Unix System Programming

Signals

Overview

Last Week:
- How to program UNIX processes ( Chapters 7-9) 
- \texttt{fork()} and \texttt{exec()} 

This Week, and next week:
- UNIX inter-process communication mechanisms: signals, 
  \hspace{1em} (next week) pipes and FIFOs.  
- How to program with UNIX signals (Chapter 10) 
  \hspace{1em} \url{http://en.wikipedia.org/wiki/Unix_signal}  
- Non-local jumps (Chapter 7) 
- Focus on the \texttt{sigaction()} function

Outline

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

What is a Signal?

- A signal is an \textit{asynchronous} event which is delivered to a process (instantiated by a small message) 
- \textit{Asynchronous} means that the event can occur at any time (e.g., posting at a bulletin board ) 
  \hspace{1em} may be unrelated to the execution of the process 
  \hspace{2em} e.g., user types Ctrl-C, or the modem hangs (SIGINT) 
  \hspace{2em} e.g., user types Ctrl-Z (SIGTSTP) 
- Sent from kernel (e.g. detects \texttt{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 (^C)</td>
<td>terminate</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>Quit from keyboard (^</td>
<td>terminate &amp; core</td>
</tr>
<tr>
<td>9</td>
<td>SIGILL</td>
<td>kill -9</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>alarm() 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 (solaris)` 
- `/usr/src/kernels/2.6.32-431.29.*/include/linux/signal.h` 
- `/usr/include/signal.h (user space)`

Signal Sources

Running an out process (division by 0, floating point exception)

Shell command

Terminal, window manager

Memory management

Kernel SIG

Other user processes
Generating a Signal

- Use the UNIX command:
  ```bash
  {saffron} kill -KILL 6676
  ```
  sends a SIGKILL signal to processor ID (pid) 6676
- check pid via (and also to make sure it died)
  ```bash
  {saffron} ps -l
  ```
- ```bash
  kill
  ``` is not a good name; `send_signal` might be better.
- How do we do this in a program?
  ```bash
  {saffron} ./fork_example
  Terminating Parent, PID = 6675
  Running Child, PID = 6676
  ```

```
PID TTY          TIME CMD
6585 ttyp9    00:00:00 tcsh
6676 ttyp9    00:00:06 fork_example
```

```bash
kill –s 9 6676
```

```
PID TTY          TIME CMD
6585 ttyp9    00:00:00 tcsh
6678 ttyp9    00:00:00 ps
```

kill()

```c
#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**
  ```c
  > 0
  ```
  send signal to process pid
  ```c
  == 0
  ```
  send signal to all processes whose process group ID equals the sender's pgid.
  e.g. parent kills all children

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**
- How is a process' disposition set?

```c
#include <signal.h>
typedef void Sigfunc(int); /* Plauger 1992 */
Sigfunc *signal( int signo, Sigfunc *handler );
```

- `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`
- `signal` returns a pointer to the previous signal handler

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

- The signal to be caught or ignored is given as argument `signo`
- The handler function receives a single integer argument and returns `void`
- The returned function takes an integer parameter.

Sketch on how to program with signals

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

```c
void foo( int signo )
{
  /* deal with SIGINT signal */
  return; /* return to program */
}
```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

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

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) {
        if( ++i > 3 ) {
            printf("BOOM!\n");
            exit(0);
        } else {
            printf("%d: ", i);
            pause(); /* until signal handler has processed signal */
            printf("/\n");
            i++;
        }
        return 0;
    }
}

ps -u
kill -SIGUSR2 2108

External Signal Example:
signal_example.c

int beeps = 0;
static void handler(int signo) {
    printf("BEEP\n");
    fflush(stdout);
    if( ++beeps < 5 ) {
        alarm(1);
    } else {
        printf("BOOM!\n");
        exit(0);
    }
    return;
}

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

kill -STOP 1255
kill -CONT 1255
kill -USR1 1255
kill -QUIT 1255

Internal Signal Example:
signal_example2.c

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

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) {
        if( ++i > 3 ) {
            printf("BOOM!\n");
            exit(0);
        } else {
            printf("%d: ", i);
            pause(); /* until signal handler has processed signal */
            printf("/\n");
            i++;
        }
        return 0;
    }
}

Internal Signal Example:
signal_example2.c

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

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) {
        if( ++i > 3 ) {
            printf("BOOM!\n");
            exit(0);
        } else {
            printf("%d: ", i);
            pause(); /* until signal handler has processed signal */
            printf("/\n");
            i++;
        }
        return 0;
    }
}

kill -STOP 1255
kill -CONT 1255
kill -USR1 1255
kill -QUIT 1255

Special Sigfunc * Values

- **Value**: Meaning
  - SIG_IGN: Ignore / discard the signal.
  - SIG_DFL: Use default action to handle signal.
  - SIG_ERR: Returned by `signal()` as an error.

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.

pause()

#include <unistd.h>
int pause(void);

- Suspend the calling process until a signal is caught.
- Returns -1 with errno assigned EINTR (Linux assigns it EESTARTNOHAND).
- 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

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

void clean_up( int signo );

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

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

Checking the Disposition

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

```c
void clean_up( int signo )
{
  unlink( "/tmp/work-file" );
  kill( my_children_pids, SIGTERM );
  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 main()
{
  read_config(0);
  /* dummy argument */
  while (1) /* work forever */
}
```

```c
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;
}
```

Report Status

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

```c
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 */
  }
  ...
}
```

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

Turn Debugging On/Off

```c
void toggle_debug(int signo);

int debug = 0;

int main()
{
  signal( SIGUSR2, toggle_debug );
  if (debug == 1)
  {
    printf("...");
  }
  ...
}
```

```c
void toggle_debug(int signo)
{
  signal(SIGUSR2, toggle_debug);
  debug = ((debug == 1) ? 0 : 1);
  return;
}
```

Restore Previous Handler

```c
Sigfunc *old_hand;

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

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

#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

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#define MAXLINE 512

void sig_alrm( int signo );

int main()
{
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
    {
        printf("signal(SIGALRM) error\n");
        exit(1);
    }
    alarm(10);
    n = read( STDIN_FILENO, line, MAXLINE );
    alarm(0);
    if( n < 0 ) /* read error */
        fprintf( stderr, "read error\n");
    else
        write( STDOUT_FILENO, line, n );
    return 0;
}

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

[sig]setjmp() and [sig]longjmp()

- In C we cannot use goto to jump to a label in another function
  - use [sig]setjmp() and [sig]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: [sig]setjmp() & [sig]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 recover
- **setjmp( jmp_buf env )**
  - 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 env** value (so it never returns)
- Execution restarts after the **setjmp()** call, but this time **setjmp()** returns **val** (so in way **val** is a way to send a message to the **setjmp** -- and consequently facilitates multiple **longjmp**’s per **setjmp**)

Stack Frames at **setjmp()**

```c
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
sigjmp_buf buf;
void handler(int sig)
{
    siglongjmp(buf, 1);
}
int main()
{
    signal(SIGINT, handler);
    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");
    while(1)
    {
        sleep(1);
        printf("processing...\n");
    }
}
```

Implementing **sleep1()**

- Using **alarm()** and **pause()** we can implement our own **sleep()** function (a sleep function puts a process to sleep for a specified amount of time).
- **Idea:** Use **pause()** that waits for a specific amount of time until we get a signal.
- Set the amount of time we want to sleep via **alarm()**.
Implementing: sleep1()

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

void sig_alrm( int signo )
{
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
    {
        return( alarm( 0 ) );
    }
    alarm( nsecs ); /* starts timer */
    pause(); /* caught signal wakes */
    return( alarm( 0 ) ); /* turn off timer return unslept time */
}
```

- 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 sigprocmask/sigsuspend)

Maria Hybinette, UGA

Problem

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

Status of Variables after longjmp?

- The POSIX standard says:
  - global and static variable values will be left alone 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?"
  - "It depends": they may be restored to their values at the first setjmp(), but maybe not
    - Most implementations do not roll back their values

Maria Hybinette, UGA

Better read() Timeout

```c
int main( void )
{
    int n;
    char line[MAXLINE];
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
    {
        return( alarm( 0 ) );
    }
    printf( "signal(SIGALRM) error\n" );
    exit( 1 );
}
```

- solves earlier race conditions:
  - Now if alarm occurs "before" it gets to "read" it jumps to setjmp at exits instead of doing nothing and blocks forever in the read
  - and if the system call is restarted the return of the signal handler still have an effect
  - still have same problem with other signal handlers...

Maria Hybinette, UGA

Interrupted Handler

```c
int main( void )
{
    int n;
    char line[MAXLINE];
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
    {
        return( alarm( 0 ) );
    }
    printf( "signal(SIGALRM) error\n" );
    exit( 1 );
}
```

- Here: longjmp() aborts the sig_int signal handler even if it did not complete (the for loop)
- We will see ways around these problems soon.

Maria Hybinette, UGA

sleep2(): Avoids the race condition

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

void sig_alrm( int signo )
{
    if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
    {
        return( alarm( 0 ) );
    }
    alarm( nsecs ); /* starts timer */
    return( alarm( 0 ) ); /* turn off timer return unslept time */
}
```

- sleep2() fixes race condition. Even if the pause is never executed.
  - a SIGALRM causes sleep2() to return
  - Avoids entering pause() via longjmp()
- There is one more problem
  - SIGALRM could interrupt some other signal handler and subsequently abort it by executing the longjmp()
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.

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

POSIX 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**
```

sigprocmask()

```c
#include <signal.h>

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

- 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.

```c
#include <signal.h>

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

- **how** – indicates how mask is modified (later)
- **oldset** - current signal mask
Meanings

- **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
: 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 );
```

---

**sigaction()**

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

- Supersedes (more powerful than) `signal()`
- `sigaction()` can be used to code a non-resetting `signal()`
- `signo` is the signal you want to perform an action on
- `act` is the action
- `oldact` is the old action (can be set to `NULL`, if uninteresting)
- Cannot handle SIGSTOP and SIGKILL

---

**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.

---

**sigaction() Structure**

```
struct sigaction {
  void (*) (int) sa_handler; /* 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)

---

Signal Handling

```
#include <signal.h>
#include <stdio.h>
void ouch( int signo )
{
  printf("OUCH! signo = %d\n", signo );
}
int main()
{
  struct sigaction act;
  act.sa_handler = ouch;
  sigemptyset( &act.sa_mask );
  act.sa_flags = 0;
  sigaction( SIGINT, &act, NULL );
  while( 1 )
  {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

- We can manipulate sets of signals...
- This call sets the signal handler for the SIGINT (Ctrl-c) signal
- No flags are needed here.
- Possible flags include:
  - SA_NOCLDSTOP
  - SA_RESETHAND
  - SA_RESTART
  - SA_NODEFER
**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`)

```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:
  ```c
  act.sa_handler = catchint /* or whatever */
  ```
  We use:
  ```c
  act.sa_handler = SIG_IGN;
  ```
  Then the ^C key will be ignored

```c
/* 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 );
```

**Restoring previous action**

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

```c
#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, NULL ) < 0 )
        return(SIG_ERR);
    return( oact.sa_handler );
}
```
Other POSIX Functions

- **sigpending()**
  - examine blocked signals

- **sigsetjmp()** and **siglongjmp()**
  - jump functions for use in signal handlers which handle masks correctly

- **sigsuspend()**
  - atomically reset mask and sleep

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

```c
#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
");
    else
        printf("restarting
");
    …
    while(1) {
        sleep(5);
        printf("waiting...
");
    }
}
```

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.
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 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

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

env
→ P1
At `setjmp`

env → X
P1
P2
P2 returns
At `longjmp`

P3
env → X