Lesson 3

Signals:

When a process terminates abnormally, it usually tries to send a signal indicating what went wrong. C programs can trap these for diagnostics.

Software interrupts: Stop executing the main program, activate the program that deals with the event and then return to the execution of the program. The program that handles the interrupt should be relatively short.

In Unix there are 31signals. Each process has algorithms associated with each signal. Each process has a table with 31 entries. Each entry contains the function that needs to be done for each signal. Most of the functions, by default, kills the process after the signal (void f() { exit() }). Some of the signals create a ***core*** file, in which they save the memory image and some other data (in a binary form), which enable to analyze and to know why the process was terminated.
The process can *change* the signal table (write other functions to handle certain signals):

The default action for each signal is one of the following:
1. Terminate the process.
2. Create a core image and then terminate the process.
3. Stop the process.
4. Discard the signal.

SYNOPSIS
#include <signal.h>

void *signal(int *sig_num*, *func*());

void (*func)(*sig*)

DESCRIPTION
Except for the SIGKILL and SIGSTOP signals, the **signal**() function allows for a signal to be caught, to be ignored, or to generate an interrupt (if you try to do *signal(SIGKILL, func);* it will execute the default handler for SIGKILL and not func). The signals are defined in the file <*signal.h*>. Some of them are listed below:

Name	Default Action	Description	No.
SIGHUP	terminate process	hang up	1
SIGINT	terminate process	interrupt program	2
SIGQUIT	create core image	quit program	3
SIGILL	create core image	illegal instruction	4
SIGFPE	create core image	floating-point exception	8
SIGKILL	terminate process	kill program **(cannot be caught or ignored)**	9
SIGBUS	create core image	bus error	10
SIGSEGV	create core image	segmentation violation	11
SIGSYS	create core image	system call given invalid argument	12
SIGPIPE	terminate process	write on a pipe with no reader	13
SIGALRM	terminate process	real-time timer expired	14
SIGSTOP	stop process	stop **(cannot be caught or ignored)**	17
SIGCONT	continue process	continue a stopped process	19
SIGCHLD	discard signal	child status has changed	20
SIGUSR1	terminate process	User defined signal 1	30
SIGUSR2	terminate process	User defined signal 2	31

- The *func* procedure allows a user to choose the action upon receipt of a signal.

- To set the default action of the signal to occur as listed above, *func* should be **SIG_DFL**. A SIG_DFL resets the default action.

- ***Blocking Signals:***

1) ***SIG_IGN:***
To ignore the signal *func* should be **SIG_IGN**. This will cause subsequent instances of the signal to be ignored and pending instances to be discarded. If SIG_IGN is not used, further occurrences of the signal are automatically blocked and *func* is called. The handled signal is unblocked with the function returns and the process continues from where it left off when the signal occurred.
Example:
```
// signal SIGILL is ignored (blocked). The signal remains ignored even after calling it.
signal(4, SIG_IGN);
// returning to the default handler:
signal(4, SIG_DFL);
```

2) ***sigblock:*** With this function we just block the signal, without changing the handler of it. When we unblock it we return to the handler function of the signal.

SYNOPSIS
```
#include <signal.h>

int sigblock(int mask);
int sigmask(int signum);
```

DESCRIPTION
sigblock() adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signals are blocked if the corresponding bit in *mask* is a 1;
the macro sigmask() is provided to construct the mask for a given *signum*.
It is not possible to block SIGKILL or SIGSTOP.

RETURN VALUES
The previous set of masked signals is returned.

Example:
```
sigblock(sigmask(2)); // blocking signal SIGINT
sigblock(sigmask(2) | sigmask(4)); // blocking signals SIGINT and SIGILL
sigblock(0); // unblock all signals
```

After a signal has been delivered to the handler func(), the handler returns to its default. In order to keep func() all the time, we need to call signal() again in func(), for example:
```
main()
{
        …
        signal(3, &func);
        …
}
void func()
{
        …
        signal(3, &func);
        …
}
```

When a process, which has installed signal handlers forks, the child process inherits the signals. All caught signals may be reset to their default action by a call to the ***exceve*** function; ignored signals remain ignored.

Usually when a process receives signal during a system call, the process continues the system call execution until it finishes and only then handles the signal. read, wait, open and wait ***do*** stop upon receiving a signal. For them the return value is -1 and errno = EINTR.

You cannot "stack" signals. In case 2 or more of the same signal are received → only the last one is handled.

RETURN VALUES
The previous action is returned on a successful call (a pointer to the handler function).
Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

Sources of Interrupts:
1. **Kernel:** The kernel sends signal to process when:
   a. Illegal command (for example, division by 0 → FPE signal)
   b. Illegal address (SEGV)
2. **Shell:** Using ***kill*** command from the shell:
   % kill -signal pid for example: % kill -11 1000 // sending SEGV signal to pid 1000
3. **Processes:** Process can send signal to another process using the ***kill*** system call:

SYNOPSIS
#include <sys/types.h>
#include <signal.h>

int kill(*pidt pid*, *int sig*);

DESCRIPTION
The kill() function sends the signal given by *sig* to *pid*, a process or a group of processes. *Sig* may be one of the signals or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.
For a process to have permission to send a signal to a process designated by *pid*, the real or effective user ID of the receiving process must match that of the sending process or the user must have appropriate privileges. A single exception is the signal SIGCONT, which may always be sent to any descendant of the current process.
If *pid* is greater than zero: *Sig* is sent to the process whose ID is equal to *pid.*
If *pid* is zero: *Sig* is sent to all processes whose group ID is equal to the process group ID of the sender, and for which the process has permission.
If *pid* is -1: If the user has superuser privileges, the signal is sent to all processes excluding system processes and the process sending the signal. If the user is not the super user, the signal is sent to all processes with the same uid as the user excluding the process sending the signal. No error is returned if any process could be signaled.
If the process number is negative but not -1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number.

RETURN VALUES
Upon successful completion, a value of 0 is returned.
Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

Summation: A process can:
1. Determine what to do upon receiving a signal (using default action or defining functions).
2. Ignore signals (continue the execution of the process).
3. Send signals to other processes.

A process can send a signal to itself, for example:
1) ***raise:***
SYNOPSIS
#include <signal.h>

int raise(*int sig*);

DESCRIPTION
The **raise**() function sends the signal *sig* to the current process. It's equivalent to: *kill(getpid(), sig);*

RETURN VALUES
Upon successful completion, a value of 0 is returned.
Otherwise, a value of -1 is returned and the global variable *errno* is set to indicate the error.

2) ***abort:***
SYNOPSIS
#include <stdlib.h>

void abort(*void*);

DESCRIPTION
The **abort**() function causes abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return. It's equivalent to: *raise(SIGABRT);*

RETURN VALUES
The **abort** function never returns.

pause: stop until a signal.

SYNOPSIS
#include <unistd.h>

int pause(*void*);

DESCRIPTION
The **pause()** function forces a process to pause until a signal is received from either the ***kill*** function or an interval timer. Upon termination of a signal handler started during a **pause()**, the **pause()** call will return. If the signal causes termination of the calling process, **pause()** does not return.

RETURN VALUES
Always returns -1 and *errno* equals EINTR (the call was interrupted).

alarm: set signal timer alarm

SYNOPSIS
#include <unistd.h>

unsigned int alarm(*unsigned int seconds*);

DESCRIPTION
The **alarm()** function instructs the alarm clock of the calling process to send the signal SIGALRM to the calling process after the number of real time *seconds* have elapsed.
Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.
If sec is 0, any previously made alarm request is canceled.
For example:
alarm(20);
alarm(5);
will generate signal after 5 seconds.

The **fork()** function sets the alarm clock of a new process to 0. A process created by the exec family of routines inherits the time left on the old process's alarm clock.

RETURN VALUES
The **alarm()** function returns the amount of time previously remaining in the alarm clock of the calling process.

Examples:

Kernel Interrupts:

(sig3_1.c)

```
#include <sys/types.h>
main()
{
        int* a = NULL;
        *a = 5;
}
1 > gcc -o sig1 3_1.c
2 > sig1
Segmentation fault (core dumped)
// the process terminated due to segmentation violation and a core file was generated.
```

(sig3_2.c)

```
#include <signal.h>

void trap();

main()
{
        int *a = NULL;
        signal(SIGSEGV, &trap);
        *a = 5;
}

void trap()
{
        printf("segmentation fault trapped!\n");
        exit(1);
}
1 > gcc -o sig2 sig3_2.c
2 > sig2
segmentation fault trapped!
```

(sig3_3.c)

```
main()
{
        int a = 1/0;
}
1 > gcc -o sig3 sig3_3.c
2 > sig3
Floating exception
```

(sig3_4.c)

```
#include <signal.h>

void handle();

main()
{
        int a = 5;
        signal(SIGFPE, &handle);
        a = 1/0;
        exit(0);
}
```

```
void handle()
{
        printf("FPE signal trapped!\n");
        exit(1);
}
1 > gcc -o sig4 sig3_4.c
2 > sig4
FPE signal trapped!
```

Shell Interrupts:

(sig3_5.c)

```
void trap();

main()
{
        signal(11, &trap);
        while(1);
}

void trap()
{
        printf("segmentation fault trapped!\n");
        exit(1);
}

2 > gcc -o sig5 sig3_5.c
3 > sig5 &
[1] 32036
4 > kill -SEGV 32036       // =  kill -11 32036
segmentation fault trapped!
[1]        Exit 1     sig5
```

Interrupts from process:

(sig3_6.c)

```
#include <signal.h>
void trap();

main()
{
        int pid, *a = NULL;
        signal(SIGSEGV, &trap);
        pid = getpid();
        kill(pid, SIGSEGV); // the process sends signal to itself
        // equivalents to:  raise(SIGSEGV);
        exit(0);
}

void trap()
{
        printf("segmentation fault trapped!\n");
        exit(1);
}
2 > gcc -o sig6 sig3_6.c
3 > sig6
segmentation fault trapped!
```

(sig3_7.c)

```
#include <signal.h>
main()
{
        sigblock(sigmask(SIGSEGV));
        while(1);
}
2 > gcc -o sig7 sig3_7.c
3 > sig7 &
[1] 32580
4 > jobs
[1]     + Running       sig7
5 > kill -SEGV 32580
6 > jobs
[1]     + Running       sig7    // the process still runs since we blocked SIGSEGV
7 > kill -KILL 32580
4 > jobs
[1]     Killed  sig7
```

More Examples:

(sig3_8.c)

```
#include <signal.h>

void father_function();

main()
{
        pid_t pid;
        int s;

        if (fork() == 0)
        { // child process
                pid = getppid();
                kill(pid, SIGUSR1);
        }
        else // parent process
        {
                signal(SIGUSR1, &father_function);
                wait(&s);
        }
}

void father_function()
{
        printf("I heard you my poor son\n");
        exit(1);
}

1 > gcc -o sig8 sig3_8.c
2 > repeat 10 sig8
I heard you my poor son
I heard you my poor son
I heard you my poor son
User signal 1
User signal 1
User signal 1
User signal 1
User signal 1
User signal 1
User signal 1
```

(sig3_9.c)

```
#include <signal.h>

void son_function();
main()
{
        pid_t pid;
        int s;

        if ( (pid = fork()) == 0)      // child process
        {
                signal(SIGUSR1, &son_function);
                while(1);
        }
        else // parent process
        {
                kill(pid, SIGUSR1);
                wait(&s);
        }
}
void son_function()
{
        printf("I heard you my poor father\n");
        exit(1);
}
```

```
3 > gcc -o sig9 sig3_9.c
4 > repeat 5 sig9
I heard you my poor father
I heard you my poor father
I heard you my poor father
User signal 1
User signal 1
```

(sig3_10.c)

```
#include <signal.h>

void father_function();
main()
{
        pid_t pid;
        int s;

        signal(SIGUSR1, &father_function);

        if (fork() == 0)
        { // child process
                pid = getppid();
                kill(pid, SIGUSR1);
        }
        else // parent process
                wait(&s);
}
void father_function()
{
        printf("I heard you my poor son\n");
        exit(1);
}
```

```
5 > gcc -o sig10 sig3_10.c
6 > sig10
I heard you my poor son
```

(sig3_11.c)

```
#include <signal.h>

void funct()
{
	printf("haha\n");
	exit(1);
}

main()
{
	int sig=SIGSEGV;
	char *a;

	signal(sig, &funct);
	signal(sig, SIG_DFL );

	*a=0;
}

7 >gcc -o sig11 sig11.c
8 >sig11
Segmentation fault (core dumped)		// default behavior
```

(sig3_12.c)

```
#include <signal.h>

void funct()
{
	printf("haha\n");
	exit(1);
}

main()
{
	int sig=SIGSEGV;

	signal(sig, SIG_IGN );

	while(1);
}

9 >gcc -o sig12 sig12.c
10 >sig12 &
[1] 20355
11 >kill -SEGV 20355
12 >kill -KILL 20355
[1]	Killed	sig12
```