

Signals

Lecture 07.01

Demo code:

https://src-code.simons-rock.edu/git/mbarsky/signals_demo.git

- Clone the repository
- Compile dots.c into executable called dots
- Run the program
- Press CTRL+C
- What do you think happened?

Signals table

- For each process, in addition to
 - Process statistics
 - Memory allocation
 - File Descriptors table

operating system stores

- **Signals table**
- The OS is constantly running an event loop to detect any of the user signals and act according to the table

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 for the instructions of how to handle each signal

Signals mapping table

<u>Signal</u>	<u>Value</u>	<u>Action</u>	<u>Comment</u>
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGABRT	6	Core	Abort signal from abort(3)
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process

To see running processes

`ps`

`ps aux`

`a` = show processes for all users

`u` = display the process's user/owner

`x` = also show processes not attached to a terminal

belonging to *user*:

`ps -u user`

by program name:

`ps aux | grep dots`

Sending signal to a process

- Terminology for delivering signals is to "kill" a process with the *kill* command
- The *kill* command is poorly named — originally, it was only used to kill or terminate a process, but it is currently used to send any kind of signal to a process
- The difference between *kill* and *killall* is that *kill* only sends signals to process identified by their pid, *killall* sends the signal to all process of a given name

```
kill -<SIGNAL> pid
```

```
killall -<SIGNAL> program_name
```

Sample signals which terminate the program

- SIGALRM
- SIGINT
- SIGKILL
- SIGUSR1
- SIGSEGV
- ...

Signal mapping table

Signal ID	Constant name	Default action
2	SIGINT	_exit()
9	SIGKILL	_exit()
10	SIGUSR1	print message, _exit()
11	SIGSEGV	print message, _exit()
14	SIGALRM	print message, _exit()
19	SIGSTOP	_stop()

How the system behaves in response to these signals can be **re-defined** in your C program

Signal mapping table

Signal ID	Constant name	Default action
2	SIGINT	_exit()
9	SIGKILL	_exit()
10	SIGUSR1	print message, _exit()
11	SIGSEGV	print message, _exit()
14	SIGALRM	print message, _exit()
19	SIGSTOP	_stop()

How the system behaves in response to all signals can be **re-defined** in your C program

EXCEPT SIGKILL and SIGSTOP

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:

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

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

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

Step 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



Step 3/3: register new signal handler with *sigaction()*

struct of type *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

Step 3/3: register new signal handler with *sigaction()*

```
sigaction (SIGINT, &action, NULL);
```

Now compile dots2.c, run, and press CTRL+C:

```
gcc dots2.c -o dots2 & ./dots2
```


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

action.sa_mask

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

killall 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?

Install new signal handler

Exercise 1: greeting

```
struct sigaction action;  
action.sa_handler = sing;  
sigemptyset(&action.sa_mask);  
action.sa_flags = 0;  
  
sigaction (SIGUSR1, &action, NULL);
```

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");
}
```

Blocking SIGINT while singing

Exercise 1: greeting

...

```
sigemptyset(&action.sa_mask);  
sigaddset(&sa.sa_mask, SIGINT);
```

...

SIGPROCMASK

Blocking-unblocking

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

```
//install signal handler
```

```
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);  
}
```

Code in unbreakable.c

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

Demo:
signals and fork

Code in: *pocking.c*

- Write a program that forks two children.
- One child sends a SIGUSR1 signal to the other child approximately every 2 seconds (use sleep(2) between signals.).
- The other child does nothing except print out numbers from 1 to 1000. But every time a SIGUSR1 signal arrives, it prints "quit poking me" to standard error.
- When it's counting is finished, this child exits with the number of times it was poked as the exit code.
- The parent then prints the number of pokes to the standard error.