Overview

Last Week:
- How to program UNIX processes (Chapters 7-9)
- `fork()` and `exec()`

This Week:
- UNIX inter-process communication mechanism: signals, pipes and FIFOs.
- How to program with UNIX signals (Chapter 10)
- Non-local jumps (Chapter 7)
- Focus on the `sigaction()` function

Outline

- What is a UNIX signal?
- Signal types
- Generating signals
- Responding to a signal
- Common uses of a signal
- Implementing a `read()` time-out
- Non-local jumps `setjmp()`/`longjmp()`
- POSIX signals
- Interrupted system calls
- System calls inside handlers

What is a Signal?

- A signal is an asynchronous event which is delivered to a process (instantiated by a small message)
- 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
- Sent from kernel (e.g. detects divide by zero (`SIGFPE`) or could be at the request of another process to send to another)
- Only information that a signal carries is its unique ID and that it arrived

Signal Types (31 in POSIX)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Interrupt from keyboard (<code>^C</code>)</td>
<td>terminate</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>Quit from keyboard (<code>^)</code> )</td>
<td>terminate &amp; core</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td><code>kill -9</code></td>
<td>terminate</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Invalid memory reference</td>
<td>terminate &amp; core</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td><code>alarm()</code> clock ‘rings’</td>
<td>terminate</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Child stopped or terminated</td>
<td>ignore</td>
</tr>
<tr>
<td>16</td>
<td>SIGUSR1</td>
<td>user-defined signal type</td>
<td>terminate</td>
</tr>
</tbody>
</table>

- `/usr/include/sys/iso/signal_iso.h` on atlas
Generating a Signal

- Use the UNIX command:
  {saffron} /fork_example
  (saffron) kill -KILL 6676
  » sends a SIGKILL signal to
  processor ID {pid} 6676
  » check pid via (and also to
  make sure it died)
  (saffron) ps -l

- Signal is not a good name; send_signal might be better.

Responding to a Signal

- After receiving a signal a process can:
  1. Ignore/Discard/Block out the signal (not possible with
     SIGKILL or SIGSTOP)
  2. Catch the signal; execute a signal handler function, and then possibly resume execution or terminate
  3. Carry out the default action for that signal

- The choice is called the process' signal disposition

5. signal 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

6. The returned function takes a single integer argument and returns void

Using the UNIX command:
{safron} /fork_example
Terminating Parent, PID = 6675
Running Child, PID = 6676
(saffron) ps
PID TTY TIME CMD
6585 ttyp9 00:00:06 /fork_example
6676 ttyp9 00:00:06 /fork_example
(saffron) kill - 9 6676
(saffron) ps
PID TTY TIME CMD
6585 ttyp9 00:00:06 /fork_example
6676 ttyp9 00:00:06 /fork_example
6678 ttyp9 00:00:06 ps

kill()

#include <signal.h>
int kill( pid_t pid, int signo );

- Send a signal to a process (or group of processes).
- Return 0 if ok, -1 on error.

pid

> 0 "send signal to process pid"

== 0 "send signal to all processes whose process group ID equals the sender's pgid. e.g. parent kills all children"

signal()

#include <signal.h>

void (*signal( int signo, void (*func)(int) ))(int);

define signal type:

typedef void (*Sigfunc( int ) ); /* Plauger 1992 definition */

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

- Signal returns a pointer to a function that returns an int (i.e. it returns a pointer to Sigfunc)
- Specify a signal handler function to deal with a signal type.
- Returns previous signal disposition if OK, SIG_ERR on error.

Sketch on how to program with signals

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

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

- If many signals of the same type are waiting to be handled (e.g. two SIGINTs), then most UNIXs will only deliver one of them.
  - the others are thrown away - i.e. pending signals are not queued
  - for each signal type, just have a single bit indicating whether or not the signal has occurred
- 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.

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```

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

static void sig_usr( int signo )
{
    if( signo == SIGUSR1 )
        printf( "BOOM!\n" );
    else if( signo == SIGUSR2 )
        printf( "BEEP\n" );
    else
        fprintf( stderr, "ERROR: received signal %d\n", signo );
    exit(1);
    return;
}
```

```c
int main( void )
{
    int i = 0;
    if( signal( SIGUSR1, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2, sig_usr ) == SIG_ERR )
        perror( "Cannot catch SIGUSR2\n" );
    while( 1 )
    {
        printf( "%d: ", i );
        pause(); /* until signal handler has processed signal */
        i++;
    }
    return 0;
}
```
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

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

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

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)

Common Uses of Signals

- Ignore a signal
- Clean up and terminate
- Dynamic reconfiguration
- Report status
- Turn debugging on/off
- Restore a previous handler

Ignore a Signal

```c
int main()
{
    signal( SIGINT, SIG_IGN );
    signal( SIGQUIT, SIG_IGN );
    /* do work without interruptions */
}
```
### Clean up and Terminate

- 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

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

### Dynamic Reconfiguration

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

```c
void read_config(int signo)
{
    int fd;
    signal(SIGUSR1, read_config);
    fd = open("config_file", O_RDONLY);
    /* read file and set global vars */
    close(fd);
    return;
}
```

### Checking the Disposition

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

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

### Report Status

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

```c
void print_status(int signo)
{
    printf( "%d
", count );
    return;
}
```

### Turn Debugging On/Off

```c
void toggle_debug(int signo)
{
    signal(SIGHUP, toggle_debug);
    debug = !debug;
    return;
}
```

### 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);
- /* work forever */
### Implementing a read() timeout

- Put an upper limit on an operation that might block forever
  - e.g. read()
- alarm()
- Implementing various timeouts
  - Bad read() timeout
  - setjmp() and longjmp()
  - Better read() timeout

### 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.

### 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 out; may block forever
  - Two ways two solve this one uses setjmp and the other uses sigprocmask and sigsuspend

### alarm()

```c
#include <unistd.h>
long alarm( long secs );
```

- Set an alarm timer that will ‘ring’ after a specified number of seconds
  - a SIGALRM signal is generated
- Returns 0 or number of seconds until previously set alarm would have ‘rung’.

### Bad read() Timeout

```c
void sig_alrm( int signo )
{
    return;
}
```

- If the kernel generates a SIGALRM signal before the read() starts, read() returns an error
  - sig_alrm() is called when there is an error

### 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
Nonlocal Jumps: `setjmp()` & `longjmp()`

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
  - Controlled way to break the procedure call/return discipline
  - Useful for error recovery and signal recovery
- `setjmp(jmp_buf)`
  - Called before `longjmp()`
  - Identified return site for subsequent `longjmp()`
  - Called once, returns one or more times
- Implementation:
  - Remember where you are by storing the current register context, stack pointer and PC value in `jmp_buf`
  - Returns 0

Prototypes

```c
#include <setjmp.h>
int setjmp( jmp_buf env );
void longjmp( jmp_buf env, int val );
```

- Returns 0 if called directly, non-zero if returning from a call to `longjmp()`.
- 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 `jmp_buf` value (never returns)
- Execution restarts after the `setjmp()` call, but this time `setjmp()` returns `val`.

Restart when ctrl-c’d: `setlongjmp.c`

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

int main()
{
    siglongjmp( buf, 1 );
}
```

Stack Frames at `setjmp()`

- Top of stack
- `main()` stack frame
- `setjmp(env)` returns 0; `env` records stack frames info

Another use for `setjmp/longjmp()`: `sleep1()`

- Alarm erases "old" set alarms
  - Look at return value from the previous `alarm()` call
  - If less than new `alarm()` - wait until old `alarm()` expires
  - If larger than new `alarm()` - reset old `alarm()` with remaining seconds when done with new `alarm()`
- Lose old disposition of `SIGALRM`
  - Save old disposition and restore when done
- Race condition between first call to alarm and the call to `pause`:
  - Never get out of pause (fix via `setjmp/longjmp` or `sigsuspend`)

Stack frames at `longjmp()`

- Top of stack
- `main()` stack frame
- `process_line()` stack frame
- `cmd_add()` stack frame
- `longjmp(env, 1)` causes stack frames to be reset
A Problem Remains!

- **Recall**: 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

**Sleep2()**: Avoids the race condition

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

static void sleep2_alrm_handler()
{
    printf( "sleep2() interrupted
    by alarm()
    return
    ");
    return;
}

void sleep2()
{
    unsigned int unslept;
    static int n;
    char line[MAXLINE];
    if( signal( SIGALRM, sleep2_alrm_handler ) == SIG_ERR )
        perror( "signal(SIGALRM) error" );
    exit(1);
}
```

**Better read() Timeout**

```c
int main( void )
{
    int n;
    char line[MAXLINE];
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
        perror( "signal(SIGALRM) error" );
    exit(1);
}
```

**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

**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 Interrupted Handler**

```c
void sig_int(int signo)
{
    unsigned int unslept;
    static int n;
    char line[MAXLINE];
    if( signal( SIGINT, sig_int ) == SIG_ERR )
        perror( "signal(SIGINT) error" );
    exit(1);
}
```
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)
  - can queue pending signals

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.

POSIX.1 Prototypes

```c
#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 );
```

- sigemptyset - initializes signal set pointed by set so that all signals are excluded
- sigfillset - all signals are included
- sigaddset - add a single signal (signo) to set
- sigdelset - remove signo from set

how Meanings

```c
#include <signal.h>
int sigprocmask( int how, const sigset_t *set, sigset_t *oldset);
```

- how - indicates how mask is modified (later)

- Value | Meaning
  - SIG_BLOCK | set signals are added to mask
  - SIG_UNBLOCK | set signals are removed from mask
  - SIG_SETMASK | set becomes new mask
### Example: A Critical Code Region

```c
#include <signal.h>
#include <stdio.h>
void ouch(int signo) {
    printf("OUCH! ", signo);
    // critical region of code
    printf("Hello World!
");
    // reset mask which unblocks SIGINT /*
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    // SIGDRL reset handler to default upon return
    if (signo == SIGINT)
        return;
    // SIGDRL is caught.
    // To terminate the program, must type ctrl-
    // the SIGQUIT signal (or sent a TERM signal via kill)
}
```

### sigaction() Structure

```c
struct sigaction
{
    void (*sa_handler)(int); /* the action 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** — modifies the behaviour of signo
  - 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 I

```c
#include <signal.h>
#include <stdio.h>
void ouch(int signo) {
    printf("OUCH! signo = %d, signo =
    \n");
    int main()
    {    struct sigaction act;
        sa_handler = ouch;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        sigaction(SIGINT, &act, NULL);
    }
```

### 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 (or sent a TERM signal via `kill`
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;
    sigfillset(&(act.sa_mask)); /* before sigaction call, SIGINT will terminate */
    /* 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;

    Then 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 “Better”Reliable 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, 0 ) < 0 )
            return(SIG_ERR);
        return( oact.sa_handler );

    }

    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
**Interruption 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.
Non-reentrant functions

- A function 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.

Limitations of Nonlocal Jumps

- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;
P1()
{  P2(); P3(); }
P2()
{  if( setjmp( env ) /* long jump to here */ )
      P3()
      longjmp( env, 1 );
  }
```

```
env ... P1
  env ... P2
  At setjmp

env ... P1
  P2
  P3
  P2 returns

env ... P1
  P3
  At longjmp
```