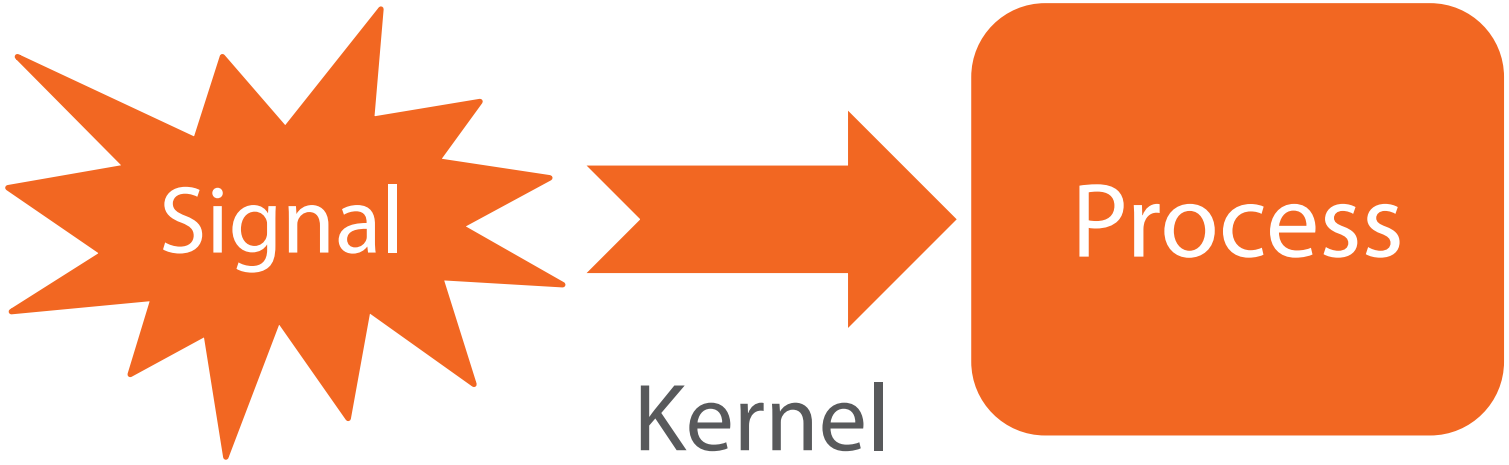
Mastering Signals

Chris Brown

What Is a Signal?



In This Module …

Common signal types and their uses

Raising signals

Writing signal handlers

Seven useful things to do with signals

Signal Types

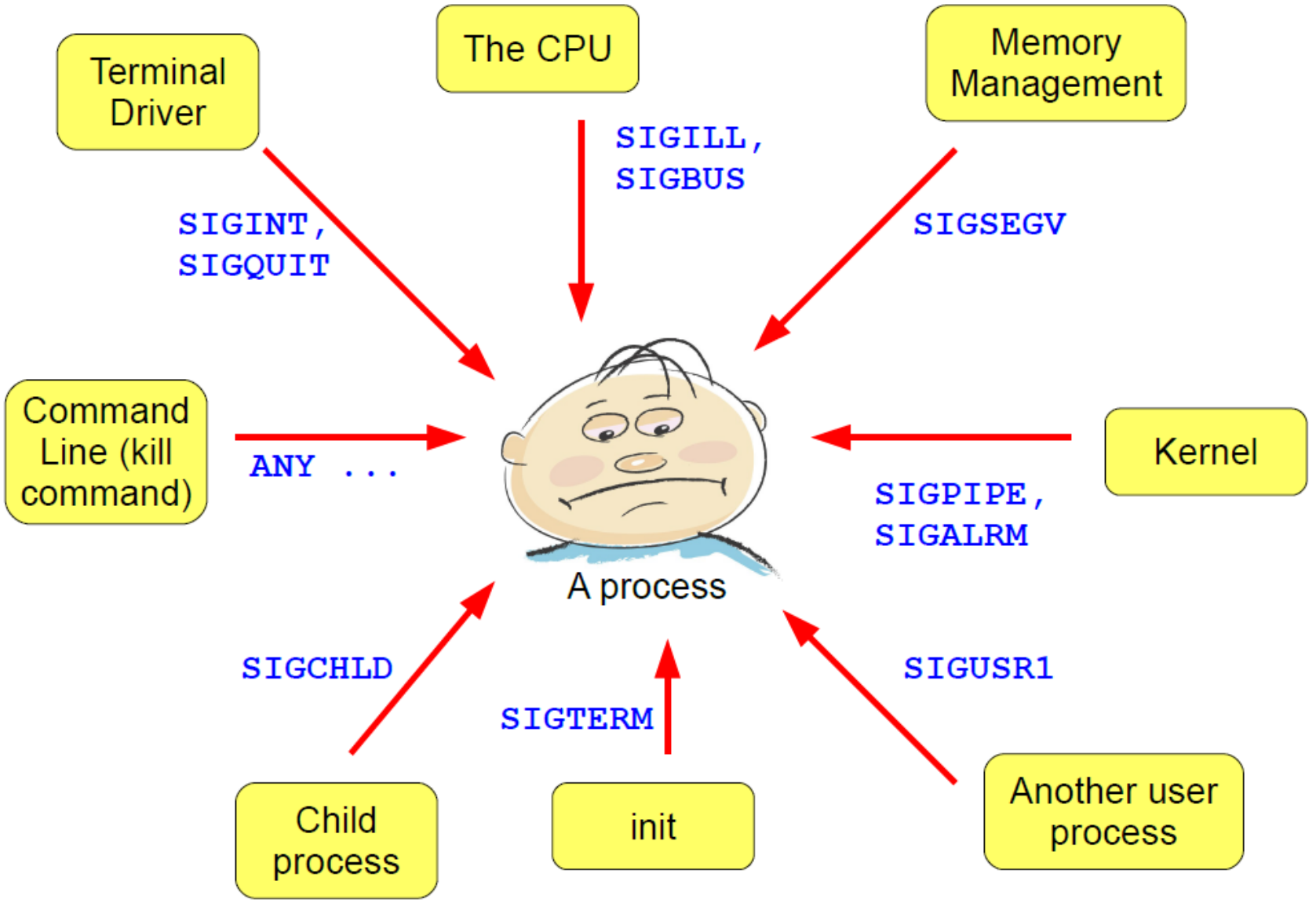
```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS
```

POSIX real-time signals

Signal Types

Signal Name	Number	Default Action	Description
SIGHUP	1	Term	Some daemons interpret this to mean "re-read your configuration file"
SIGINT	2	Term	The signal sent by ^C on terminal
SIGTRAP	5	Core	Trace/breakpoint trap
SIGFPE	8	Core	Arithmetic error, e.g. divide by zero
SIGKILL	9	Term	Lethal signal, cannot be caught or ignored
SIGSEGV	11	Core	Invalid memory reference
SIGALRM	14	Term	Expiry of alarm clock timer
SIGTERM	15	Term	Polite "please terminate" signal
SIGCHLD	17	Ignore	Child process has terminated

Where Do Signals Come From?



Sending Signals

Target process ID

Signal number
(or symbolic constant such as SIGTERM)
Use `0` to test for process existence

```
kill(3644, 15);
```

Returns 0 on success
-1 on error

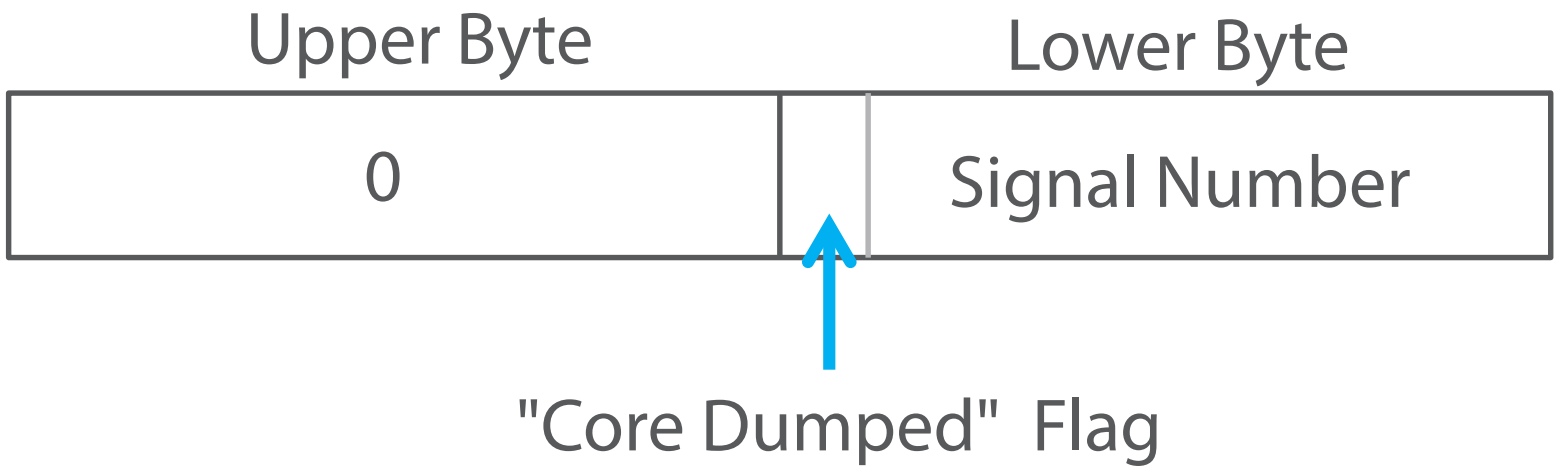
Use `raise(signum)`
to send yourself a signal

Your real or effective UID must match
the UID of the target process

Process Termination

- The default action of most signal types is to terminate the process
  - For some signals a memory image (core file) is also written

MACRO	Meaning
`WIFSIGNALED(status)`	True if child terminated by signal
`WTERMSIG(status)`	The signal number
`WCOREDUMP(status)`	True if child produced core dump



Establishing a Signal Handler

We will see a better way to do this later

Signal type

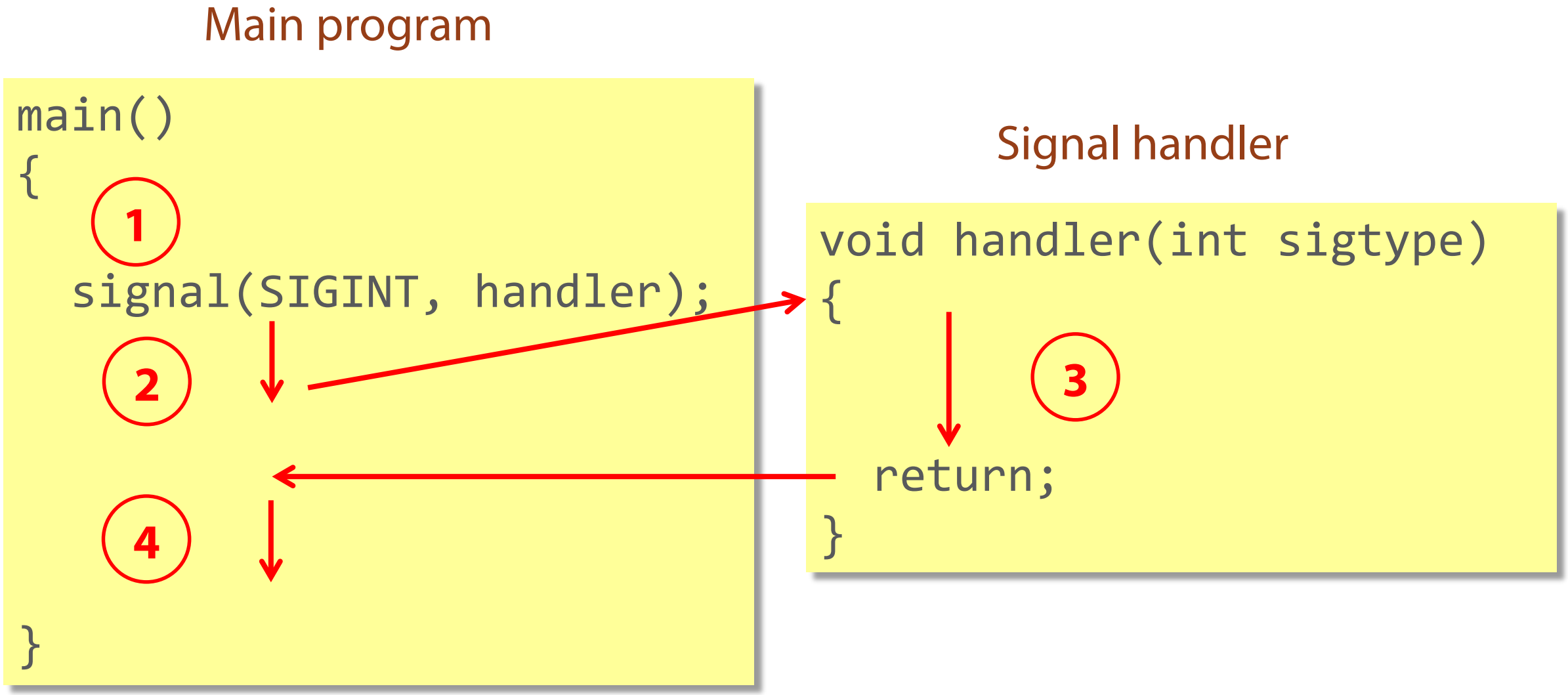
```
signal(SIGHUP, hup_handler);
```

Returns a pointer to the previous signal handler

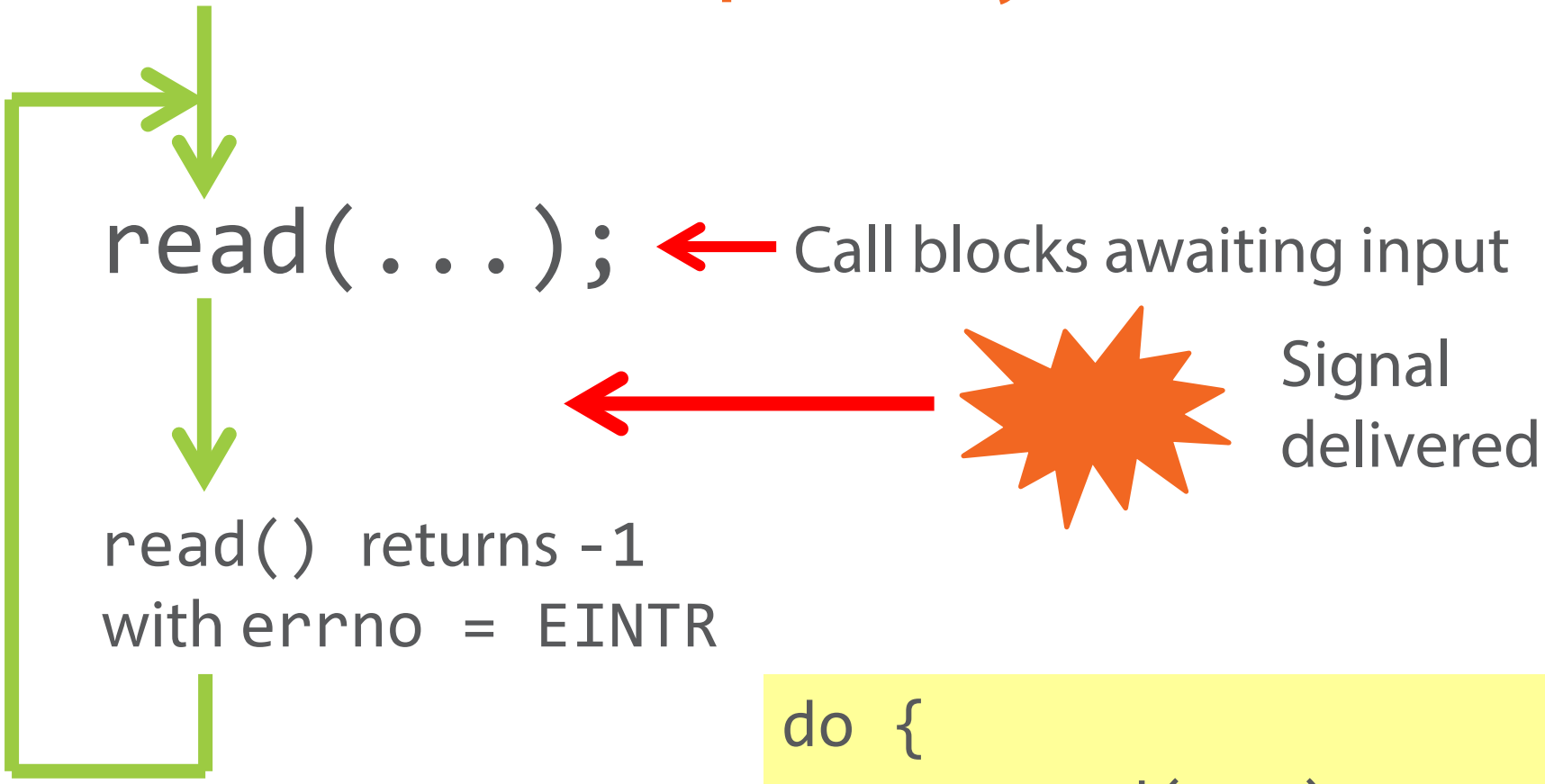
Pointer to signal handler function, or one of:

`SIG_IGN`	Ignore signal
`SIG_DFL`	Restore default

Flow Control on Receipt of a Signal

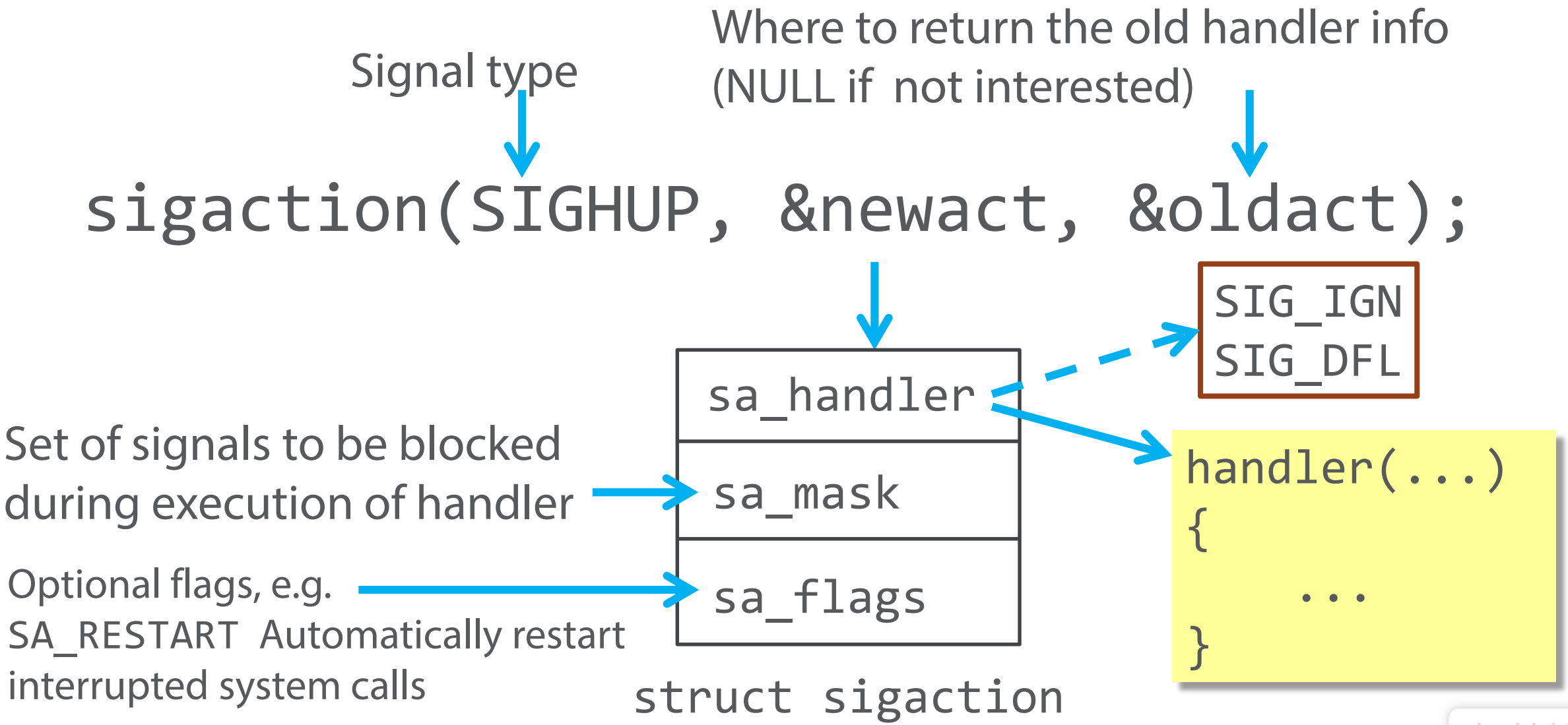


Interrupted System Calls




```
do {
   n = read(...);
} while (n < 0 || errno == EINTR)
```

Establishing Handlers with `sigaction()`



The sigaction Structure

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

Flag	Meaning
SA_NODEFER	Do not prevent the signal from being received within its own signal handler
SA_RESTART	Enables automatic restarting of interrupted system calls
SA_RESETHAND	Restore the signal to its default action on entry to the signal handler

Signal Sets

The `sigset_t` data type is an array of booleans representing a set of signals

`sigset_t s`

```
sigemptyset(&s);
```

Signal Sets

The `sigset_t` data type is an array of booleans representing a set of signals

`sigset_t s`

```
sigaddset(&s, SIGTERM);

sigaddset(&s, SIGQUIT);
```

Signal Sets

The `sigset_t` data type is an array of booleans representing a set of signals

`sigset_t s`

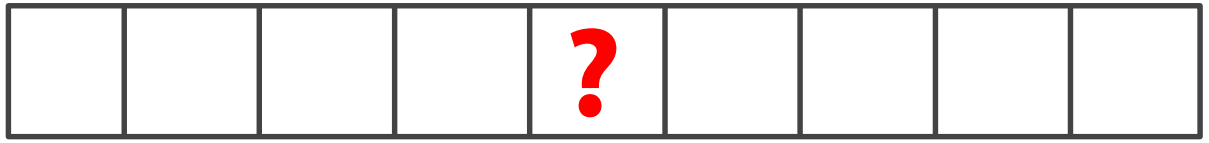
```
sigdelset(&s, SIGTERM);

sigfillset(&s);
```

Signal Sets

The `sigset_t` data type is an array of booleans representing a set of signals

`sigset_t s`




```
sigismember(&s, SIGTERM);
```

Returns 1 if signal is in set
0 if not

sigaction() Example

```
void handler(int sigtype)
{   ...   }

main()
{  struct sigaction action;

   action.sa_handler = handler;
   sigemptyset(&action.sa_mask);
   action.sa_flags = SA_RESTART;
   sigaction(SIGINT, &action, NULL);
   ...
}
```

Blocking Signals

Each process has a *signal mask* – a set of signals currently blocked from delivery

Blocked signals are held pending and will be delivered when unblocked

- Multiple pending signals of the same type are *not* queued

Add or remove signals from the mask using `sigprocmask()`

Blocking Signals

SIG_BLOCK:	Add these signals to the mask
SIG_UNBLOCK:	Remove these signals from the mask
SIG_SETMASK:	Assign this signal set to the mask

```
sigprocmask(how, set, oldset);
```

Set of signals to add/subtract

Return the previous mask here

Blocking Signals Example

```
sigset_t set;

sigemptyset(&set);
sigaddset(&set, SIGHUP);

sigprocmask(SIG_BLOCK, &set, NULL);

/* Code here will not be interrupted by SIGHUP */

sigprocmask(SIG_UNBLOCK, &set, NULL);
```

Seven Things to Do with Signals

1. Ignore them
2. Clean up and terminate
3. Reconfigure on-the-fly
4. Report status dynamically
5. Turn debugging on and off
6. Implement a timeout
7. Schedule periodic actions

1

Ignore them

```
signal(SIGINT, SIG_IGN);
signal(SIGQUIT, SIG_IGN);
... and so on ...
```

or …

Attempts to block SIGKILL are silently ignored

```
sigset_t s;
sigfillset(&s);
sigdelset(&s, SIGHUP);
sigprocmask(SIG_SETMASK, &s, NULL)
... proceed with only SIGHUP unblocked ...
```

2
Clean up and terminate

`SIGTERM` is usually intended as a request to terminate gracefully

- Flush any in-memory data to file
- Remove any temporary resources that would outlive the process: Files, message queues, named pipes, child processes, etc.

```
int my_child_pid;
void cleanup(int sigtype)
{ unlink("/tmp/myworkfile");
  kill(my_child_pid, SIGTERM);
  wait(NULL);
  exit(1);
}

main()
{ signal(SIGTERM, cleanup);
  open("/tmp/myworkfile", O_RDWR|O_CREAT, 0644);
  my_child_pid = fork() .... ;
  /* Get on with some work ... */
}
```

2

Clean up and terminate

Daemon processes often consult a configuration file when they start up

To allow the daemon to be re-configured without restarting, it typically installs a signal handler to re-read the file.

3 Reconfigure on-the-fly

The daemon can control when this can and cannot happen by setting a process mask

4

Report status dynamically

Long-running programs may accumulate internal state information

- Client state tables
- Routing tables
- Traffic summaries / usage counts

Install a handler to print this on demand, to aid debugging

- Write to terminal if there is one, else to a file
- State information must be global to be accessible within signal handler

```
int count;  /* Global state information */

void print_state_info(int sig)
{ printf("%d blocks copies\n", count);
}

void main()
{
  signal(SIGUSR1, print_state_info);
  for (count = 0; count < A_BIG_NUMBER; count++) {
    read_block_from_input_tape();
    write_block_to_output_tape();
  }
}
```

4

Report status dynamically

5

Turn debugging on and off

The idea is to sprinkle `printf()` or `fprintf()` statements into the code to provide a debug trace, and be able to enable and disable them dynamically

Boolean flag controls `printf()` calls

— Flag is flipped each time signal is delivered

5

Turn debugging on and off

```
int debug_on = 0;

void toggle_debug_flag(int sig)
{  debug_on ^= 1;
}

main()
{  signal(SIGHUP, toggle_debug_flag);

   /* Then sometime later in the code ... */

   if (debug_on)
      printf("something useful for debugging");
}
```

6
Implement a timeout

Blocking system calls such as `read()` do not timeout
- Potentially block forever

Using the interval timer to generate `SIGALRM`, a timeout can be implemented

Relies on *not* restarting the system call after a signal

7
Schedule periodic actions

`SIGALRM` signals can be used to trigger periodic actions

The trick is simply to request another alarm before exiting the handler

This program simply accumulates text from `stdin` into a buffer
Every 10 seconds the buffer is automatically saved to a file

Module Summary

Common signals and their uses

Raising signals

Writing signal handlers

Seven uses for signals