

UNIX System Programming

Signals

- ❖ Objectives
 - Introduce signals
 - Concentrate on `sigaction()` function

Overview

1. Definition
2. Signal Types
3. Generating a Signal
4. Responding to a Signal
5. Common Uses of Signals
6. Implementing a `read()` Timeout

7. POSIX Signal Functions
8. Interrupted System Calls
9. System Calls inside Handlers
10. More Information

1. Definition

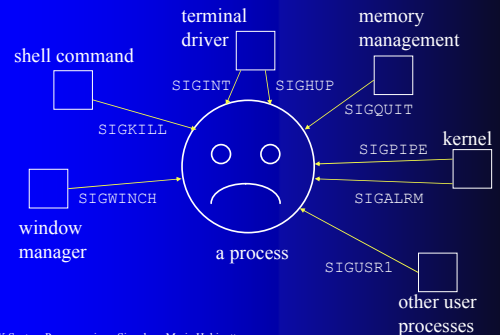
- ❖ A signal is an *asynchronous* event which is delivered to a process.
- ❖ Asynchronous means that the event can occur at any time
 - may be unrelated to the execution of the process
 - e.g. user types `ctrl-C`, or the modem hangs

2. Signal Types (31 in POSIX)

<i>Name</i>	<i>Description</i>	<i>Default Action</i>
SIGINT	Interrupt character typed	terminate process
SIGQUIT	Quit character typed (^\\)	create core image
SIGKILL	kill -9	terminate process
SIGSEGV	Invalid memory reference	create core image
SIGPIPE	Write on pipe but no reader	terminate process
SIGALRM	alarm() clock 'rings'	terminate process
SIGUSR1	user-defined signal type	terminate process
SIGUSR2	user-defined signal type	terminate process
	:	

- ❖ See man 7 signal

Signal Sources



3. Generating a Signal

- ❖ Use the UNIX command:

```
$ kill -KILL 4481
```

 - send a SIGKILL signal to pid 4481
 - check
 - ♦ `ps -l`
 - to make sure process died
- ❖ `kill` is not a good name; `send_signal` might be better.

kill()

- ❖ Send a signal to a process (or group of processes).
- ❖ `#include <signal.h>`

```
int kill( pid_t pid, int signo );
```
- ❖ Return 0 if ok, -1 on error.

Some pid Values

- | ❖ <i>pid</i> | <i>Meaning</i> |
|--------------|---|
| > 0 | send signal to process <code>pid</code> |
| == 0 | send signal to all processes whose process group ID equals the sender's <code>pgid</code> .
e.g. parent kills all children |

4. Responding to a Signal

- ❖ A process can:
 - ignore/discard the signal (not possible with SIGKILL or SIGSTOP)
 - execute a **signal handler** function, and then possibly resume execution or terminate
 - carry out the default action for that signal
- ❖ The choice is called the process' *signal disposition*

signal(): library call

- ❖ Specify a signal handler function to deal with a signal type.
- ❖ `#include <signal.h>`

```
typedef void Sigfunc(int); /* my defn */
```

```
Sigfunc *signal( int signo, Sigfunc *handler );
```

 - `signal` returns a pointer to a function that returns an `int` (i.e. it returns a pointer to `Sigfunc`)
- ❖ Returns *previous* signal disposition if ok, `SIG_ERR` on error.

Actual Prototype

- ❖ The actual prototype, listed in the “man” page is a bit perplexing but is an expansion of the `Sigfunc` type:

```
void (*signal(int signo, void(*handler)(int)))(int);
```
- ❖ In Linux:

```
typedef void (*sig_handler_t)(int);
```

```
sig_handler_t signal(int signo, sig_handler_t handler);
```
- ❖ `Signal` returns a pointer to a function that returns an `int`

The signal function itself returns a **pointer** to a function. The return type is the same as the function that is passed in, i.e., a function that takes an int and returns a void.

The **handler** function receives a single integer argument and returns void.

Signal is a function that takes two arguments: **sig** and **handler**.

The function to be called when the signal is received is referred to as the **handler**. The returned function takes a integer parameter.

```

#include <signal.h>
void (*signal( int sig, void (*handler)(int)))(int) ;
/* signal returns a pointer to the PREVIOUS signal handler */

```

1730 UNIX System Programming: Signals Maria Hybinette 13

Example

```

int main()
{
    signal( SIGINT, foo );
    :
    /* do usual things until SIGINT */
    return 0;
}

void foo( int signo )
{
    : /* deal with SIGINT signal */
    return; /* return to program */
}

```

1730 UNIX System Programming: Signals Maria Hybinette 14

sig_examp.c

```

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

void sig_usr( int signo ); /* handles two signals */

int main()
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        printf( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        printf( "Cannot catch SIGUSR2\n" );
    :
}

```

1730 UNIX System Programming: Signals Maria Hybinette 15 *continued*

```

while(1)
{
    printf( "%2d\n", I );
    pause();
    /* pause until signal handler
    * has processed signal */
    i++;
}

return 0;
}

```

1730 UNIX System Programming: Signals Maria Hybinette 16 *continued*

```

void sig_usr( int signo )
/* argument is signal number */
{
    if( signo == SIGUSR1 )
        printf( "Received SIGUSR1\n" );
    else if( signo == SIGUSR2 )
        printf( "Received SIGUSR2\n" );
    else
        printf( "Error: received signal
        %d\n", signo );

    return;
}

```

1730 UNIX System Programming: Signals Maria Hybinette 17

Usage

```

$ sig_examp &
[1] 4720
0
$ kill -USR1 4720
Received SIGUSR1
1
$ kill -USR2 4720
Received SIGUSR2
2
$ kill 4720 /* send SIGTERM */
[1] + Terminated sig_examp &
$

```

1730 UNIX System Programming: Signals Maria Hybinette 18

Special Sigfunc * Values

❖ Value	Meaning
SIG_IGN	Ignore / discard the signal.
SIG_DFL	Use default action to handle signal.
SIG_ERR	Returned by <code>signal()</code> as an error.

Multiple Signals

- ❖ If many signals of the *same* type are waiting to be handled (e.g. two `SIGINT`s), then most UNIXs will only deliver **one** of them.
 - the others are thrown away
- ❖ If many signals of *different* types are waiting to be handled (e.g. a `SIGINT`, `SIGSEGV`, `SIGUSR1`), they are not delivered in any fixed order.

pause()

- ❖ Suspend the calling process until a signal is caught.
- ❖ `#include <unistd.h>`
`int pause(void);`
- ❖ Returns `-1` with `errno` assigned `EINTR`. (Linux assigns it `ERESTARTNOHAND`).
- ❖ `pause()` only returns after a signal handler has returned.

The Reset Problem

- ❖ In Linux (and many other UNIXs), the signal disposition in a process is **reset** to its **default action** immediately after the signal has been delivered.
- ❖ Must call `signal()` again to reinstall the signal handler function.

Reset Problem Example

```
int main()
{
    signal(SIGINT, foo);
    :
    /* do usual things until SIGINT */
}

void foo(int signo)
{
    signal(SIGINT, foo); /* reinstall */
    :
    return;
}
```

Reset Problem

```
void ouch( int sig )
{
    printf( "OUCH! I got signal %d\n", sig );
    (void) signal(SIGINT, ouch);
}

int main()
{
    (void) signal( SIGINT,
while(1)
{
    printf("Hello World!\n");
    sleep(1);
}
}
```

To keep catching the signal with this function, must call the `signal` system call again.

Problem: from the time that the interrupt function starts to just before the signal handler is re-established the signal will not be handled.

If another `SIGINT` signal is received during this time, default behavior will be done, i.e., program will terminate.

Re-installation may be too slow!

- ❖ There is a (very) small time period in `foo()` when a new `SIGINT` signal will cause the default action to be carried out -- process termination.
- ❖ With `signal()` there is no answer to this problem.
 - POSIX signal functions solve it (and some other later UNIXs)

5. Common Uses of Signals

- 5.1. Ignore a Signal
- 5.2. Clean up and Terminate
- 5.3. Dynamic Reconfiguration
- 5.4. Report Status
- 5.5. Turn Debugging on/off
- 5.6. Restore Previous Handler

5.1. Ignore a Signal

```
int main()
{
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
    /* do work without interruptions */
}
```

- ❖ Cannot ignore/handle `SIGKILL` or `SIGSTOP`
- ❖ Should check for `SIG_ERR`

5.2. Clean up and Terminate

```
/* global variables */
int my_children_pids;

void clean_up(int signo);

int main()
{
    signal(SIGINT, clean_up);
}
```

continued

```
void clean_up(int signo)
{
    unlink("/tmp/work-file");
    kill(my_children_pids, SIGTERM);
    wait((int *)0);
    fprintf(stderr,
            "Program terminated\n");
    exit(1);
}
```

Problems

- ❖ If a program is run in the **background** then the interrupt and quit signals (`SIGINT`, `SIGQUIT`) are automatically ignored.
- ❖ Your code should not override these changes:
 - check if the signal dispositions are `SIG_IGN`

Checking the Disposition

```
new disposition      old disposition
:
if( signal(SIGINT, SIG_IGN ) != SIG_IGN )
    signal(SIGINT, clean_up);

if( signal(SIGQUIT, SIG_IGN ) != SIG_IGN )
    signal(SIGQUIT, clean_up);
:
```

- ❖ *Note:* cannot check the signal disposition without changing it (sigaction that we will look at later, is different)

5.3. Dynamic Reconfiguration

```
:
void read_config(int signo);

int main()
{
    read_config(0); /* dummy argument */

    while (1)
        /* work forever */
}
```

```
void read_config(int signo)
{
    int fd;

    signal( SIGHUP, read_config );

    fd = open("config_file", O_RDONLY);
    /* read file and set global vars */
    close(fd);

    return;
}
```

Problems

- ❖ Reset problem
- ❖ Handler interruption
 - what is the effect of a SIGHUP in the middle of read_config()'s execution?
- ❖ Can only affect global variables.

5.4. Report Status

```
:
void print_status(int signo);
int count; /* global */

int main()
{ signal(SIGUSR1, print_status);
:
  for( count=0; count < BIG_NUM; count++ )
  {
    /* read block from tape */
    /* write block to disk */
  }
  ...
}
```

```
void print_status(int signo)
{
    signal(SIGUSR1, print_status);
    printf("%d blocks copied\n", count);
    return;
}
```

- ❖ Reset problem
- ❖ count value not always defined.
- ❖ Must use global variables for status information

5.5. Turn Debugging on/off

```
void toggle_debug(int signo);

int debug = 0; /* initialize here */

int main()
{
    signal(SIGUSR2, toggle_debug);

    /* do work */
    if (debug == 1)
        printf("...");
    ...
}
```

```
void toggle_debug(int signo)
{
    signal(SIGUSR2, toggle_debug);

    debug = ((debug == 1) ? 0 : 1);

    return;
}
```

5.6. Restore Previous Handler

```
sigfunc *old_hand;

/* set action for SIGTERM;
   save old handler */
old_hand = signal(SIGTERM, foobar);

/* do work */

/* restore old handler */
signal(SIGTERM, old_hand);
:
```

6. Implementing a read() Timeout

- ❖ Put an upper limit on an operation that might block forever
 - e.g. `read()`
- ❖ 6.1. `alarm()`
- ❖ 6.2. Bad `read()` Timeout
- ❖ 6.3. `setjmp()` and `longjmp()`
- ❖ 6.4. Better `read()` Timeout

6.1. `alarm()`

- ❖ Set an alarm timer that will ‘ring’ after a specified number of seconds
 - a `SIGALRM` signal is generated
- ❖ `#include <unistd.h>`
`long alarm(long secs);`
- ❖ Returns 0 or number of seconds until previously set alarm would have ‘rung’.

Some Tricky Aspects

- ❖ A process can have **at most one** alarm timer running at once.
- ❖ If `alarm()` is called when there is an **existing** alarm set then it returns the number of seconds remaining for the old alarm, and sets the timer to the new alarm value.
 - What do we do with the “old alarm value”?
- ❖ An `alarm(0)` call causes the previous alarm to be cancelled.

6.2. Bad read() Timeout

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

#define MAXLINE 512

void sig_alm( int signo );

int main()
{
    int n;
    char line[MAXLINE];
    :
```

```
    if( signal(SIGALRM, sig_alm) == SIG_ERR )
    {
        printf("signal(SIGALRM) error\n");
        exit(1);
    }

    alarm(10);
    n = read( 0, line, MAXLINE );
    alarm(0);

    if( n < 0 ) /* read error */
        fprintf( stderr, "\nread error\n" );
    else
        write( 1, line, n );
    return 0;
}
```

```
void sig_alm(int signo)
/* do nothing, just handle signal */
{
    return;
}
```

Problems

- ❖ The code assumes that the `read()` call terminates with an error after being interrupted (talk about this later).
- ❖ **Race Condition:** The kernel may take longer than 10 seconds to start the `read()` after the `alarm()` call.
 - the alarm may ‘ring’ before the `read()` starts
 - then the `read()` is not being timed; may block forever
 - Two ways to solve this one uses `setjmp` and the other uses `sigprocmask` and `sigsuspend`

6.3. setjmp() and longjmp()

- ❖ In C we cannot use `goto` to jump to a label in another function
 - use `setjmp()` and `longjmp()` for those ‘long jumps’
- ❖ Only uses which are good style:
 - error handling which requires a deeply nested function to recover to a higher level (e.g. back to `main()`)
 - coding timeouts with signals

Prototypes

- ❖ `#include <setjmp.h>`
`int setjmp(jmp_buf env);`
- ❖ Returns 0 if called directly, non-zero if returning from a call to `longjmp()`.
- ❖ `#include <setjmp.h>`
`void longjmp(jmp_buf env, int val);`

Behavior

- ❖ In the `setjmp()` call, `env` is initialized to information about the current state of the stack.
- ❖ The `longjmp()` call causes the stack to be reset to its `env` value.
- ❖ Execution restarts after the `setjmp()` call, but this time `setjmp()` returns `val`.

Example

```
jmp_buf env;          /* global */

int main()
{
    char line[MAX];
    int errval;

    if(( errval = setjmp(env) ) != 0 )
        printf( "error %d: restart\n", errval );

    while( fgets( line, MAX, stdin ) != NULL )
        process_line(line);
    return 0;
}
```

```
void process_line( char * ptr )
{
    :
    cmd_add()
    :
}

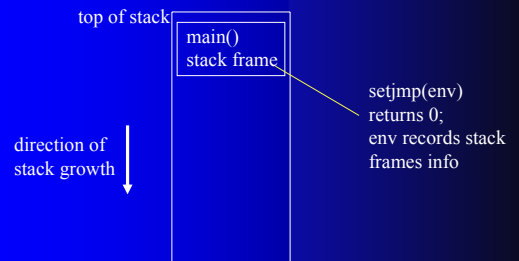
void cmd_add()
{
    int token;

    token = get_token();
    if( token < 0 ) /* bad error */
        longjmp( env, 1 );

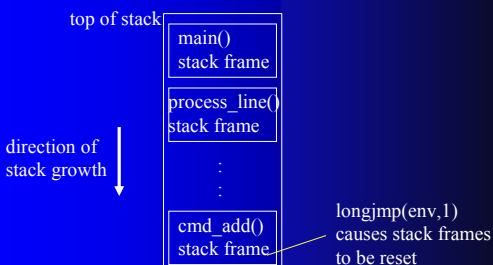
    /* normal processing */
}

int get_token()
{
    if( some error )
        longjmp( env, 2 );
}
```

Stack Frames at setjmp()



Stack Frames at longjmp()



sleep1()

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

void sig_alrm( int signo )
{
    return; /* return to wake up pause */
}

unsigned int sleep1( unsigned int nsecs )
{
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
        return( nsecs );

    alarm( nsecs ); /* starts timer */
    pause(); /* next caught signal wakes */
    return( alarm( 0 ) ); /* turn off timer, return unslept
    * time */
}
```

sleep2()

```
static void jmp_buf env_alarm;

void sig_alrm( int signo )
{
    longjmp( env_alarm, 1 );
}

unsigned int sleep2( unsigned int nsecs )
{
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
        return( nsecs );
    if( setjmp( env_alarm ) == 0 )
    {
        alarm( nsecs );           /* starts timer */
        pause();                 /* next caught signal wakes */
    }
    return( alarm( 0 ) );
}
```

Sleep1 and Sleep2

- ❖ Sleep2 fixes race condition. Even if the pause is never executed.
- ❖ There is one more problem (will talk about that after “fixing the earlier read function”)

Status of Variables?

- ❖ The POSIX standard says:
 - global and static variable values will not be changed by the longjmp() call
- ❖ Nothing is specified about local variables, are they “rolled back” to their original values (at the setjmp call) as the stack?
 - they may be restored to their values at the first setjmp(), but maybe not
 - ◆ Most implementations do not roll back their values

6.4. Better read() Timeout

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

#define MAXLINE 512
void sig_alrm( int signo );
jmp_buf env_alarm;

int main()
{ int n;
  char line[MAXLINE];
  :
}
```

```
if( signal(SIGALRM, sig_alrm) == SIG_ERR)
{
    printf("signal(SIGALRM) error\n");
    exit(1);
}

if( setjmp(env_alarm) != 0 )
{
    fprintf(stderr, "\nread() too slow\n");
    exit(2);
}

alarm(10);
n = read(0, line, MAXLINE);
alarm(0);
:
```

```

:
if( n < 0 ) /* read error */
    fprintf( stderr, "\nread error\n" );
else
    write( 1, line, n );
return 0;
}
```

```

void sig_alm(int signo)
/* interrupt the read() and jump to
  setjmp() call with value 1
*/
{
  longjmp(env_alm, 1);
}

```

Caveat: Non-local Jumps

From the UNIX **man** pages:

WARNINGS

If `longjmp()` or `siglongjmp()` are called even though `env` was never primed by a call to `setjmp()` or `sigsetjmp()`, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

A Problem Remains!

- ❖ If the program has several signal handlers then:
 - execution might be inside one when an alarm ‘rings’
 - the `longjmp()` call will jump to the `setjmp()` location, and abort the other signal handler -- might lose / corrupt data

7. POSIX Signal Functions

- ❖ The POSIX signal functions can control signals in more ways:
 - can *block signals* for a while, and deliver them later (good for coding critical sections)
 - can *switch off the resetting* of the signal disposition when a handler is called (no reset problem)

- ❖ The POSIX signal system, uses **signal sets**, to deal with pending signals that might otherwise be missed while a signal is being processed

7.1. Signal Sets

- ❖ The signal set stores collections of signal types.
- ❖ Sets are used by signal functions to define which signal types are to be processed.
- ❖ POSIX contains several functions for creating, changing and examining signal sets.

Prototypes

❖ #include <signal.h>

```
int sigemptyset( sigset_t *set );
int sigfillset( sigset_t *set );

int sigaddset( sigset_t *set, int signo );
int sigdelset( sigset_t *set, int signo );

int sigismember( const sigset_t *set,
                 int signo );
```

7.2. sigprocmask()

❖ A process uses a signal set to create a mask which defines the signals it is **blocking** from delivery. – good for critical sections where you want to block certain signals.

❖ #include <signal.h>

```
int sigprocmask( int how,
                 const sigset_t *set,
                 sigset_t *oldset );
```

❖ **how** – indicates how mask is modified

how Meanings

<i>Value</i>	<i>Meaning</i>
SIG_BLOCK	set signals are added to mask
SIG_UNBLOCK	set signals are removed from mask
SIG_SETMASK	set becomes new mask

A Critical Code Region

```
        :
sigset_t newmask, oldmask;

sigemptyset( &newmask );
sigaddset( &newmask, SIGINT );

/* block SIGINT; save old mask */
sigprocmask( SIG_BLOCK, &newmask, &oldmask );

/* critical region of code */

/* reset mask which unblocks SIGINT */
sigprocmask( SIG_SETMASK, &oldmask, NULL );
        :
```

7.3. sigaction()

❖ Supercedes (more powerful than) signal()
– sigaction() can be used to code a non-resetting signal()

❖ #include <signal.h>

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

sigaction Structure

```
struct sigaction
{
    void      (*sa_handler)( int );
              /* action to be taken or SIG_IGN, SIG_DFL */
    sigset_t  sa_mask; /* additional signal to be blocked */
    int       sa_flags; /* modifies action of the signal */
    void      (*sa_sigaction)( int, siginfo_t *, void * );
}
;
```

❖ **sa_flags** –
– SIG_DFL reset handler to default upon return
– SA_SIGINFO denotes **extra information** is passed to handler (i.e. specifies the use of the “second” handler in the structure.

sigaction() Behavior

- ❖ A `signo` signal causes the `sa_handler` signal handler to be called.
- ❖ While `sa_handler` executes, the signals in `sa_mask` are blocked. Any more `signo` signals are also blocked.
- ❖ `sa_handler` remains installed until it is changed by another `sigaction()` call. No reset problem.

Signal Raising

```
int main()
{
    struct sigaction act;

    act.sa_handler = ouch;

    sigemptyset( &act.sa_mask );
    act.sa_flags = 0;

    sigaction( SIGINT, &act, 0 );

    printf( "Hello\n" );
    sleep(1);
}
```

```
struct sigaction
{
    void (*) (int) sa_handler;
    sigset_t sa_mask;
    int sa_flags;
}
```

Set the signal handler to be the function ouch

We can manipulate sets of signals..

No flags are needed here. Possible flags include: SA_NOCLDSTOP SA_RESETHAND

This call sets the signal handler for the SIGINT (ctrl-C) signal

Signal Raising

- ❖ This function will continually capture the ctrl-C (SIGINT) signal.
- ❖ Default behavior is **not** restored after signal is caught.
- ❖ To terminate the program, must type ctrl-\, the SIGQUIT signal.

sigexPOS.c

```
/* sigexPOS.c - demonstrate sigaction() */
/* include files as before */

int main(void)
{
    /* struct to deal with action on signal set */
    static struct sigaction act;

    void catchint(int); /* user signal handler */

    /* set up action to take on receipt of SIGINT */
    act.sa_handler = catchint;
}
```

```
/* create full set of signals */
sigfillset(&(act.sa_mask));

/* before sigaction call, SIGINT will terminate
 * process */

/* now, SIGINT will cause catchint to be executed */
sigaction( SIGINT, &act, NULL );
sigaction( SIGQUIT, &act, NULL );

printf("sleep call #1\n");
sleep(1);

/* rest of program as before */
```

Signals - Ignoring signals

- ❖ Other than SIGKILL and SIGSTOP, signals can be ignored:
- ❖ Instead of in the previous program:

```
act.sa_handler = catchint /* or whatever */
```

We use:

```
act.sa_handler = SIG_IGN;
```

The ^C key will be ignored

Restoring previous action

- ❖ The third parameter to `sigaction`, `oact`, can be used:

```
/* save old action */
sigaction( SIGTERM, NULL, &oact );

/* set new action */
act.sa_handler = SIG_IGN;

sigaction( SIGTERM, &act, NULL );

/* restore old action */
sigaction( SIGTERM, &oact, NULL );
```

A Basic signal()

```
#include <signal.h>

Sigfunc *signal( int signo, Sigfunc *func )
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset( &act.sa_mask );
    act.sa_flags = 0;

    act.sa_flags |= SA_INTERRUPT;
    if( signo != SIGALRM )
        act.sa_flags |= SA_RESTART;
    /* any system call interrupted by a signal
     * other than alarm is restarted */
    if( sigaction( signo, &act, &oact ) < 0 )
        return( SIG_ERR );
    return( oact.sa_handler );
}
```

7.4. Other POSIX Functions

- ❖ `sigpending()` examine blocked signals
- ❖ `sigsetjmp()`
`siglongjmp()` jump functions for use in signal handlers which handle masks correctly
- ❖ `sigsuspend()` atomically reset mask and sleep

[sig]longjmp & [sig]setjmp

NOTES (longjmp, sigjmp)

POSIX does not specify whether `longjmp` will restore the signal context. If you want to save and restore **signal masks**, use `siglongjmp`.

NOTES (setjmp, sigjmp)

POSIX does not specify whether `setjmp` will save the signal context. (In `SYSV` it will not. In `BSD4.3` it will, and there is a function `_setjmp` that will not.) If you want to save signal masks, use `sigsetjmp`.

Example

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

sigjmp_buf buf;

void handler(int sig)
{
    siglongjmp(buf, 1);
}

main()
{
    signal(SIGINT, handler);
    if( sigsetjmp(buf, 1) == 0 )
        printf("starting\n");
    else
        printf("restarting\n");
    ;
    ...
}
```

```
-
while(1)
{
    sleep(5);
    printf(" waiting...\n");
}
```

```
> a.out
starting
waiting...
waiting...
restarting ← Control-c
waiting...
waiting...
waiting...
restarting ← Control-c
waiting...
restarting ← Control-c
waiting...
restarting
waiting...
waiting...
```

8. Interrupted System Calls

- ❖ When a system call (e.g. `read()`) is interrupted by a signal, a signal handler is called, returns, and then what?
- ❖ On many UNIXs, *slow* system function calls do not resume. Instead they return an error and `errno` is assigned `EINTR`.
 - true of Linux, but can be altered with (Linux-specific) `siginterrupt()`

Slow System Functions

- ❖ Slow system functions carry out I/O on things that can possibly block the caller forever:
 - pipes, terminal drivers, networks
 - some IPC functions
 - `pause()`, some uses of `ioctl()`
- ❖ Can use signals on slow system functions to code up timeouts (e.g. did earlier)

Non-slow System Functions

- ❖ Most system functions are non-slow, including ones that do *disk* I/O
 - e.g. `read()` of a disk file
 - `read()` is sometimes a slow function, sometimes not
- ❖ Some UNIXs resume non-slow system functions after the handler has finished.
- ❖ Some UNIXs only call the handler after the non-slow system function call has finished.

9. System Calls inside Handlers

- ❖ If a system function is called inside a signal handler then it may interact with an interrupted call to the same function in the main code.
 - e.g. `malloc()`
- ❖ This is not a problem if the function is *reentrant*
 - a process can contain multiple calls to these functions at the same time
 - e.g. `read()`, `write()`, `fork()`, many more

Non-reentrant Functions

- ❖ A functions may be non-reentrant (only one call to it at once) for a number of reasons:
 - it uses a static data structure
 - it manipulates the heap: `malloc()`, `free()`, etc.
 - it uses the standard I/O library
 - ◆ e.g. `scanf()`, `printf()`
 - ◆ the library uses global data structures in a non-reentrant way

errno Problem

- ❖ `errno` is usually represented by a global variable.
- ❖ Its value in the program can be changed suddenly by a signal handler which produces a new system function error.