



# Signals and Alarms

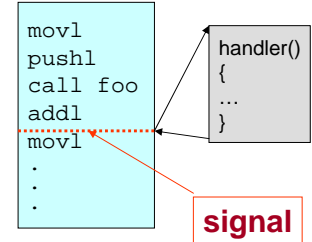
CS 217



# What Are Signals?

- Signals are notifications sent to a process
  - ^C, ^Z, Alarm, . . .
- Each signal may have a signal handler
  - When a signal is sent to a process, the OS stops the process immediately
  - Handler executes and finishes
  - Resume the process
- Signals are not interrupts
  - Interrupts are sent to OS by HW
  - Signals are sent to processes by OS
- Each UNIX signal has an integer number and a symbolic name
  - Defined in <signal.h>

A Process



# Some Predefined Signals

(/usr/include/bits/signum.h included by signal.h)

```

#define SIGHUP      1      /* Hangup (POSIX).  */
#define SIGINT      2      /* Interrupt (ANSI). */
#define SIGQUIT     3      /* Quit (POSIX).   */
#define SIGILL      4      /* Illegal instruction (ANSI). */
#define SIGTRAP     5      /* Trace trap (POSIX). */
#define SIGABRT     6      /* Abort (ANSI).   */
#define SIGFPE      8      /* Floating-point exception (ANSI). */
#define SIGKILL     9      /* Kill, unblockable (POSIX). */
#define SIGUSR1    10      /* User-defined signal 1 (POSIX). */
#define SIGSEGV    11      /* Segmentation violation (ANSI). */
#define SIGUSR2    12      /* User-defined signal 2 (POSIX). */
#define SIGPIPE    13      /* Broken pipe (POSIX). */
#define SIGALRM    14      /* Alarm clock (POSIX). */
#define SIGTERM    15      /* Termination (ANSI). */
#define SIGCHLD    17      /* Child status has changed (POSIX). */
#define SIGCONT    18      /* Continue (POSIX). */
#define SIGSTOP    19      /* Stop, unblockable (POSIX). */
#define SIGTSTP    20      /* Keyboard stop (POSIX). */
#define SIGTTIN    21      /* Background read from tty (POSIX). */
#define SIGTTOU    22      /* Background write to tty (POSIX). */
#define SIGPROF    27      /* Profiling alarm clock (4.2 BSD). */

```



# Predefined and Defined Signals

- Find out the predefined signals
 

```
% kill -l
```

```
% HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV
USR2 PIPE ALRM TERM STKFLT CHLD CONT STOP TSTP
TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR
SYS RTMIN RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2
RTMAX-1 RTMAX
```
- Applications can define their own signals
  - An application can define signals with unused values

## Catchable and Non-Catchable Signals



- Non-catchable signals
  - KILL
    - Terminate the process immediately
    - Catchable termination signal is TERM
  - STOP
    - Suspend the process immediately
    - Catchable suspension signal is TSTP
    - Can resume the process with signal CONT
- Catchable signals
  - All other predefined signals
  - All user-defined signals

5

## Sending Signals from Keyboard



- Steps
  - Pressing keys generates interrupts to OS
  - OS interprets a key sequence and sends a signal to the running process
- Examples
  - Ctrl-C causes the OS to send an INT signal to the running process.
    - By default, this causes the process to immediately terminate.
  - Ctrl-Z causes the OS to send a TSTP signal to the running process.
    - By default, this causes the process to suspend execution.
  - Ctrl-\ causes the OS to send a ABRT signal to the running process.
    - By default, this causes the process to immediately terminate.
- Question
  - Why do we have both Ctrl-C and Ctrl-\?

6

## Sending Signals From The Shell



- `kill -<signal> <PID>`
  - If no signal name or number is specified, the default is to send an SIGTERM signal to the process,
  - Signal SIGKILL or 9 is special; it cannot be caught
  - Example: send the INT signal to process with PID 1234:  
`kill -INT 1234`
    - The same affect as pressing Ctrl-C if process 1234 is running.
- `fg`
  - The command is “foreground”
  - On UNIX shells, this command will send a **CONT** signal
  - Resume execution of the process (that was suspended with Ctrl-Z or a command “**bg**”)
  - See man pages for **fg** and **bg**

7

## Sending Signals from a Program



- The kill command is implemented by a system call

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```
- Example: send a signal to itself

```
if (kill(getpid(), SIGABRT))
    exit(0);
```

  - The equivalent in ANSI C is:

```
int raise(int sig);

if (raise(SIGABRT) > 0)
    exit(1);
```

8

## Installing A Signal Handler



- Predefined signal handlers
  - `SIG_DFL`: default handler
  - `SIG_IGN`: Ignore the signal
- To install a handler, use

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

  - Handler `handler` will be invoked, when signal `sig` occurs
  - Return the old handler on success; `SIG_ERR` on error
  - On most non-Linux UNIX systems, after the handler executes, the OS resets the handler to `SIG_DFL`
- Example

```
#include <signal.h>
...
if (signal(SIGINT, SIG_IGN) == SIG_ERR)
    exit(1);
```

9

## Example: Catch INT Signal



```
#include <stdio.h>
#include <signal.h> /* signal interface */

void handler(int sig_num) {
    if (signal(SIGINT, catch_int) == SIG_ERR)
        ...;
    printf("Don't do that.\n");
    fflush(stdout);
}

main(void) {
    if (signal(SIGINT, handler) == SIG_ERR)
        exit(1);
    for ( ;; )
        pause();
}
```

10

## Example: Cleanup on Termination



```
#include <signal.h>

char *tmpfile = "temp.xxx";
void cleanup(int sig) {
    unlink(tmpfile);
    exit(1);
}

void main(void) {
    int fd;
    if (signal(SIGINT, cleanup) == SIG_ERR)
        fprintf(stderr, "cannot setup signal\n");
    fd = open(tmpfile, O_CREAT, 0666);
    ...
    close(fd);
}
```

11

## Masking Signals



- Why masking out signals?
  - An application wants to ignore certain signals
  - Avoid race conditions when another signal happens in the middle of the signal handler's execution
- Two ways to mask signals
  - Affect all signal handlers  
`sigprocmask()`
  - Affect a specific handler  
`sigaction()`

12

## Mask Signals for All Handlers



- Each Unix process has a signal mask in the kernel
  - OS use this mask to decide which signals to deliver
  - `sigprocmask()` takes a user-defined mask install it in the kernel, with some limitations
- Use `sigprocmask()`

```
#include <signal.h>
int sigprocmask(
    int how, /* SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
    */
    const sigset_t *set, /* set of signals */
    sigset_t *oldset /* set of old signals */
);
```

  - `SIG_BLOCK`: Add **set** to the current mask
  - `SIG_UNBLOCK`: Remove **set** from the current mask
  - `SIG_SETMASK`: Install **set** as the signal mask

13

## Example: Masking SIGINT Signal



```
#include <signal.h>
#define MYSIG 40

void handler(int sig) {
    sigset_t mask_set, old_set;
    signal(MYSIG, handler); /* reinstall handler */
    sigfillset(&mask_set); /* fill all signals */
    sigprocmask(SIG_SETMASK, &mask_set, &old_set);
    ...
}

main(void) {
    signal(MYSIG, handler);
    ...
}
```

- Anything wrong with this example?

14

## Install Handler and Mask Together



- Use `sigaction()` with a data structure

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

  - Use either `sa_handler` or `sa_sigaction`, but not both
  - Do not use `sa_restorer` (obsolete)

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

15

## Coarse-Grained Alarm



- Sends an `SIGALRM` signal after `n` seconds

```
unsigned int alarm(unsigned seconds);
```

  - This call may be different on other UNIX systems
- Example

```
#include <signal.h> /* signal names and API */
void catch_alarm(int sig) {
    if (signal(SIGALRM, catch_alarm) == SIG_ERR)
        ...;
    ...
    alarm(10);
}

main(void) {
    if (signal(SIGALRM, catch_alarm) == SIG_ERR)
        ...;
    alarm(10);

    ...;
}
```

16



## Fine-Grained Alarm

- Send an SIGALRM signal after a fine-grained timer expires

```
#include <sys/time.h>
int setitimer(
    int which, /* ITIMER_REAL, ITIMER_VIRTUAL, ITIMER_PROF */
    const struct itimerval *value,
    struct itimerval *ovalue
);
```
- Example

```
struct itimerval timer;

timer.it_interval.tv_sec = 0;
timer.it_interval.tv_usec = 10000; /* reload alarm 10ms */
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 10000; /* 10ms */
if (setitimer(ITIMER_PROF, &timer, NULL) == ...)
    ...;
```
- On Linux, the minimal effective granularity is **10ms**.

17



## Summary

- Signals
  - An asynchronous event mechanism, but not the only one
  - Use sigaction() to avoid race conditions
  - Signal handlers should be simple and short
  - Most predefined signals are catchable, but be careful with the “fault” signals (such as SIGSEGV).
- Alarms or timers
  - Use one timer at a time
  - Linux imposes 10ms as the minimal granularity

18