Kontola signala

- UNIX guru W. Richard Stevens aptly describes signals as software interrupts.

  When a signal is sent to a process or thread, a signal handler may be entered (depending on the current disposition of the signal), which is similar to the system entering an interrupt handler as the result of receiving an interrupt.

- An application program executes sequentially if every instruction runs properly. In case of an error or any anomaly during the execution of a program, the kernel can use signals to notify the process.

  Signals also have been used to communicate and synchronize processes and to simplify interprocess communications (IPCs).

Kontola signala

- A signal is generated when an event occurs, and then the kernel passes the event to a receiving process.

  Sometimes a process can send a signal to other processes. Besides process-to-process signaling, there are many situations when the kernel originates a signal, such as when file size exceeds limits, when an I/O device is ready, when encountering an illegal instruction or when the user sends a terminal interrupt like Ctrl-C or Ctrl-Z.

- Every signal has a name starting with SIG and is defined as a positive unique integer number.

  1 to 31 - standard signals

  32 to 63 - a new class of signals designated as real-time signals

```
kill -l
 display all signals with signal number and corresponding
 signal name.
/usr/include/bits/signum.h  - signal numbers
/usr/src/linux/kernel/signal.c - source file
```

Kontola signala

When a process receives a signal, one of three things could happen:

1. the process could ignore the signal.
2. it could catch the signal and execute a special function called a signal handler.
3. it could execute the default action for that signal

   There are four possible default dispositions:

   - Exit: forces the process to exit.
   - Core: forces the process to exit and create a core file.
   - Stop: stops the process.
   - Ignore: ignores the signal; no action taken.

- `signal(7)` - page for a reference list of signal names, numbers, default actions and whether they can be caught.

Kontola signala

Name	Number	Default Action	Description
SIGHUP	1	Exit	Hangup (ref termio(7I))
SIGINT	2	Exit	Interrupt (ref termio(7I))
SIGQUIT	3	Core	Quit (ref termio(7I))
SIGILL	4	Core	Illegal Instruction
SIGFPE	8	Core	Arithmetic exception
SIGKILL	9	Exit	Kill
SIGBUS	10	Core	Bus error--a misaligned address error
SIGSEGV	11	Core	Segmentation fault, an address reference boundary error
SIGSYS	12	Core	Bad system call
SIGCHLD	18	Ignore	Child process status changed
SIGPWR	19	Ignore	Power fail or restart
SIGSTOP	23	Stop	Stop (cannot be caught or ignored)
SIGTSTP	24	Stop	Stop (job control, e.g., CTRL-z))
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped--tty input (ref termio(7I))
SIGTTOU	27	Stop	Stopped--tty output (ref termio(7I))
SIGXFSZ	31	Core	File size limit exceeded (ref getrlimit(2))
SIGRTMIN	38	Exit	Highest priority real-time signal
SIGRTMAX	45	Exit	Lowest priority real-time signal

signal()

The POSIX standards provided a set of interfaces for using signals in code, and today the Linux implementation of signals is fully POSIX-compliant.

Reliable signals require the use of the newer sigaction interface, as opposed to the traditional signal call.

Synopsis

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

ili kraće

```
void (*signal(int sig, void (*func)(int)))(int);
```

Description

The `signal()` system call installs a new signal handler for the signal with number `signum`. The signal handler is set to sighandler which may be a user specified function, or either **`SIG_IGN`** or **`SIG_DFL`**.

Pr.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
int main(void)
{ void sigint_handler(int sig); /* prototype */
  char s[200];
  if (signal(SIGINT, sigint_handler) == SIG_ERR) {
     perror("signal");
     exit(1);}
  printf("Enter a string:\n");
  if (gets(s) == NULL) perror("gets");
   else
   printf("You entered: \"%s\"\n", s);
   return 0;
}
void sigint_handler(int sig) {
   printf("Not this time!\n");
}
```

Pr.2.

```
#include <signal.h>
void my_handler (int sig); /* function prototype */
int main(void) {
/* Part I: Catch SIGINT */
    signal (SIGINT, my_handler);
    printf ("Catching SIGINT\n");
    sleep(3);
    printf (" No SIGINT within 3 seconds\n");
/* Part II: Ignore SIGINT */
    signal (SIGINT, SIG_IGN);
    printf ("Ignoring SIGINT\n");
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");
/* Part III: Default action for  SIGINT */
    signal (SIGINT, SIG_DFL);
    printf ("Default action for SIGINT\n");
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");
    return 0;
}
/* User-defined signal handler function */
void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);
    exit(0);
}
```

Pr.3.

ŠALJE SIGNAL

```
#include <signal.h>

main ( ) {
    int process_id;
    printf ("Enter process_id which you want to send a signal : ");
    scanf ("%d", &process_id);

    if (!(kill ( process_id, SIGINT)))
       printf ("SIGINT sent to %d\n", process_id);
    else if (errno == EPERM) printf ("Operation not permitted.\n");
            else printf ("%d doesn't exist\n", process_id);
}
```

PRIMA SIGNAL

```
/* Listing 4a. This program will run until it receives SIGINT */
#include <signal.h>

 main ( ) {
   printf (" This process id is %d. "
   "Waiting for SIGINT.\n", getpid());
   for (;;);
}
```

UBACITI KONTROLU SIGNALA SIGINT