

Signals

Lecture 04.04

The O/S controls your program with signals

- A signal is a short message – just an integer value – which can be sent to a process by O/S
- When a signal arrives, the process has to stop whatever it is doing and deal with a signal. Signals interrupt normal process execution
- The process looks into a *mapping table* of 32 signal numbers and finds there instructions of how to handle each signal

Some signals (0-31) and their default handling (defined in <signal.h>)

SIGINT 2 /* Interrupt from keyboard (happy termination): ctrl+C */

SIGQUIT 3 /* Quit from keyboard, dump memory (unhappy termination): ctrl-\ */

SIGKILL 9 /* hard kill. The only behavior is to kill the process, immediately. No cleanup, and thus this is a signal of last resort. */

SIGUSR1 10 /* left for programmers to do whatever they want*/

SIGUSR2 12 /* left for programmers to do whatever they want*/

SIGALRM 14 /* alarm clock – i.e. timer */

SIGTERM 15 /*kill the process, gracefully or not, but allow it a chance to cleanup*/

SIGCONT 18 /* continue a stopped process */

SIGSTOP 19 /* The only behavior is to pause the process; The shell uses pausing (and its counterpart, resuming via SIGCONT) to implement job control.*/

SIGCHLD 20 /* to parent on child stop or exit */

To know more: [man 7 signal](#)

Some signals (0-31) and their default handling (defined in <signal.h>)

SIGINT 2

SIGQUIT 3

SIGKILL 9

SIGUSR1 10

SIGUSR2 12

SIGALRM 14

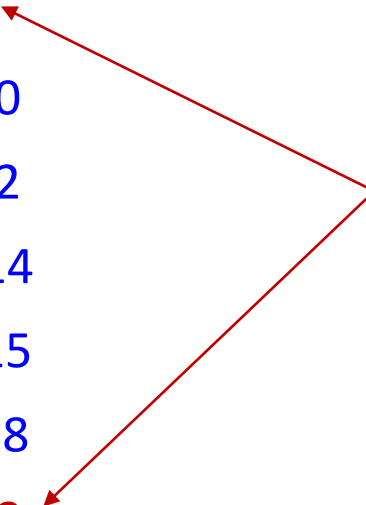
SIGTERM 15

SIGCONT 18

SIGSTOP 19

SIGCHLD 20

These two signals cannot have a different behavior – they always kill or stop the process



The reaction to all other signals can be re-defined in your C program

Sending signal to a process from command-line

To see all running processes (belonging to *user*) and their pids:

```
ps (-u user)
```

```
kill pid (kill defaults to sending a SIGTERM)
```

```
kill -9 pid (hard kill, no escape)
```

```
kill 19 pid
```

```
kill -SIGSTOP pid
```

```
kill -STOP pid
```

```
pkill -STOP pname
```

Redefining signal handler: SIGINT

- The default signal handler for the interrupt signal just calls the *exit()* function
- The signal table lets you run your own code when your process receives a signal
- For example, if your process has files or network connections open, it might want to close things down and tidy up before exiting

Example: replace default behavior with *sigaction*

- For example, you want O/S to call a function called *diediedie()* if someone sends an interrupt signal to your process

Example 1/3: Define a new handler function of type `void f (int)`

```
void diediedie (int sig) {  
    puts ("Goodbye cruel world....\n");  
    exit(1);  
}
```


Example 2/3: Set fields in a variable of type struct sigaction

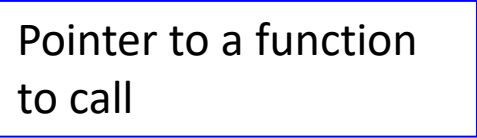
```
struct sigaction action;
```

```
action.sa_handler = diediedie;
```

```
sigemptyset(&action.sa_mask);
```

```
action.sa_flags = 0;
```

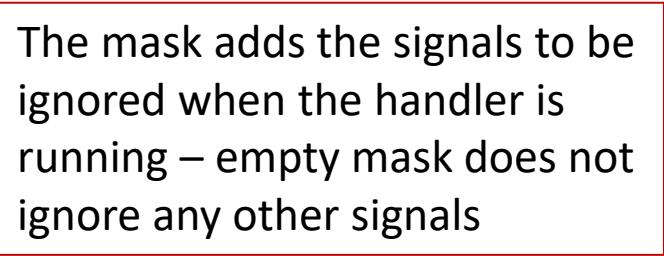
Pointer to a function to call



Some additional flags



The mask adds the signals to be ignored when the handler is running – empty mask does not ignore any other signals



Example 3/3: sigactions are registered with sigaction()

```
sigaction (signal_no, &new_action, &old_action);
```

- **signal_no** - the integer value of the signal you want to handle. Usually, you'll pass one of the standard signal constants, like SIGINT or SIGQUIT
- **new_action** - the address of the new sigaction you want to register (that we just created)
- **old_action** - if you pass a pointer to another sigaction, it will be filled with details of the current handler that you're about to replace. If you don't care about the existing signal handler, you can set this to NULL

Function for registering custom signals

```
int catch_signal(int sig, void (*handler)(int)) {  
    struct sigaction action;  
    action.sa_handler = handler;  
    sigemptyset(&action.sa_mask);  
    action.sa_flags = 0;  
    return sigaction (sig, &action, NULL);  
}
```

- This function will allow you to set a signal handler by calling `catch_signal()` with a signal number and a function name:

```
catch_signal (SIGINT, diediedie)
```

Summary:

installing custom signal handler

1. Write a new function *handler* that returns void and has a single int as a parameter:

```
void handler (int sig_num);
```

1. Declare and initialize a new variable of type `struct sigaction`
2. Register your new handler using function `sigaction ()`

Ignoring signals

- Ignoring
- Blocking

Ignoring signals: not even receiving a signal

```
struct sigaction action;  
action.sa_handler = SIG_IGN;  
sigaction (SIGINT, &action, NULL);
```

Blocking signals with *sigaction*

- Sometimes you want to block other signals from interrupting your handler function while it is handling the current signal
- That way you can have your signal handler modify some non-atomic state (say, a counter of how many signals have come in) in a safe way
- So *sigaction* takes a **mask of signals** it should block while the handler is executing

Blocking other signals while handler is running

```
struct sigaction action;  
action.sa_handler = &my_handler;  
sigemptyset(&action.sa_mask);  
sigaddset(&action.sa_mask, SIGINT);  
sigaddset(&action.sa_mask, SIGTERM);  
sigaction(SIGINT, &action, NULL);
```

- Here, we're masking both SIGINT and SIGTERM: if either of these signals comes in while *my_handler* is running, they'll be blocked until it completes.

Exercise 1: greeting

Infinite loop

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

int main(int argc, char **argv) {
    for (;;) {
    }
    return 0;
}
```

Moving processes to the background and foreground

Exercise 1: greeting

```
./greeting
```

```
ps
```

```
./greeting &
```

```
bg pid
```

```
kill -STOP pid
```

```
kill -CONT pid
```

Exercise 1: greeting

Killing the process: many ways

kill -KILL pid #kill hard

kill -STOP pid

kill -TERM pid

kill -INT pid

kill pid #default - SIGTERM

pkill greeting

Signal handler: always *void f (int)*

```
void sing (int sig) {  
    puts ("Happy birthday to you,");  
    puts ("Happy birthday to you,");  
  
    puts ("Happy birthday to you,");  
    puts ("Happy birthday to you");  
}
```

- How to make it to print a name?

Exercise 1: greeting

Install new signal handler

```
int catch_signal(int sig, void (*handler)(int)) {
    struct sigaction action;
    action.sa_handler = handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    return sigaction (sig, &action, NULL);
}
```

```
catch_signal (SIGUSR1, sing);
```

Test: compile and run

`ps`

`kill -USR1 pid`

Exercise 1: greeting

Making handler slower: sleep

Exercise 1: greeting

```
char * name;
void sing (int sig) {
    puts ("Happy birthday to you,");
    puts ("Happy birthday to you,");
    sleep (20);
    printf ("Happy birthday, dear %s,\n", name);
    puts ("Happy birthday to you");
}
```


Exercise 1: greeting

Blocking SIGINT while singing

```
int catch_signal_nointerrupt(int sig, void (*handler)(int)) {  
    struct sigaction action;  
    action.sa_handler = handler;  
    sigemptyset(&action.sa_mask);  
    sigaddset(&sa.sa_mask, SIGINT);  
    action.sa_flags = 0;  
    return sigaction (sig, &action, NULL);  
}
```

```
catch_signal_nointerrupt (SIGUSR1, sing);
```

Blocking signals in critical sections of code

- You might have a critical section where you don't want to be interrupted, but afterwards you want to know what came in
- You can block and unblock signals at any time using `sigprocmask`

Blocking/unblocking signals in code

```
catch_signal (SIGUSR1, sing);
```

```
sigset_t sigset; ← new sig_set
```

```
sigemptyset(&sigset);
```

```
sigaddset(&sigset, SIGINT); ← Set signals we want to intercept
```

```
printf("Blocking signals...\n");
```

```
sigprocmask (SIG_BLOCK, &sigset, NULL);
```

```
// Critical section
```

```
sleep(5);
```

```
printf("Unblocking signals...\n");
```

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

We can block all the signals at once
(except SIGKILL and SIGSTOP)

```
int main() {  
    sigset_t block_set;  
    sigfillset(&block_set); //fills in all possible signals  
    sigprocmask(SIG_BLOCK, &block_set, NULL);  
    while (1);  
}
```

Adding to Exercise 1

- Let's add to our program *greeting* to demonstrate using *sigprocmask*
- Let's say that our program is busy studying for 30 seconds, and during this time it cannot sing
- After 30 seconds it takes a break and can sing for about 20 seconds.

Code in `greeting_extended.c`

Example: blocking/unblocking

```
for (;;) {  
    puts("Busy studying! Go away.");  
    // Don't be interrupted by SIGUSR1.  
    sigset_t block_set;  
    sigemptyset(&block_set);  
    sigaddset(&block_set, SIGUSR1);  
    sigprocmask (SIG_BLOCK, &block_set, NULL);  
        sleep(30);  
    printf("Okay I can party now.\n");  
    sigprocmask (SIG_UNBLOCK, &block_set, NULL);  
        sleep(20);  
}
```

Using *raise()* to raise signals inside the same process

- Sometimes you might want a process to send a signal to itself, which you can do with the *raise()* command:

```
raise(SIGUSR1);
```

- Normally, the *raise()* command is used inside your own custom signal handlers. It means your code can receive a signal for something minor and then choose to raise a more serious signal
- This is called *signal escalation*
- Another way to send signal to the same process:

```
kill (getpid(), SIGUSR1);
```

Using signals for communication between parent and child

```
if ((pid = fork()) == 0) {    /* child */
    catch_signal (SIGUSR1, sing); //install signal handler
    for(;;);                // loop for ever
}
else {                       /* parent */
    kill (pid,SIGUSR1);      // pid holds id of child
    sleep(3);
}
```


Exercise 2: signals and fork