

Signals

Signals are how kernel notifies processes of some events and severe errors.

Only a constant representing type/case, no data. Examples:

- ▶ interrupt (Ctrl-C): SIGINT
- ▶ broken pipe: SIGPIPE
- ▶ suspend and resume: SIGSTOP, SIGCONT
- ▶ child died/suspended/resumed: SIGCHLD
- ▶ request for termination (shell 'kill' default): SIGTERM
- ▶ hard request for termination: SIGKILL
- ▶ illegal memory access (two types: SIGBUS, SIGSEGV)
- ▶ application-specific: SIGUSR1, SIGUSR2

Signal Life Cycle

Some event “generates” a signal.
Kernel tries to “deliver” the signal.

The signal is “pending” until delivered. Common cause of prolonged pending: Process may “mask” (aka “block”) a signal—pending until unmasked.

No multiplicity: Only which types are pending, not how many times.

Upon delivery: Default actions vary over ignore, suspend, resume, killed, killed with memory dump (core dump). Most overridable and may install signal handler functions, except SIGKILL.

Normal execution resumes if signal ignored or handler returns normally, but: If handler, syscalls fail with EINTR (but overridable).

Programmatically Generate A Signal

Shell command:

```
kill -SIGKILL 31337
```

```
kill -KILL 31337
```

```
kill -9 31337
```

System calls:

```
int kill(pid_t pid, int sig);
```

```
int raise(int sig); (to self)
```

Setting Signal Actions And Handlers

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

sig: Signal type in question.

act: New action you want.

oldact: for saving old action (e.g., if you want to restore later).

On fork: Signal actions cloned.

On exec: Handlers replaced by default, ignored remains ignored.

Demo (but also needs next 2 slides):

[signal-demo-1.c](#), [signal-demo-2.c](#), [signal-demo-3.c](#)

struct sigaction

```
struct sigaction {  
    void (*sa_handler)(int sig);  
    // ptr to handler function  
    // or SIG_IGN, or SIG_DFL  
  
    sigset_t sa_mask;  
    // mask which signals when running handler  
    // use next slide to set/query  
  
    int sa_flags;  
    // options  
  
    void (*sa_restorer)(void);  
    // not for application use  
};
```

sigset_t Operations

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
    // add all signals

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

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

Some Flags For sa_flags

If you install handler:

SA_NODEFER: Don't mask this signal when running handler. (Default: mask even if you didn't request, to avoid chicken-egg problems.)

SA_RESETHAND: Reset action to default before running handler.

SA_RESTART: Auto-restart most syscalls after handler returns. (Default: syscalls fail with `errno = EINTR`.) Some exceptions: `select`, `epoll`.

For SIGCHLD:

SA_NOCLDSTOP: Don't signal for child stop/cont.

SA_NOCLDWAIT: Don't turn terminated child into zombie.

Setting Signal Actions: Old Way

Old but simpler (but has a problem):

```
typedef void (*sighandler_t)(int sig);  
sighandler_t signal(int sig, sighandler_t handler);  
// i.e.,  
void (*signal(int sig, void (*handler)(int))(int));
```

The problem: When running your handler, are signals masked? Is action reset to default? After your handler returns, are syscalls restarted?

Answer: Vary across systems.

Not recommended unless you just set SIG_IGN or SIG_DFL.

Broken Pipe, SIGPIPE

When you write to pipe/socket but the other end has closed: “broken pipe”. Your process gets SIGPIPE.

Default action: Process killed.

Default makes sense for common pipelines, e.g.,

```
sort bigfile | head -1
```

head quits right after 1st line, no point letting sort continue.

Simplest way to override: Set action to SIG_IGN (ignore). Then process not killed, write returns -1, errno is EPIPE, you can check and react.

Handler Limitations

Unsafe to call e.g. `printf` inside handler. Reason:

Normal code is running another `printf`. In the middle, interrupted, signal arrives, handler is run.

`printf` has buffer and bookkeeping vars to update. If unfinished, in a not-yet-valid state.

If handler calls `printf` now, toasted.

Corollary: Unsafe to call `fclose(stdin)` too, same problem. Unsafe to call `exit` too, it includes `fclose(stdin)`.

Corollary: Inside handler, can't even clean up.

Likewise for some library functions (e.g., `free`), a few syscalls.

And using your own data structures that your normal code uses.

Handler Strategies

If non-trivial things to do upon signal: Do it outside handler.

- ▶ Make a global var or pipe (pipe preferred).
- ▶ Signal handler writes var/pipe to notify normal code that signal has happened. (write and many syscalls are safe in handler.)
- ▶ Normal code regularly checks var/pipe at convenient times. E.g., surely by the time you check, your recent `printf` has finished. Now safe to react, clean up, or exit.

For SIGCHLD: `wait` and `waitpid` are safe in handlers.