

Sistemas Operativos 2010-1

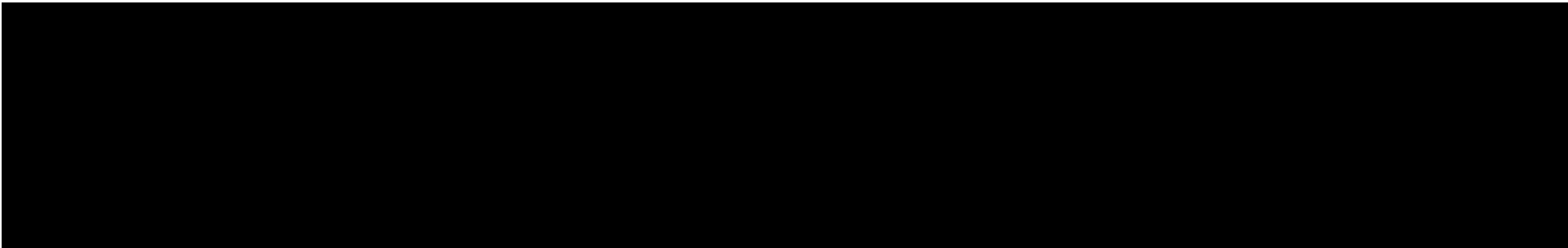
Señales

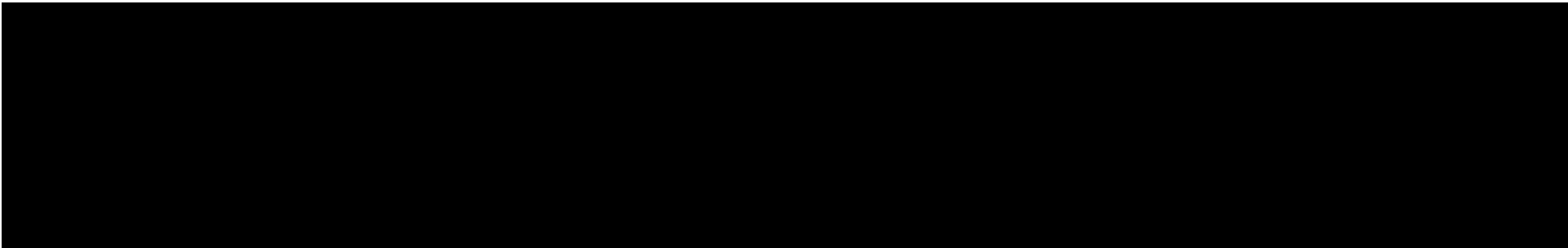
Señales

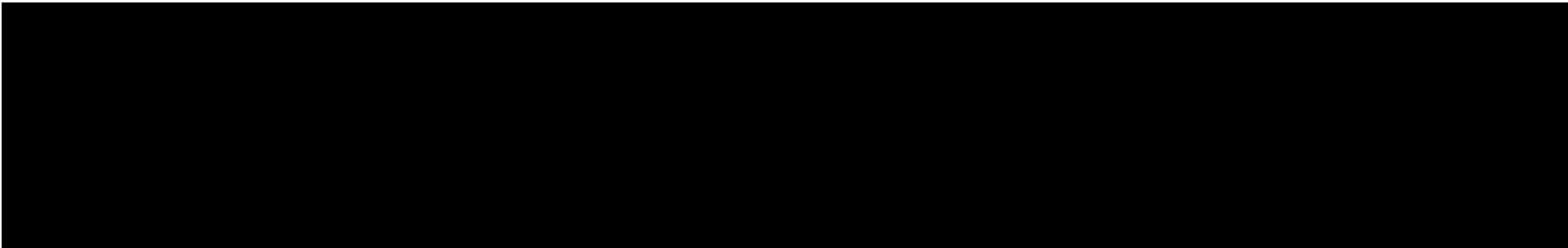
- Podemos considerar a las señales como la forma más primitiva de comunicación entre procesos. El sistema utiliza señales para informar a un determinado proceso sobre alguna condición, realizar esperas entre procesos, etc. Sin embargo la señal en sí no es portadora de datos.

Señales

- Permite a un proceso enterarse de una determinada condición, pero sin poder transmitir cantidades grandes de información entre ambos procesos.
- Por lo tanto, además de las señales, es preciso disponer de mecanismos que permitan intercambiar datos entre los procesos.

- 
- Cuando implementamos un programa, línea a línea vamos definiendo el curso de ejecución del mismo, con condicionales, bucles, etc. Sin embargo hay ocasiones en las que nos interesaría contemplar sucesos asíncronos, es decir, que pueden suceder en cualquier momento, no cuando nosotros los comprobemos.

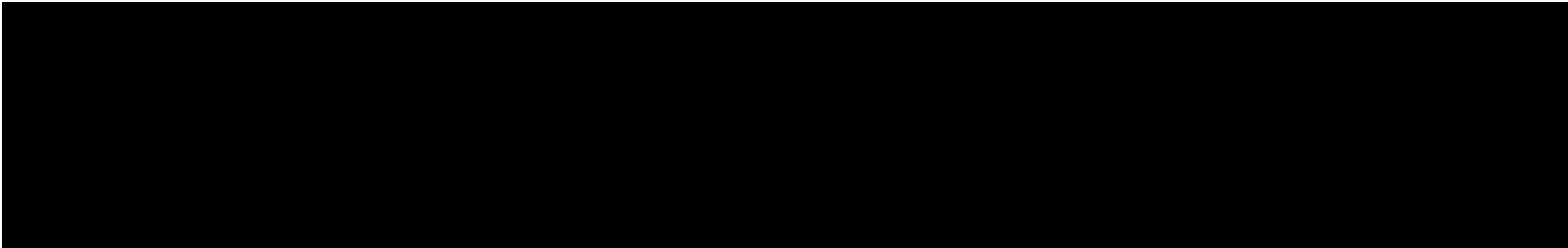
- 
- La manera más sencilla de contemplar esto es mediante el uso de señales
 - La pérdida de la conexión con el terminal, una interrupción de teclado o una condición de error como la de un proceso intentando acceder a una dirección inexistente de memoria podrían desencadenar que un proceso recibiese una señal. Una vez recibida, es tarea del proceso atrapar o capturarla y tratarla. Si una señal no se captura, el proceso muere.

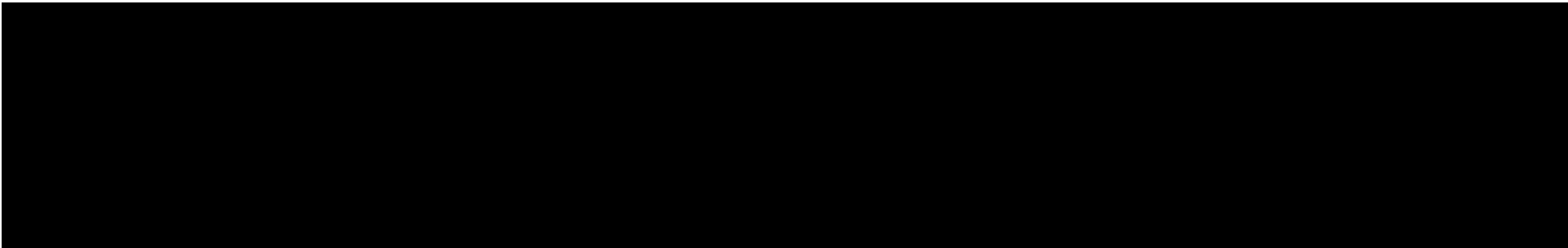
- 
- En función del sistema en el que nos encontremos, bien el núcleo del Sistema Operativo, bien los procesos normales pueden elegir entre un conjunto de señales predefinidas, siempre que tengan los privilegios necesarios. Es decir, no todos los procesos se pueden comunicar con procesos privilegiados mediante señales. Esto provocaría que un usuario sin privilegios en el sistema sería capaz de matar un proceso importante mandando una señal SIGKILL, por ejemplo.

:)- kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

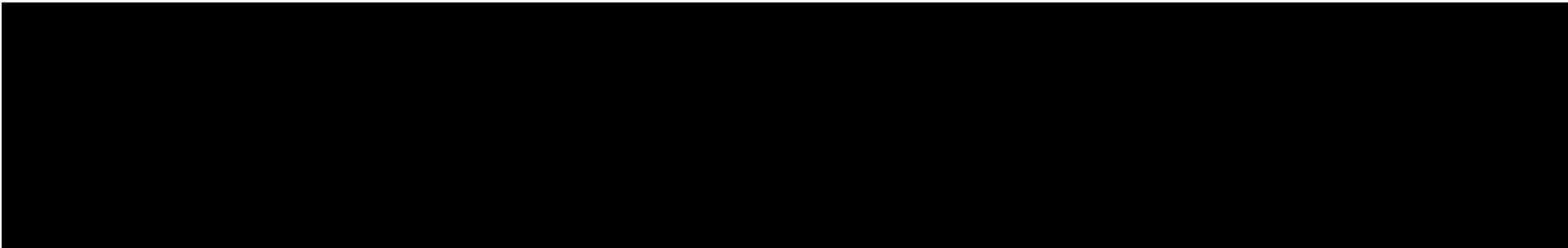
- Todas las señales pueden ser ignoradas o bloqueadas, a excepción de SIGSTOP y SIGKILL, que son imposibles de ignorar. En función del tratamiento que especifiquemos para cada señal realizaremos la tarea predeterminada, una propia definida por el programador, o la ignoraremos (siempre que sea posible). Es decir, nuestro proceso modifica el tratamiento por defecto de la señal realizando llamadas al sistema que alteran la sigaction de la señal apropiada.

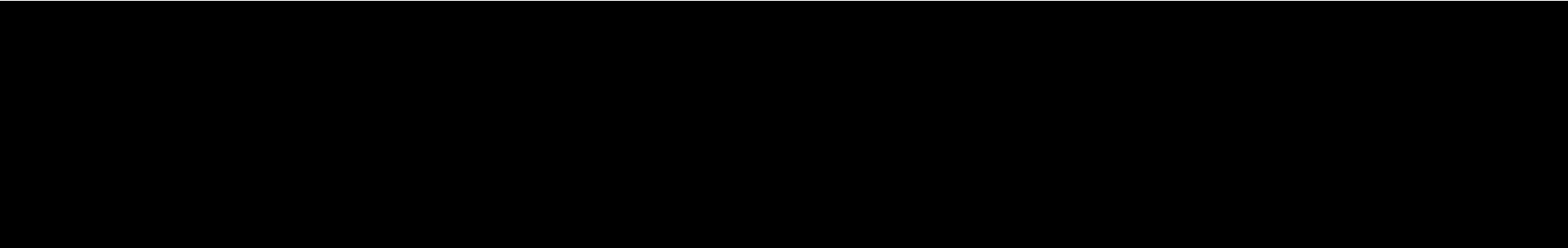
- 
- Una limitación importante de las señales es que no tienen prioridades relativas, es decir, si dos señales llegan al mismo tiempo a un proceso puede que sean tratadas en cualquier orden, no podemos asegurar la prioridad de una en concreto. Otra limitación es la imposibilidad de tratar múltiples señales iguales: si nos llegan 14 señales SIGCONT a la vez, por ejemplo, el proceso funcionará como si hubiera recibido sólo una.

- 
- Cuando queremos que un proceso espere a que le llegue una señal, usaremos la función `pause()`. Esta función provoca que el proceso (o thread) en cuestión “duerma” hasta que le llegue una señal. Para capturar esa señal, el proceso deberá haber establecido un tratamiento de la misma con la función `signal()`.

- `int pause(void);`
- no recibe ningún parámetro y retorna `-1` cuando la llamada a la función que captura la señal ha terminado.
- La función `signal()` recibe dos parámetros, el número de señal que queremos capturar, y un puntero a una función que se encargará de tratar la señal especificada.

```
#include <signal.h>
#include <unistd.h>
void trapper(int);
int main(int argc, char *argv[]) {
    int i;
    for(i=1;i<=64;i++)
        signal(i, trapper);
    printf("Identificativo de proceso: %d\n", getpid() );
    pause();
    printf("Continuando...\n");
    return 0;
}
void trapper(int sig) {
    printf("Recibida la señal: %d\n", sig);
}
```

- 
- Si queremos enviar una señal desde la línea de comandos, utilizaremos el comando “kill”.
 - La función de C que hace la misma labor se llama, originalmente, kill(). Esta función puede enviar cualquier señal a cualquier proceso, siempre y cuando tengamos los permisos adecuados (las credenciales de cada proceso, explicadas anteriormente, entran ahora en juego (uid, euid, etc.)).



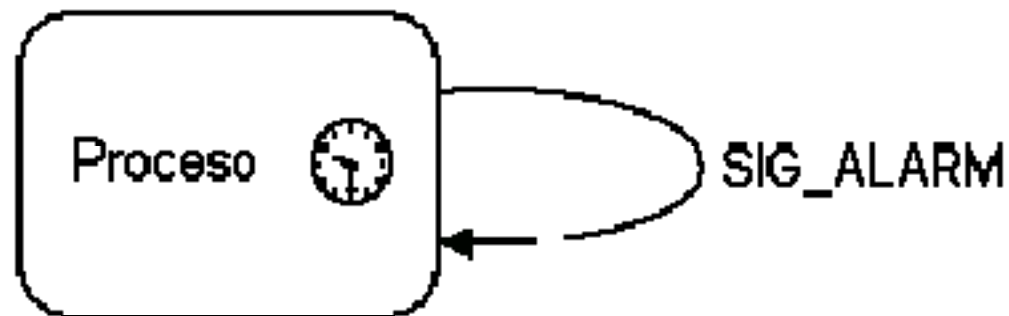
No tiene mucha complicación, recibe dos parámetros, el PID del proceso que recibirá la señal, y la señal.

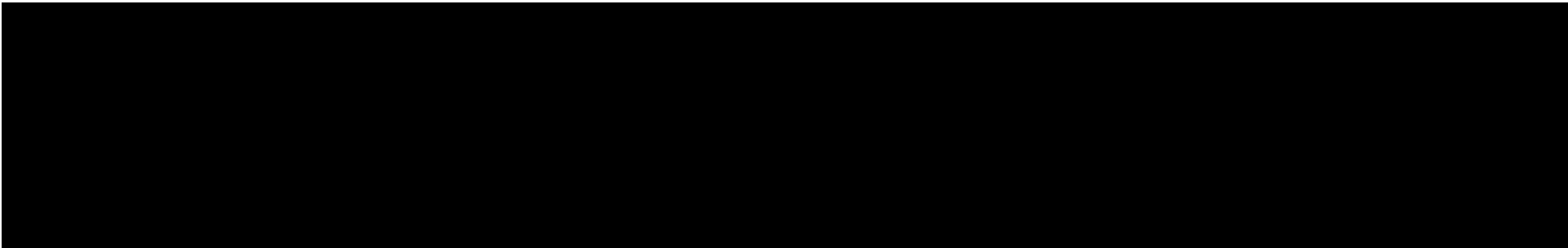
```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    pid_t pid;
    int sig;
    if(argc==3) {
        pid=(pid_t)atoi(argv[1]);
        sig=atoi(argv[2]);
        kill(pid, sig);
    } else {
        printf("%s: %s pid signal\n", argv[0], argv[0]);
        return -1;
    }
    return 0;
}
```

- Una utilización bastante potente de las señales es el uso de SIGALARM para crear temporizadores en nuestros programas. Con la función alarm() lo que conseguimos es que nuestro proceso se envíe a sí mismo una señal SIGALARM en el número de segundos que especifiquemos. El prototipo de alarm() es el siguiente:

```
unsigned int alarm(unsigned int seconds);
```

- En su único parámetro indicamos el número de segundos que queremos esperar desde la llamada a `alarm()` para recibir la señal `SIGALARM`.



- 
- La llamada a la función `alarm()` generará una señal `SIG_ALARM` hacia el mismo proceso que la invoca.
 - El valor devuelto es el número de segundos que quedaban en la anterior alarma antes de fijar esta nueva alarma. Esto es importante: sólo disponemos de un temporizador para usar con `alarm()`, por lo que si llamamos seguidamente otra vez a `alarm()`, la alarma inicial será sobrescrita por la nueva.

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

```
void trapper(int);
```

```
int main(int argc, char *argv[])
{
    int i;
```

```
    signal(14, trapper);
```

```
    printf("PID de proceso:
%d\n", getpid() );
```

```
    alarm(5);
    pause();
    alarm(3);
    pause();
```

```
for(;;)
{
```

```
    alarm(1);
    pause();
```

```
}
```

```
    return 0;
```

```
}
```

```
void trapper(int sig)
```

```
{
```

```
    printf("RIIIIIIIING!\n");
```

```
}
```

```

#include <stdlib.h>
#include <signal.h>

int numcortes=0;          /* Contador de CTRL-C */
int bucle=1;              /* Controlador de salida del bucle de espera */

void alarma ();           /* Captura la se al de alarma SIGALRM */
void cortar ();           /* Captura la se al de interrupci n SIGINT */

int main ()
{
    signal (SIGINT, cortar);
    signal (SIGALRM, alarma);
    printf ("Ejemplo de signal.\n");
    printf ("Pulsa varias veces CTRL-C durante 15 segundos.\n");
    alarm (15);
    while (bucle);
    signal (SIGINT, SIG_IGN);
    printf ("Has intentado cortar %d veces.\n", numcortes);
    printf ("Fin de ejercicio.\n");
    exit (0);
}

void alarma ()
{
    signal (SIGALRM, SIG_IGN);
    bucle=0;              /* Salir del bucle */
    printf (" ;Alarma!\n");
}

void cortar ()
{
    signal (SIGINT, SIG_IGN);
    printf ("Has pulsado CTRL-C\n");
    numcortes++;
    signal (SIGINT, cortar);
}

```

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void trapper(int sig)
{
    signal(sig, trapper);
    printf("SIGUSR1\n");
}

int main(int argc, char *argv[])
{
    pid_t padre, hijo;

    padre = getpid();

    signal( SIGUSR1, trapper );
}

```

```

if ( (hijo=fork()) == 0 )
{ /* hijo */
    sleep(1);
    kill(padre, SIGUSR1);
    sleep(1);
    kill(padre, SIGUSR1);
    sleep(1);
    kill( padre, SIGUSR1);
    sleep(1);
    kill(padre, SIGKILL);
    exit(0);
}
else
{ /* padre */
    for (;;)
}

return 0;

```