Señales de Unix

Esteban Ruiz
eruiz0@fceia.unr.edu.ar

Sistemas Operativos II - 2022
Departamento de Ciencias de la Computación
FCEIA-UNR

Índice

1. Conceptos generales

1.1. Introducción

Las *señales* son una forma limitada de comunicación entre procesos (IPC, Inter Process Communication) usada en Unix y similares y otros sistemas POSIX. Una señal es una notificación asíncrona enviada a un proceso o a un hilo específico dentro del mismo proceso para notificarlo de un evento que ocurrió. Las señales han estado desde el Unix de Bell Labs en los '70 y fueron luego especificadas por el estándar POSIX.

Cuando se envía una señal, el sistema operativo interrumpe[1] el flujo de ejecución normal del proceso destinatario para enviar la señal. La ejecución puede ser interrumpida durante cualquier instrucción no atómica. Si el proceso registró previamente un manejador[2] (*signal handler*) entonces se ejecuta esa rutina. En caso contrario se ejecuta el manejador por defecto.

Las señales son en general un mecanismo liviano computacionalmente y en consumo de memoria y por tanto pueden ser útiles en sistemas embebidos.

Como ya dijimos, una señal está asociada a un evento asíncrono. Los eventos pueden ser de muy diferente naturaleza y para identificar el tipo de evento cada señal incluye un código de evento, por ejemplo: `SIGSEGV` indica que se produjo un fallo por violación de segmento. Además, los procesos pueden enviar cualquier señal (sin necesidad de que exista realmente el evento usualmente asociado a esa señal).

Los primeros UNIX utilizaron un mecanismo muy básico de señales y luego el UNIX de BSD incorporó un mecanismo más sofisticado que permite proveer más información sobre los eventos y tener un control más minucioso de los detalles del mecanismo. Este mecanismo fue básicamente el que estandarizó POSIX.

2. Mecanismo de señales del UNIX original

2.1. Enviando señales

La llamada `kill` envía la señal especificada a un proceso (siempre que cuente con los permisos necesarios):

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

int kill(pid_t pid, int sig); //enviar al proceso pid la señal sig
                              //retorna 0 o -1 (indicando que hubo un error
                              //cuyo código queda en la variable errno)
```

La llamada `getpid` retorna el PID (identificador de proceso) del proceso actual:

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

pid_t getpid(void); //retorna el process ID (pid) del proceso llamante
```

[1] En un sentido muy parecido al de las interrupciones de hardware

[2] Esto es una mala traducción -de hecho la palabra no existe- del término en inglés "handler", qué significa despachador, responsable, etc..

La función de librería `raise` envía una señal al proceso actual (es decir: un proceso puede usar `raise` para auto-enviarse una señal):

```
#include <signal.h>

int raise(int sig); //auto-enviar la señal sig
```

Ejemplo de uso:

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

main(){
        kill(getpid(),17);
        raise(SIGCHLD);
        return 0;
}
```

En este caso, en la llamada `kill` el proceso obtiene su propio PID y envía la señal 17 a ese número de proceso. Esto es equivalente a la siguiente línea, que utiliza la función de librería `raise`. En `raise` se usa `SIGCHLD`, que tiene el código 17 (es mucho más portable especificar `SIGCHLD` en lugar del código 17).

Desde el shell se puede utilizar el comando `kill` para enviar señales. Si se omite el código de la señal, kill envía SIGTERM:

```
ps [options] #reporta los procesos actuales
kill [ -signal | -s signal ] pid ... # enviar signal a los pids
```

Ejemplos de uso:

```
$ ps
  PID TTY          TIME CMD
23170 pts/3    00:00:00 bash
25888 pts/3    00:00:00 ps
$ kill -SIGCHLD 25888
-bash: kill: (25888) - No existe el proceso
$ kill -SIGCHLD 23170
$ kill 23170
$ kill -SIGKILL 23170 # termina sí o sí el proceso
$ kill -9 23170       # IDEM
```

Las excepciones como división por cero o violación de segmento generarán señales (`SIGFPE` y `SIGSEGV` respectivamente), ambas causan por defecto un volcado de la memoria -core dump- y la terminación del programa.

El núcleo puede generar señales para notificar eventos, como `SIGPIPE` que se genera cuando un proceso escribe en un pipe[3] que ha sido cerrado por el

[3] tubería

lector (que por defecto causa la terminación del proceso, útil para hacer pipelines de shell).

Algunas combinaciones de teclas en la terminal asociada al proceso causan envío de señales:

- `Ctrl-C` envía `SIGINT` (interrupt, que por defecto termina el proceso).
- `Ctrl-Z` envía `SIGTSTP` (suspende la ejecución).
- `Ctrl-\` envía `SIGQUIT` (termina y vuelca la memoria).
- `Ctrl-T` envía (en algunos UNIXES) `SIGINFO` (muestra información sobre el proceso).

2.2. Manejando señales

Los manejadores se instalan con la llamada `signal()`. Si no hay un manejador instalado para una señal en particular, entonces se usa el manejador por defecto si llega esa señal (que según la señal puede interrumpir el proceso, sencillamente ignorar la señal, provocar un volcado, etc...). Si hay un manejador, la señal se intercepta y se invoca al manejador configurado con `signal`. El proceso puede especificar dos comportamientos sin necesidad de crear una nueva función manejadora: ignorar la señal (`SIG_IGN`) y usar el manejador por defecto (`SIG_DFL`). Hay dos señales que no se pueden interceptar: `SIGKILL` y `SIGSTOP`.

Así, `signal` tiene la siguiente declaración:

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Es decir: toma el número de señal (`signum`) que se quiere configurar y el nuevo handler (que es una función) que se invocará cuando llegue esa señal. Retorna el handler anterior (esto permite restaurarlo posteriormente).

Anteriormente `signal` estaba declarada en una forma equivalente pero más difícil de leer:

```
void (*signal (int sig, void (*func) (int))) (int)
```

2.3. Ejemplo: interceptando Control-C

En este ejemplo se demuestra cómo interceptar cuatro veces la combinación de teclas `Control-C` (que envía la señal `SIGINT`) y a la quinta vez finalizar el programa. La finalización se produce luego de restaurarse el handler por defecto (`SIG_DFL`), que para el caso de `SIGINT` termina el proceso, y recibirse nuevamente la misma señal.

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

```
void handler(int sig) {
        static int cuenta=0;
        cuenta++;
        printf("\nOuch: %d\n",cuenta) ;
        if (cuenta==4) {
                signal(SIGINT, SIG_DFL);
        }
}

main(){
        signal(SIGINT, handler);
        for(;;);
        return 0;
}
```

2.4. Ejemplo 2: especificando alarmas

Este ejemplo utiliza `SIGALRM`: esta señal llega luego de que expira un temporizador programado con la función `alarm`:

```
#include <unistd.h>

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

El programa configura un handler que termina el programa luego de que expira la alarma. La alarma se setea para que la señal llegue luego de 5 segundos.

```
#include <signal.h> // for signal et al.
#include <stdio.h>  // for printf
#include <unistd.h> // for alarm
#include <stdlib.h> // for exit

void handler(int sig) {
        printf("Alarmaaa\n");
        exit(1);
}

main(){
        signal(SIGALRM, handler);
        alarm(5);
        for(;;);
        return 0;
}
```

2.5. Problemas

El manejo de señales en UNIX tradicional es susceptible de problemas de concurrencia. Como son asíncronas, otra señal (incluso del mismo tipo) puede llegar cuando se está ejecutando el handler. Se puede usar `sigprocmask` para controlar un poco esto.

Además pueden causar la interrupción de una llamada a sistema, dejando a la aplicación el manejo de ese error.

Se deben evitar los efectos secundarios: modificar `errno`, cambios en la máscaras de señales o su disposición y otros atributos globales. El uso de funciones no re-entrantes como `malloc` o `printf` en un handler también es inseguro. Hay formas complicadas de encolar las señales para procesarlas después y evitar algunos de los problemas relacionados.

2.6. Relación con excepciones de hardware

La ejecución de un proceso puede resultar en una excepción de hardware, por ejemplo, si un proceso intenta dividir por cero o incurre en un fallo de TLB.

Esto generalmente produce la ejecución de una manejador de excepciones en el núcleo. En algunos casos (p. ej.: un fallo de TLB o un fallo de paginación causado porque la página no está en memoria), el núcleo tiene información suficiente para resolver el evento y el proceso puede continuar su ejecución.

Otras excepciones, por ejemplo la división por cero, no pueden resolverse en el núcleo y entonces se utiliza el mecanismo de señales para que el proceso maneje la excepción (en este caso, el proceso recibirá una señal `SIGFPE`). En el caso de intentar acceder a una dirección no permitida (p. ej.: referenciar un puntero NULL o fuera de una región reservada con `malloc`), el proceso recibirá una señal indicativa de la violación de la segmentación (`SIGSEGV`).

2.7. Listado de señales

El estándar POSIX documenta las siguientes señales. Los nombres consisten del prefijo "`SIG`" y varios caracteres para identificar la señal. Cada macro se expande a un entero (el entero puede variar entre distintas plataformas).

- SIGABRT: indica a un proceso que debería abortar (terminar). La función de librería `abort` genera esta señal.
- SIGALRM, SIGVTALRM y SIGPROF: se envían cuando se llegó a un límite de tiempo especificado previamente con alguna función de alarma (p. ej.: `alarm` provoca SIGLARM).
- SIGBUS: error de bus (p. ej.: acceso a memoria no alineado o dirección física inexistente).
- SIGCHLD: se envía a un proceso padre cuando algún proceso hijo termina (también cuando se interrumpe o vuelve a continuar).
- SIGCONT: le indica al sistema operativo que un proceso puede continuar (usada cuando un proceso ha sido pausado con SIGSTOP o SIGTSTP.
- SIGFPE: floating point exception (aunque incluye también la división por cero entre enteros).
- SIGHUP: indica que su terminal controladora se cerró (hangup: colgó, esto viene de cuando las terminales estaban conectadas por una línea serie).

- SIGILL: instrucción ilegal.
- SIGINT: interrumpir. Típicamente asociada a la combinación de teclas `Control-C`.
- SIGKILL: terminar inmediatamente. No puede ser capturada ni ignorada.
- SIGPIPE: se intentó escribir en un pipe (tubería) pero el otro extremo no está conectado.
- SIGQUIT: salir (y realizar un volcado de memoria)
- SIGSEGV: segmentation violation: violación de segmento o referencia inválida a memoria virtual (típicamente se referenció un puntero que estaba apuntando a NULL o a una dirección que no es válida para el proceso).
- SIGSTOP: detenerse (para continuar posteriormente, el proceso NO se termina).
- SIGTERM: terminar. Similar a SIGINT, enviada por el sistema por ejemplo al apagarse para permitir a los procesos guardar convenientemente el estado.
- SIGTSTP: similar a SIGSTOP y típicamente asociada a `Control-Z`, pero a diferencia de SIGSTOP se puede ignorar.
- SIGTTIN y SIGTTOU: indican intento de leer (IN) o escribir (OUT) en la terminal cuando el proceso está en background.
- SIGUSR1 y SIGUSR2: para condiciones definidas por el usuario (o programa).
- SIGPOLL: evento asíncrono de entrada/salida.
- SIGSYS: argumento incorrecto pasado a una llamada a sistema.
- SIGTRAP: trap, utilizada por debuggers.
- SIGURG: llegaron datos urgentes (o fuera de banda) por un socket.
- SIGXCPU: se excedió el tiempo de CPU asignado al proceso (permite guardar resultados intermedios antes de que el sistema envíe SIGKILL).
- SIGXFSZ: se excedió el tamaño máximo de archivo.
- SIGRTMIN a SIGRTMAX: rango de señales para uso del usuario (real-time)

El cuadro 1 resume brevemente el significado de cada señal y la acción por defecto asociada (ver cuadro 2).

Señal	Acción	Evento
SIGABRT	A	Abortar
SIGALRM	T	Alarma
SIGBUS	A	Acceso a memoria no definida
SIGCHLD	I	Un proceso hijo terminó o se detuvo
SIGCONT	C	Continuar ejecución
SIGFPE	A	Operación aritmética errónea
SIGHUP	T	Colgó (la línea serie)
SIGILL	A	Instrucción ilegal
SIGINT	T	Interrupción desde la terminal
SIGKILL	T	Kill
SIGPIPE	T	Intento de escribir a pipe sin lector
SIGQUIT	A	Salir (desde la terminal)
SIGSEGV	A	Referencia a memoria inválida
SIGSTOP	S	Stop
SIGTERM	T	Terminar
SIGTSTP	S	Stop (desde la terminal)
SIGTTIN	S	Intento de lectura en background
SIGTTOU	S	Intento de escritura en background
SIGUSR1	T	Para usuario
SIGUSR2	T	Para usuario
SIGPOLL	T	Evento sondeable (pollable)
SIGPROF	T	Timer de rendimiento expirado
SIGSYS	A	Llamada a sistema inválida
SIGTRAP	A	Trace/breakpoint trap (debuggers)
SIGURG	I	Datos urgentes (sockets)
SIGVTALRM	T	Timer virtual expirado
SIGXCPU	A	Exceso de tiempo de CPU
SIGXFSZ	A	Exceso de tamaño de archivo

Cuadro 1: Señales y acción por defecto

Acción	Significado
T	Terminar
A	Terminación anormal (puede generar volcado de memoria)
I	Ignorar
S	Stop (detener)
C	Continuar (si estaba detenido)

Cuadro 2: Acciones por defecto

2.8. Relación con los estados de un proceso

La figura 1 ilustra la relación entre las señales más importantes y el ciclo de vida de un proceso.

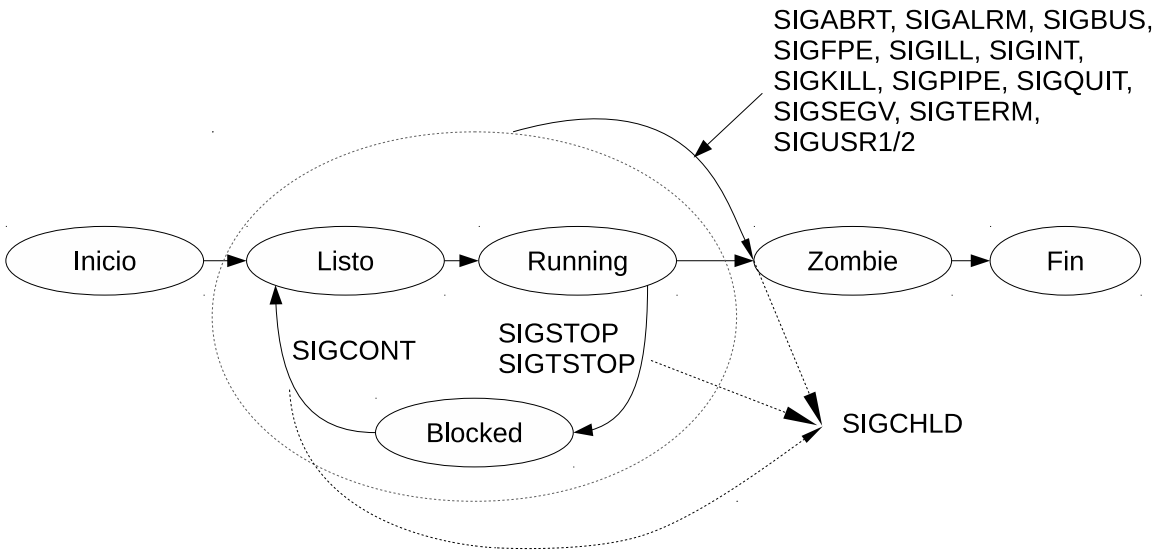

Figura 1: Relación entre algunas señales y los estados de un proceso

3. Mecanismo de señales de BSD

El mecanismo de señales implementado por BSD y luego estandarizado por POSIX añade más funcionalidad y permite más control sobre los problemas mencionados anteriormente. Así, se puede especificar qué pasará cuando lleguen concurrentemente varias señales, stacks separados para los manejadores, máscaras, comportamiento durante llamadas al sistema, etc... Y también obtener más información sobre los eventos, como ser: qué proceso lo originó, qué dirección provocó un fallo, etc...

Se sigue manteniendo el esquema anterior en el que cada señal tiene un código numérico asociado, pero ahora un manejador puede recibir mucha más información:

```
void (*sa_sigaction)(int, siginfo_t *, void *);
```

En donde el primer argumento de tipo `int` es el código de la señal, el segundo (de tipo `siginfo_t`) contiene información más detallada sobre el evento y el tercer argumento es en algunos UNIXes (incluyendo Linux) un puntero a una estructura `ucontext_t`, que incluye información del contexto (muy dependiente del sistema operativo).

3.1. Información de eventos

La estructura `siginfo_t` contiene la siguiente información:

```
siginfo_t {
    int      si_signo;    /* Signal number */
    int      si_errno;    /* An errno value */
    int      si_code;     /* Signal code */
    int      si_trapno;   /* Trap number that caused
```

```
                                    hardware-generated signal
                                    (unused on most architectures) */
            pid_t    si_pid;      /* Sending process ID */
            uid_t    si_uid;      /* Real user ID of sending process */
            int      si_status;   /* Exit value or signal */
            clock_t  si_utime;    /* User time consumed */
            clock_t  si_stime;    /* System time consumed */
            sigval_t si_value;    /* Signal value */
            int      si_int;      /* POSIX.1b signal */
            void    *si_ptr;      /* POSIX.1b signal */
            int      si_overrun;  /* Timer overrun count; POSIX.1b timers */
            int      si_timerid;  /* Timer ID; POSIX.1b timers */
            void    *si_addr;     /* Memory location which caused fault */
            int      si_band;     /* Band event */
            int      si_fd;       /* File descriptor */
        }
```

Notar que el sistema operativo rellenará los campos relevantes de acuerdo al tipo de señal (p. ej.: si hay un fallo de segmentación seguramente `si_addr` tendrá la ubicación de memoria que provocó el fallo).

3.2. Especificando manejadores

Especificar un handler ahora no es tan sencillo (muchos UNIXes soportan simultáneamente la vieja función `signal`):

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Nuevamente `signum` será el número de señal a manejar, pero ahora hay dos acciones (`sigactions`) involucradas. Las acciones no sólo especifican a qué función se llamará cuando ocurra la señal, también especifican banderas para indicar cuestiones de concurrencia y otros detalles que veremos luego.

- `act` Nueva acción. Si se deja en NULL, no cambiará el comportamiento de esa señal.
- `oldact` Acción anterior. Si se deja en NULL ya no se podrá recuperar la acción anterior.

Notar que no se debe dejar a ambos punteros en NULL (en ese caso `sigaction` sencillamente no tendría nada que hacer). Ahora, a diferencia de `signal`, `sigaction` devuelve 0 si todo anduvo bien.

3.3. Especificando acciones

La estructura `sigaction` es la siguiente:

```
struct sigaction {
            void     (*sa_handler)(int);
            void     (*sa_sigaction)(int, siginfo_t *, void *);
            sigset_t   sa_mask;
```

```
        int        sa_flags;
        void     (*sa_restorer)(void); //Obsoleto, no usar.
    };
```

Los dos primeros miembros son punteros a funciones: `sa_handler` permite especificar un handler al estilo de UNIX tradicional (tal como se haría con `signal`) y `sa_sigaction` permite especificar una función que podrá tener mucha más información sobre el evento.

Sólo se debe especificar uno de estos punteros: `sigaction` sabrá que debe usar el handler especificado en `sa_sigaction` si en las banderas (`sa_flags`) se incluye la bandera SA_SIGINFO. Si no se especifica `SA_SIGINFO` entonces `sigaction` instalará el handler especificado en `sa_handler`.

`sa_mask` permite especificar un conjunto de señales que se bloquearán durante la ejecución del manejador. Además, salvo que en `sa_flags` se especifique `SA_NODEFER`, la señal que se está manejando también se bloqueará si llega nuevamente durante la ejecución del manejador.

El resto de las banderas son, en forma *muy* resumida:

- `SA_NOCLDSTOP`: sólo tiene sentido si se está estableciendo un handler para SIGCHLD, no notificar si el proceso hijo se detiene (stop) o continua (por SIGCONT).
- `SA_NOCLDWAIT`: no dejar zombies a los hijos (sólo para SIGCHLD).
- `SA_NODEFER`: permitir que el handler sea reentrante.
- `SA_ONSTACK`: utilizar un stack alternativo.
- `SA_RESETHAND`: volver al default handler luego de invocar al handler.
- `SA_RESTART`: relanzar las syscalls interrumpidas por esta señal.

3.4. Ejemplo de uso

El siguiente es un ejemplo sencillo de uso de sigaction (disponible en la web de la asignatura -`sigaction_example.c`-) para especificar un nuevo manejador:

```
/* 2014 - from http://www.linuxprogrammingblog.com/code-examples/sigaction */
/* Example of using sigaction() to setup a signal handler with 3 arguments
 * including siginfo_t.
 */
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
        printf ("Sending PID: %ld, UID: %ld\n",
                        (long)siginfo->si_pid, (long)siginfo->si_uid);
}
```

```
int main (int argc, char *argv[])
{
        struct sigaction act;

        memset (&act, '\0', sizeof(act));

        // Use the sa_sigaction field because the handler has two additional
        // parameters
        act.sa_sigaction = &hdl;

        // The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field,
        // not sa_handler.
        act.sa_flags = SA_SIGINFO;

        if (sigaction(SIGTERM, &act, NULL) < 0) {
                perror ("sigaction");
                return 1;
        }

        while (1)
                sleep (10);

        return 0;
}


//Salida: Sending PID: 19121, UID: 1001
```

En este caso el handler `hdl` imprime un mensaje indicando el proceso que originó la señal y el UID del usuario que lanzó tal proceso, a partir de la estructura `siginfo_t`.

Analicemos en detalle el `main`:

- Utilizaremos una sola acción (`act`): sólo nos interesa configurar una nueva acción, no haremos nada con la acción anterior.
- `memset` se encarga de poner en null todos los punteros y vaciar máscaras y flags.
- Al asignar `sa_sigaction` se guarda el puntero al nuevo handler, pero el sistema aún no conoce esta acción.
- Al asignar `SA_SIGINFO` a `sa_flags` se indica que se quiere utilizar el puntero especificado en `sa_sigaction`.
- La llamada a `sigaction` configura la nueva acción para la señal SIGTERM. El tercer argumento `NULL` indica que no se requiere el `sigaction` anterior.
- Luego el programa ingresa en un loop infinito.

Para probar el programa, ejecutarlo y desde otra terminal hacer:

```
ps ax  # buscar aquí el PID del proceso sigaction_example
kill PID # reemplazar PID por el pid del proceso sigaction_example
         # recordar que kill por defecto envía SIGTERM
kill PID     # ¿qué pasa ahora?
kill -9 PID  # ¿qué pasa ahora?
```

3.5. Un ejemplo más complejo

El siguiente ejemplo demuestra cómo puede obtenerse mucha más información utilizando la estructura `ucontext_t`. No veremos todos los detalles, pero vale la pena observar:

- En este caso se utiliza la misma acción para `SIGSEGV` y para `SIGUSR`.
- Se usan funciones de librería para trabajar con la máscara de señales: `sigemptyset`.
- El manejador puede obtener la dirección del fallo utilizando el campo `si_addr` de la estructura `siginfo_t`.
- La información de contexto permite obtener el estado de los registros al momento de producirse el evento (`gregs[REG_RIP]`).

```
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>
#include <execinfo.h>
#include <stdlib.h>
/* get REG_RIP from ucontext.h */
#include <ucontext.h>
void bt_sighandler(int sig, siginfo_t *info,
                                    void *secret) {
  void *trace[16];
  char **messages = (char **)NULL;
  int i, trace_size = 0;
  ucontext_t *uc = (ucontext_t *)secret;

  if (sig == SIGSEGV)
        printf("Got signal %d, faulty address is %p, "
                    "from %p\n", sig, info->si_addr,
                    uc->uc_mcontext.gregs[REG_RIP]);
  else
        printf("Got signal %d\n", sig);

  trace_size = backtrace(trace, 16);
  /* overwrite sigaction with caller's address */
  trace[1] = (void *) uc->uc_mcontext.gregs[REG_RIP];

  messages = backtrace_symbols(trace, trace_size);
  /* skip first stack frame (points here) */
  printf("[bt] Execution path:\n");
  for (i=1; i<trace_size; ++i)
```

```
        printf("[bt] %s\n", messages[i]);
  exit(0);
}

int func_a(int a, char b) {
  char *p = (char *)0xdeadbeef;
  a = a + b;
  *p = 10;        /* CRASH here!! */
  return 2*a;
}

int func_b() {
  int res, a = 5;
  res = 5 + func_a(a, 't');
  return res;
}

int main() {
  struct sigaction sa;

  sa.sa_sigaction = (void *)bt_sighandler;
  sigemptyset (&sa.sa_mask);
  sa.sa_flags = SA_RESTART | SA_SIGINFO;

  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGUSR1, &sa, NULL);
  printf("%d\n", func_b());

}
```