virtuales del proceso que la llama. La dirección inicial para el nuevo mapeo se especifica en el argumento `addr`. El argumento `length` especifica la longitud del mapeo (que debe ser mayor que 0). En caso de éxito, `mmap()` devuelve un puntero al área mapeada.

La llamada al sistema `munmap()` elimina las asignaciones para el rango de direcciones especificado y hace que otras referencias a direcciones dentro del rango generen referencias de memoria no válidas. La región también se desasigna automáticamente cuando se finaliza el proceso. Por otro lado, cerrar el descriptor de archivo no desasigna la región.

Ejemplo: Shared-Memory



Compartir memoria entre un padre y un hijo.