Unix System Programming

Signals

Overview

Last Week:

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

This Week, and next week:

- UNIX inter-process communication mechanisms: signals,
  - » (next week) pipes and FIFOs.
- How to program with UNIX signals (Chapter 10)
  - » http://en.wikipedia.org/wiki/Unix_signal
- 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?

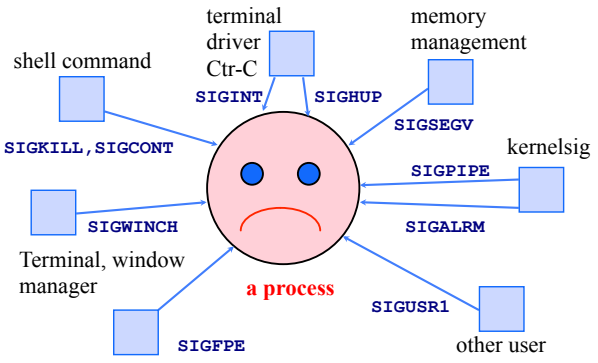
- A signal is an *asynchronous* event which is delivered to a process (instantiated by a small message)
- Asynchronous means that the event can occur at any time (e.g., posting at a bulletin board)
  - » may be unrelated to the execution of the process
    - – e.g., user types `Ctrl-C`, or the modem hangs (`SIGINT`)
    - – e.g,, user types `Ctrl-Z` (`SIGTSTP`)
- 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)

ID	Name	Description	Default Action
2	SIGINT	Interrupt from keyboard (^C)	terminate
3	SIGQUIT	Quit from keyboard (^\)	terminate & core
9	SIGKILL	kill -9	terminate
11	SIGSEGV	Invalid memory reference	terminate & core
14	SIGALRM	alarm() clock 'rings'	terminate
17	SIGCHLD	Child stopped or terminated	ignore
16	SIGUSR1	user-defined signal type	terminate

- `/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 a.out process (division by 0, floating point exception)

Generating a Signal

- Use the UNIX command:
  `{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`

```
{saffron} ./fork_example
Terminating Parent, PID = 6675
Running Child, PID = 6676
{saffron} ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 fork_example
{saffron} kill -s 9 6676
{saffron} ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- `kill` is not a good name; `send_signal` might be better.
- How do we do this in a program?

`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	*Meaning*
`> 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

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?

`signal()`

```
#include <signal.h>

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

typedef void Sigfunc( int );  /* Plauger 1992 definition */
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`)
- Specify a signal handler function to deal with a signal type.
- Returns *previous* signal disposition if OK, `SIG_ERR` on error.

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`

2. The signal to be *caught* or *ignored* is given as argument *signo*

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

- listed in the "man" p
- xpansion of the `Sigfunc` type:

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

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

1. `signal` takes two arguments: *signo* and *handler*

3. The function to be called when the specified signal is received is given as a *pointer* to the function *handler*

6. The returned function takes a integer parameter.

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 */
     }
```

External Signal Example: `signal_example.c`

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

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

```
ps -u
kill -SIGUSR2 21084
```



```
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;
  }
```

```
{saffron:ingrid:54} signal_example
0: Received SIGUSR1
1: Received SIGUSR1
2: Received SIGUSR2
[1] + Stopped (signal)
signal_example
{saffron:ingrid:26} fg
signal_example
3: Received SIGUSR1
Quit
```

```
{saffron:ingrid:55} kill -l=
{saffron:ingrid:56} ps -l
```

```
{saffron:ingrid:23} kill -USR1 1255
{saffron:ingrid:24} kill -USR1 1255
{saffron:ingrid:25} kill -USR2 1255
{saffron:ingrid:26} kill -STOP 1255
{saffron:ingrid:27} kill -CONT 1255
{saffron:ingrid:28} kill -USR1 1255

{saffron:ingrid:29} kill QUIT 1255
```


Internal Signal Example: `signal_example2.c`

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

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(); i++;
        }
   return 0;
  }
```

```
{cinnamon} signal_example2
0: BEEP
1: BEEP
2: BEEP
3: BEEP
4: BEEP
BOOM!
```


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 `SIGINT`s), 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 occured
- 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 `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, ouch );
        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)

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

```
     :
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`

Clean up and Terminate

```
    :
/* global variables */
int my_children_pids;
    :
void clean_up( int signo );

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

- 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`

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


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)

Dynamic Reconfiguration

```
   :
void read_config( int signo );

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

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

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

```
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

```
    :
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

Turn Debugging On/Off

```
   :
void toggle_debug(int signo);

/* initialize here */
int debug = 0;

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;
}
```


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 );
    :
```

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

`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'.

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.

Bad `read()` Timeout

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

#define MAXLINE  512

void sig_alrm( int signo );

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

          :
```

```
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, "\nread error\n" );
else
    write( STDOUT_FILENO, line, n );
return 0;
}
```

```
    void sig_alrm( 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 out; may block forever
  - » Two ways two solve this one uses `setjmp` and the other uses `sigprocmask` and `sigsuspend`

`[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 j )`
  - » 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

```
#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`)

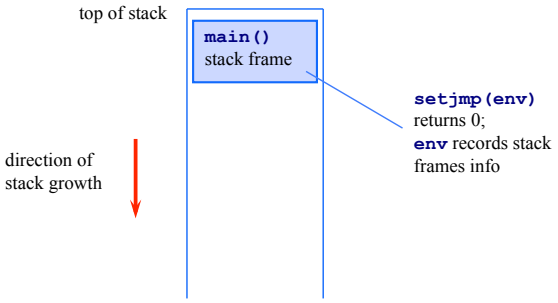
Restart when ctrl-c'd: setlongjmp.c

```
#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");
```

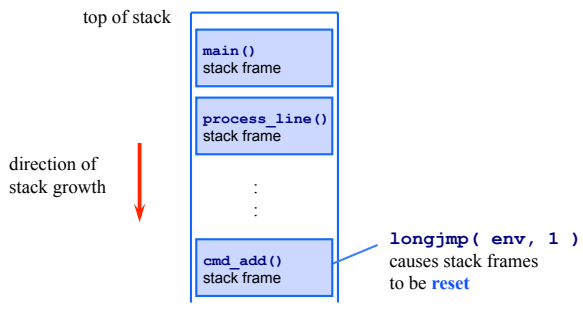
```
while( 1 )
  {
  sleep(1);
  printf("processing...\n");
  }
}
```

```
{cinnamon:ingrid:34} setlongjmp
starting
processing...
^Crestarting        <- Ctrl-C
restarting
processing...
Terminated          <- kill -9
{cinnamon:ingrid:35}
```


Stack Frames at setjmp()



Stack frames at longjmp()



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()

```
#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();                 /* caught signal wakes */
    return( alarm( 0 ) );    /* turn off timer return
                                un-slept 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`)

sleep2(): Avoids the race condition

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

static void jmp_buf env_alrm;

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

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

- `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()`

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

Interrupted Handler

```
int main( void )
 {
 unsigned int unslept;

 if( signal( SIGINT, sig_int ) == SIG_ERR )
    perror( "signal(SIGINT) error" );
 unslept = sleep2( 5 );
 printf( "sleep2 returned: %u\n", unslept );
 exit(0)
 }
```

```
void sig_int( int signo )
 {
 int i;
 int j;
 printf( "sig_int starting\n" );
 for( i = 0; i < 2000000; i++ )
    j += i * i ;
 printf( "sig_int finished\n" );
 return;
 }
```

```
{saffron} a.out
^C
sig_int starting
sleep2 returned: 0
```

- 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.

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

Better read() Timeout

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

if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
    {
    printf( "signal(SIGALRM) error\n" );
    exit(1);
    }
if( setjmp( env_alrm ) != 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_alrm(int signo)
   {
   longjmp( env_alrm, 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 re-started the return of the signal handler still have an effect
  - » still have same problem with other signal handlers…

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

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

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

```
#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()`

- 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 (later)
- `oldset` - current signal mask

how 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

```
        :
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 );
```

- Supercedes (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
- `oact` is the old action (can be set to `NULL`, if uninteresting)
- Cannot handle `SIGSTOP` and `SIGKILL`

sigaction() Structure

```
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

```
#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 );

                                  rld!\n");
```

```
struct sigaction
        {
        void (*) (int) sa_handler
{cinnamon:ingrid:8} sigact
Hello World!        sigset_t sa_mask
Hello World!        int sa_flags
Hello World!
Hello World!

OUCH! signo = 2
Hello World!
Hello World! Hello World!
Hello World!
terminated
```

No flags are needed here.
Possible flags include:
SA_NOCLDSTOP

Set the signal handler to be the function ouch

We can manipulate sets of signals..

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 (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;
```



```
/* 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;
```

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, &oact ) < 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

[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...        <-- Control-c
restarting
  waiting...
  waiting...
  waiting...
restarting          <-- Control-c
  waiting...
restarting          <-- Control-c
  waiting...
  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 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.

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 );
}
```