Sharif-OS-Lab /
summer1403-5-99101087_99100422

Code Issues 2 Pull requests Actions Projects Security Insights

Edit New issue Jump to bottom

Session 5 Report #1

Open 22 tasks done Amirreza81 opened this issue 5 days ago · 0 comments

Assignees

Labels documentation

Amirreza81 commented 5 days ago • edited

Team Name: `99101087-99100422`

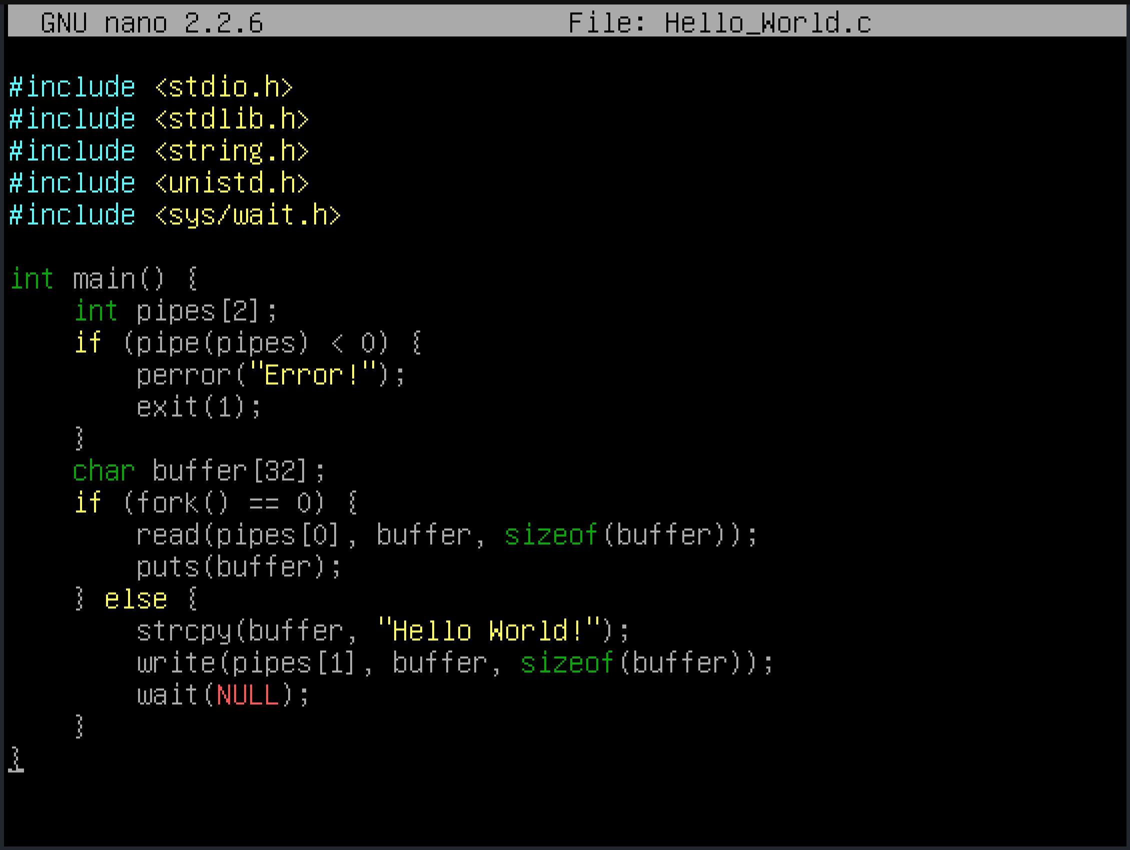
Student Name of member 1: `AmirReza Azari`
Student No. of member 1: `99101087`

Student Name of member 2: `Bozorgmehr Zia`
Student No. of member 2: `99100422`

- ☑ Read Session Contents.

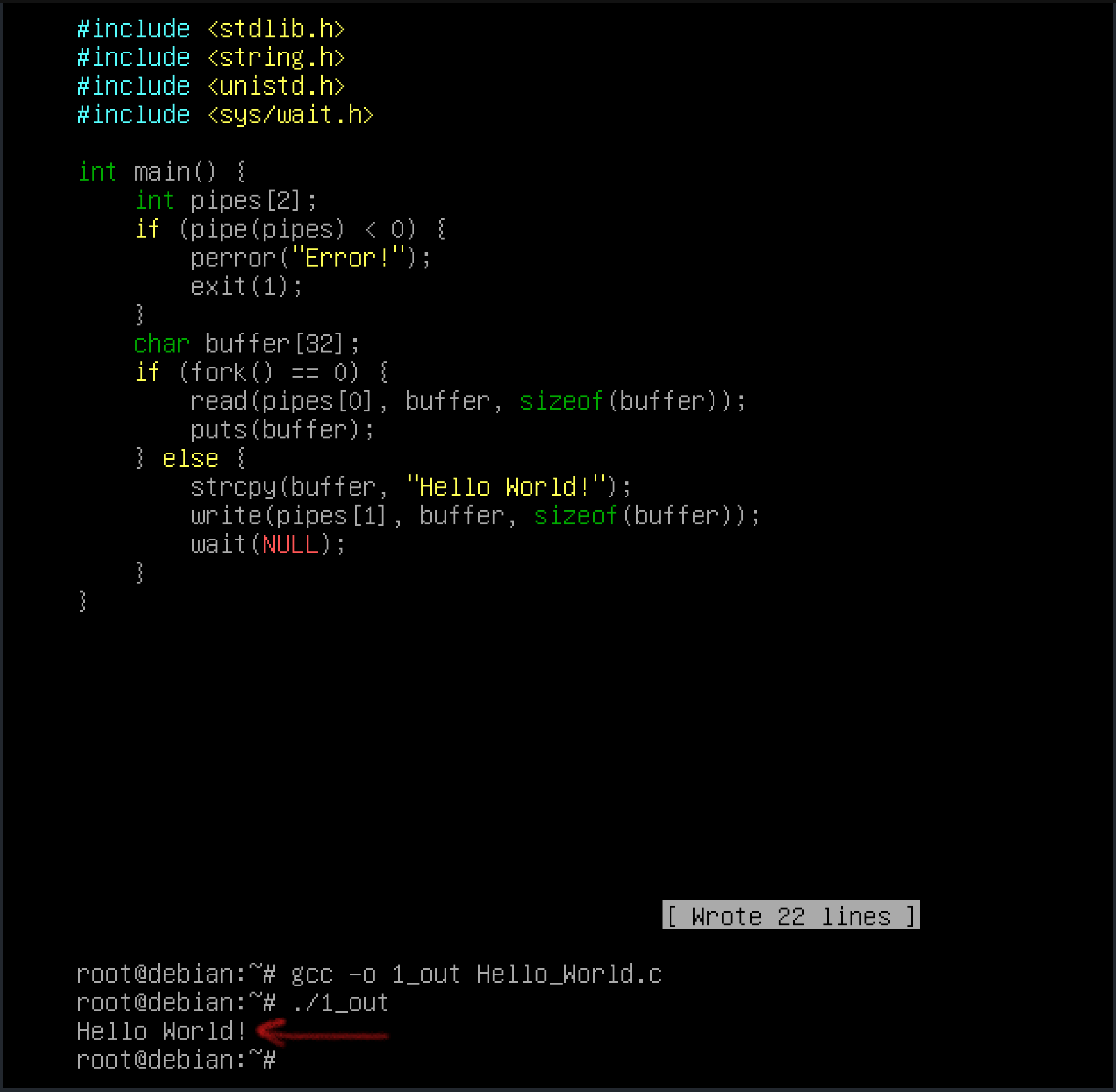
Section 5.3.1

- ☑ Write the `Hello World!` program
  - ☑ `FILL HERE with your source code`
    The program is here in Hello_World.c:



```
  GNU nano 2.2.6                    File: Hello_World.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pipes[2];
    if (pipe(pipes) < 0) {
        perror("Error!");
        exit(1);
    }
    char buffer[32];
    if (fork() == 0) {
        read(pipes[0], buffer, sizeof(buffer));
        puts(buffer);
    } else {
        strcpy(buffer, "Hello World!");
        write(pipes[1], buffer, sizeof(buffer));
        wait(NULL);
    }
}
```

Then we have:

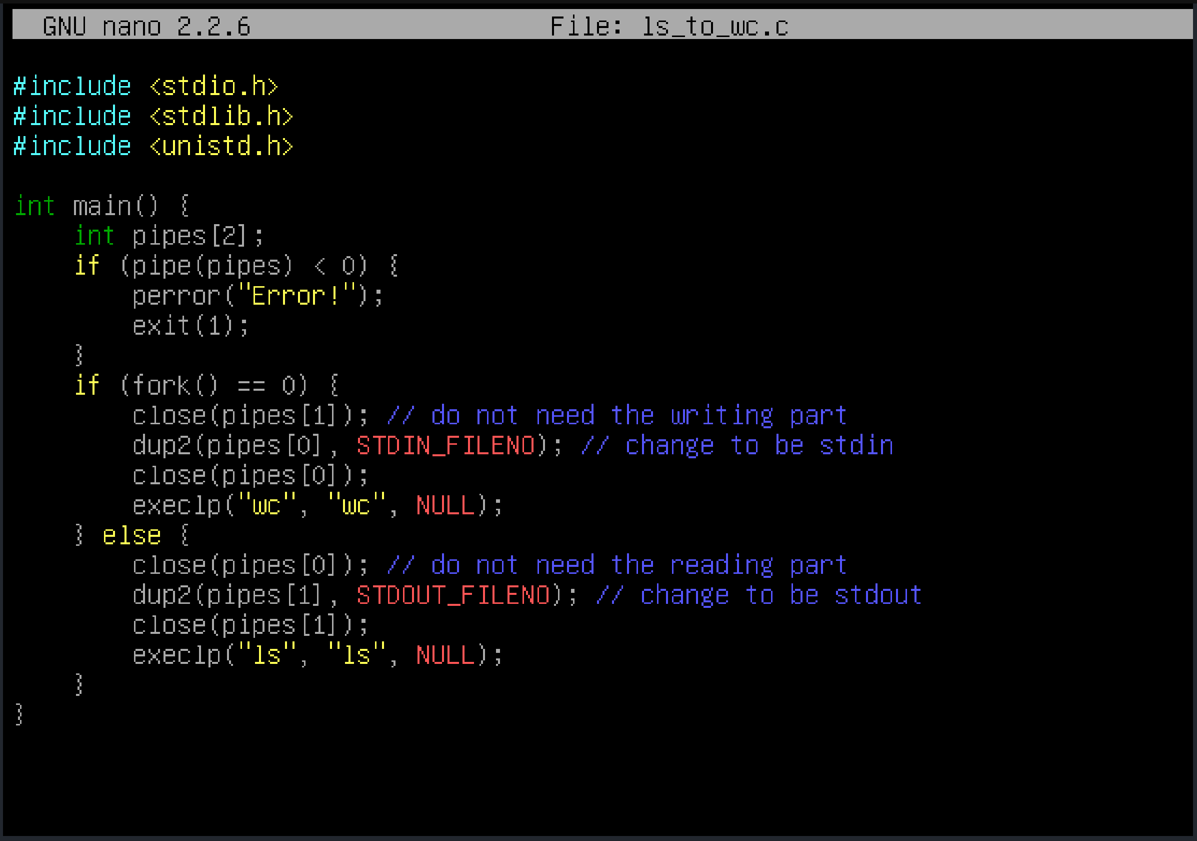


```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pipes[2];
    if (pipe(pipes) < 0) {
        perror("Error!");
        exit(1);
    }
    char buffer[32];
    if (fork() == 0) {
        read(pipes[0], buffer, sizeof(buffer));
        puts(buffer);
    } else {
        strcpy(buffer, "Hello World!");
        write(pipes[1], buffer, sizeof(buffer));
        wait(NULL);
    }
}

                                                  [ Wrote 22 lines ]

root@debian:~# gcc -o 1_out Hello_World.c
root@debian:~# ./1_out
Hello World!
root@debian:~#
```

☑ Write the `ls` to `wc` program

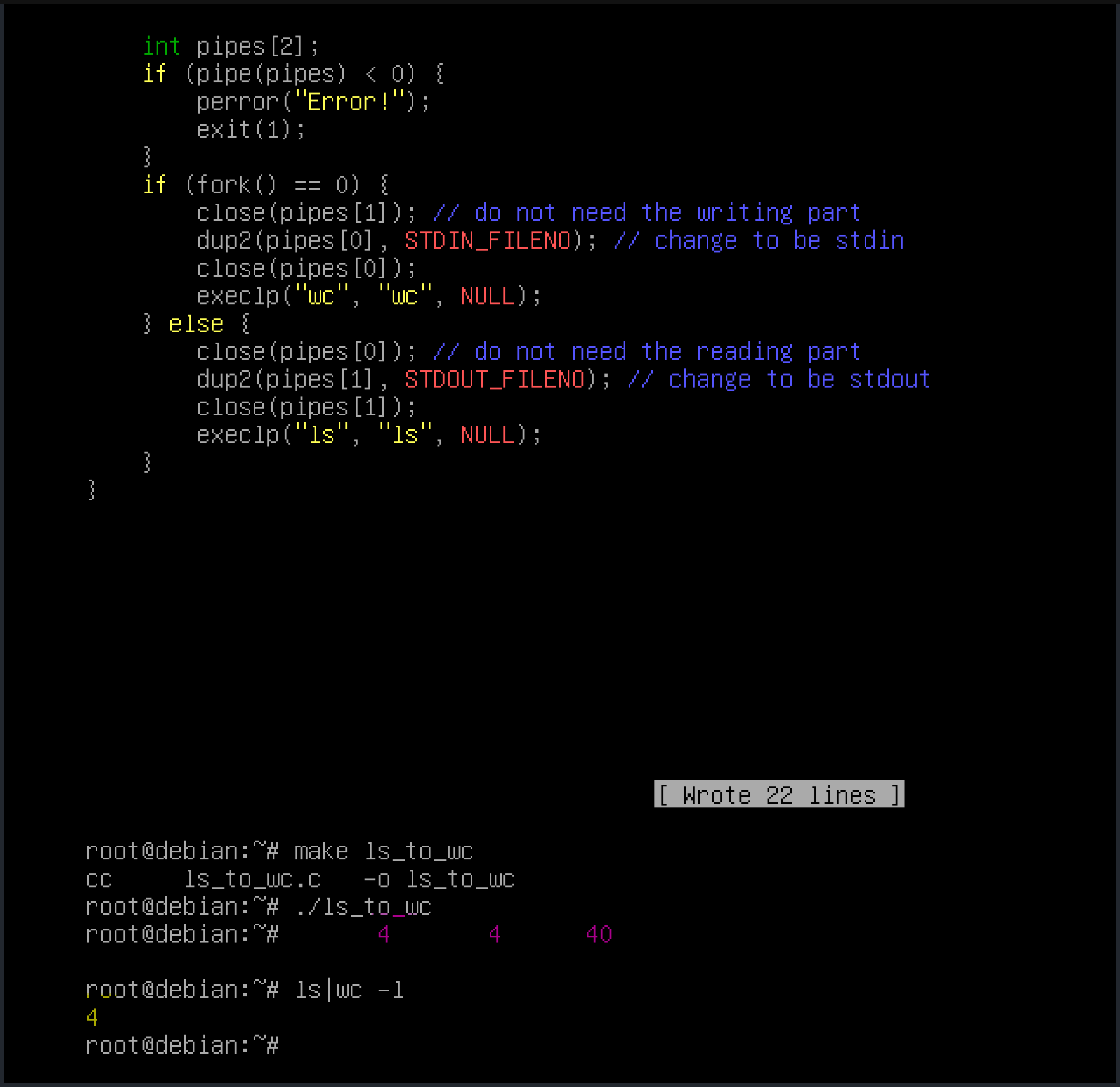
- ☑ `FILL HERE with your source code`

  The program is here in ls_to_wc.c:



```
  GNU nano 2.2.6                    File: ls_to_wc.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int pipes[2];
    if (pipe(pipes) < 0) {
        perror("Error!");
        exit(1);
    }
    if (fork() == 0) {
        close(pipes[1]); // do not need the writing part
        dup2(pipes[0], STDIN_FILENO); // change to be stdin
        close(pipes[0]);
        execlp("wc", "wc", NULL);
    } else {
        close(pipes[0]); // do not need the reading part
        dup2(pipes[1], STDOUT_FILENO); // change to be stdout
        close(pipes[1]);
        execlp("ls", "ls", NULL);
    }
}
```

Then the result is:



```
    int pipes[2];
    if (pipe(pipes) < 0) {
        perror("Error!");
        exit(1);
    }
    if (fork() == 0) {
        close(pipes[1]); // do not need the writing part
        dup2(pipes[0], STDIN_FILENO); // change to be stdin
        close(pipes[0]);
        execlp("wc", "wc", NULL);
    } else {
        close(pipes[0]); // do not need the reading part
        dup2(pipes[1], STDOUT_FILENO); // change to be stdout
        close(pipes[1]);
        execlp("ls", "ls", NULL);
    }
}

                                        [ Wrote 22 lines ]

root@debian:~# make ls_to_wc
cc     ls_to_wc.c    -o ls_to_wc
root@debian:~# ./ls_to_wc
root@debian:~#       4       4      40

root@debian:~# ls|wc -l
4
root@debian:~#
```

☑ Investigate how to have a bi-direction pipe

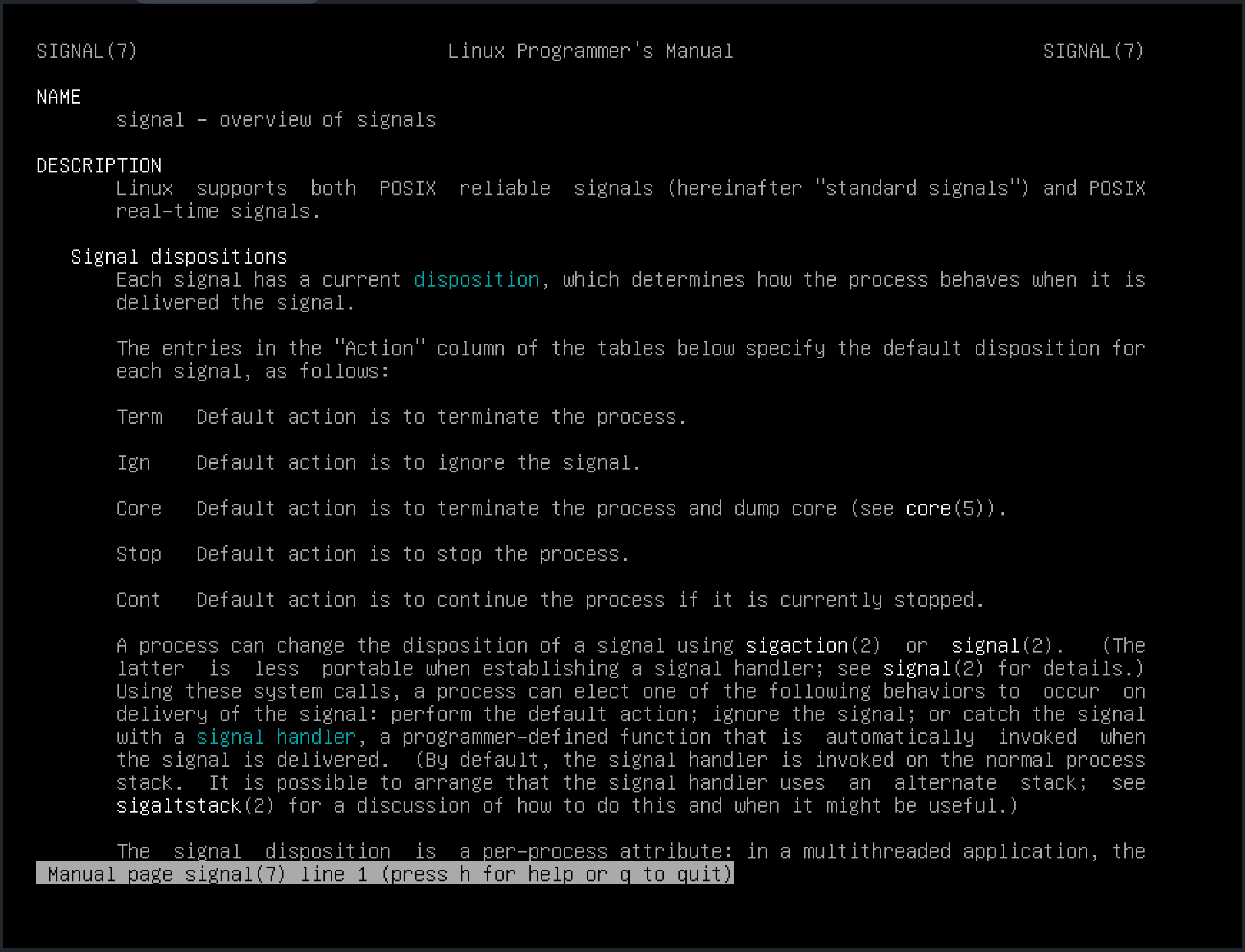
- ☑ `FILL HERE with your descriptions`
  We have two options.

  i. We can either just use two pipes (one from parent to child and one vice versa)

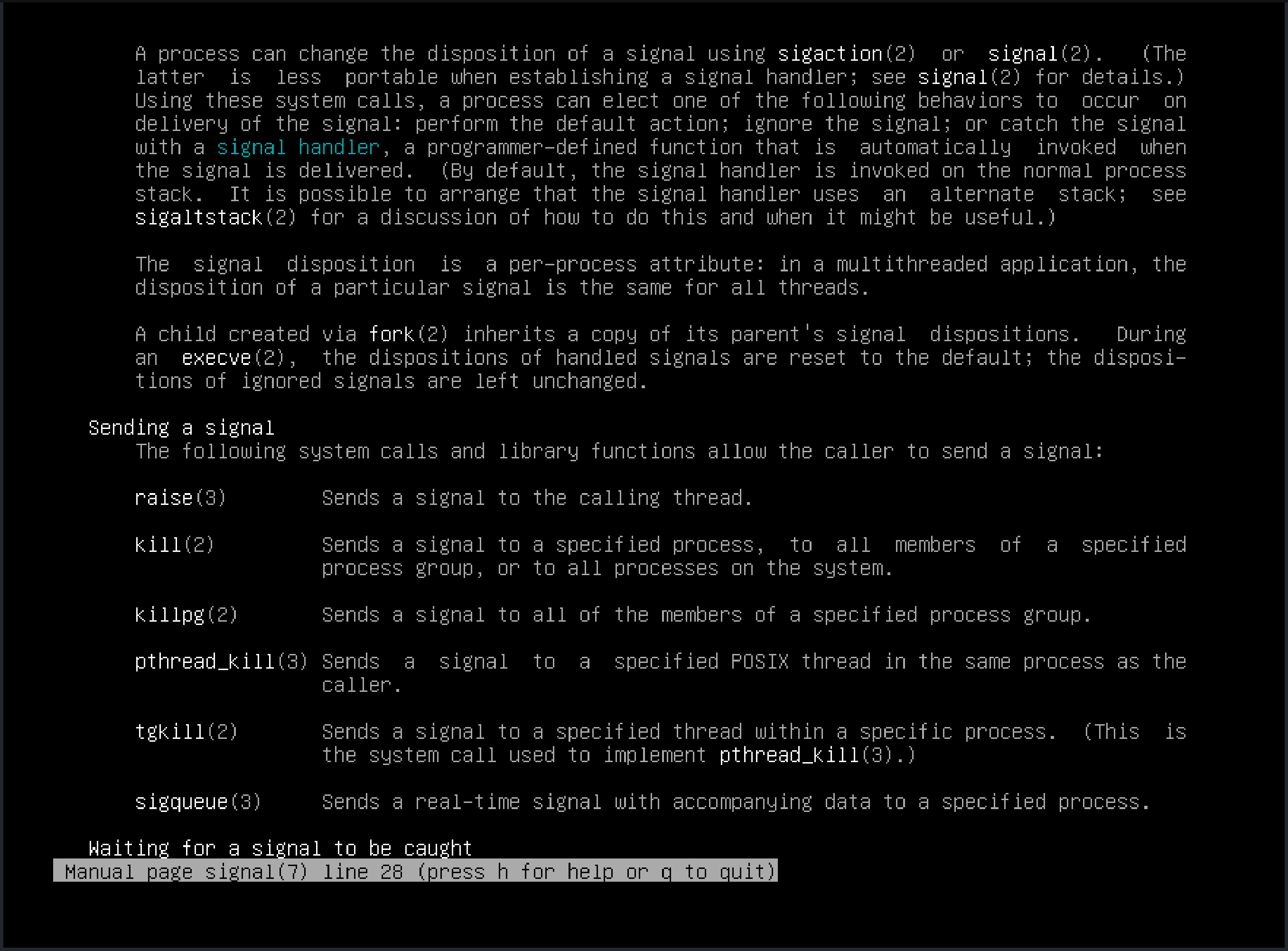
  ii. We can use unix sockets. Unix sockets are somehow a combination of TCP/UDP sockets and named pipes. They can operate in three different ways. SOCK_STREAM , SOCK_DGRAM and SOCK_SEQPACKET.
  For an experience like pipes, we can use SOCK_DGRAM which is mostly like UDP packets.
  Unix sockets can improve the speed of TCP and UDP connections in compared to sending packets to loopback. Thus it's recommended to use unix sockets for reverse proxying.

Section 5.3.2

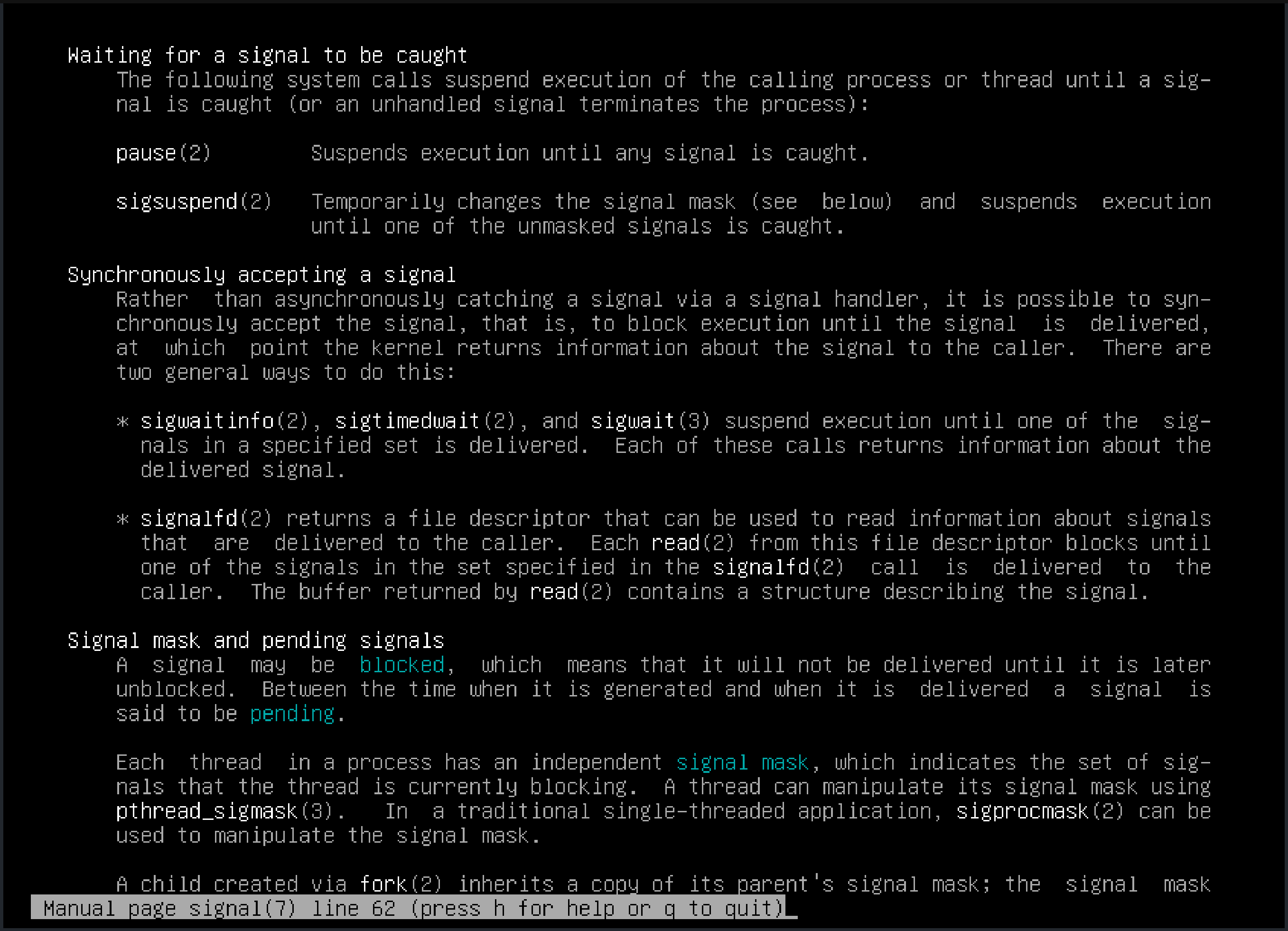
First we do `man 7 signal`. The result is:



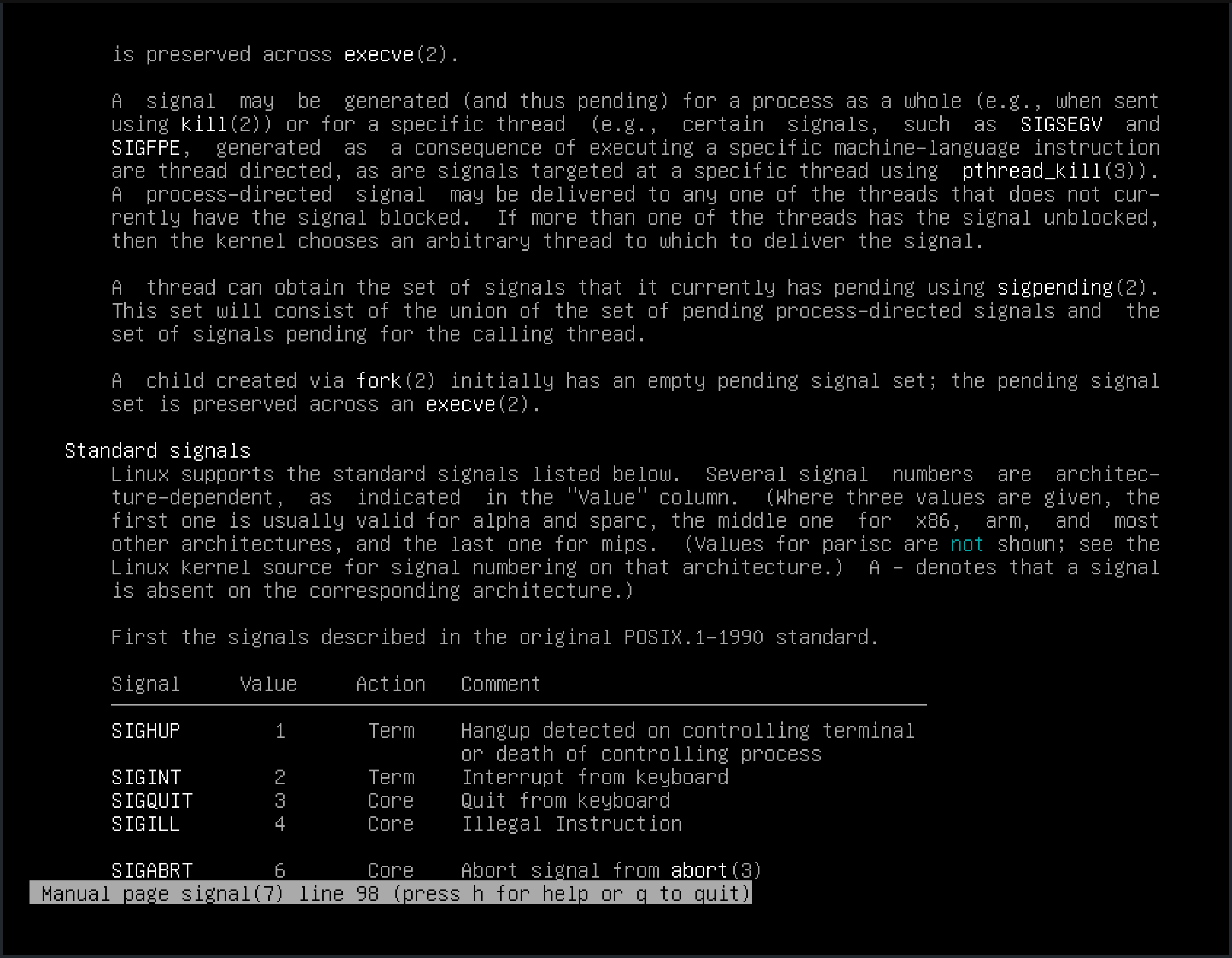
```
SIGNAL(7)                    Linux Programmer's Manual                    SIGNAL(7)

NAME
       signal - overview of signals

DESCRIPTION
       Linux  supports  both  POSIX  reliable  signals (hereinafter "standard signals") and POSIX
       real-time signals.

   Signal dispositions
       Each signal has a current disposition, which determines how the process behaves when it is
       delivered the signal.

       The entries in the "Action" column of the tables below specify the default disposition for
       each signal, as follows:

       Term   Default action is to terminate the process.

       Ign    Default action is to ignore the signal.

       Core   Default action is to terminate the process and dump core (see core(5)).

       Stop   Default action is to stop the process.

       Cont   Default action is to continue the process if it is currently stopped.

       A process can change the disposition of a signal using sigaction(2)  or  signal(2).   (The
       latter  is  less  portable when establishing a signal handler; see signal(2) for details.)
       Using these system calls, a process can elect one of the following behaviors to  occur  on
       delivery of the signal: perform the default action; ignore the signal; or catch the signal
       with a signal handler, a programmer-defined function that is  automatically  invoked  when
       the signal is delivered.  (By default, the signal handler is invoked on the normal process
       stack.  It is possible to arrange that the signal handler uses  an  alternate  stack;  see
       sigaltstack(2) for a discussion of how to do this and when it might be useful.)

       The  signal  disposition  is  a per-process attribute: in a multithreaded application, the
 Manual page signal(7) line 1 (press h for help or q to quit)
```



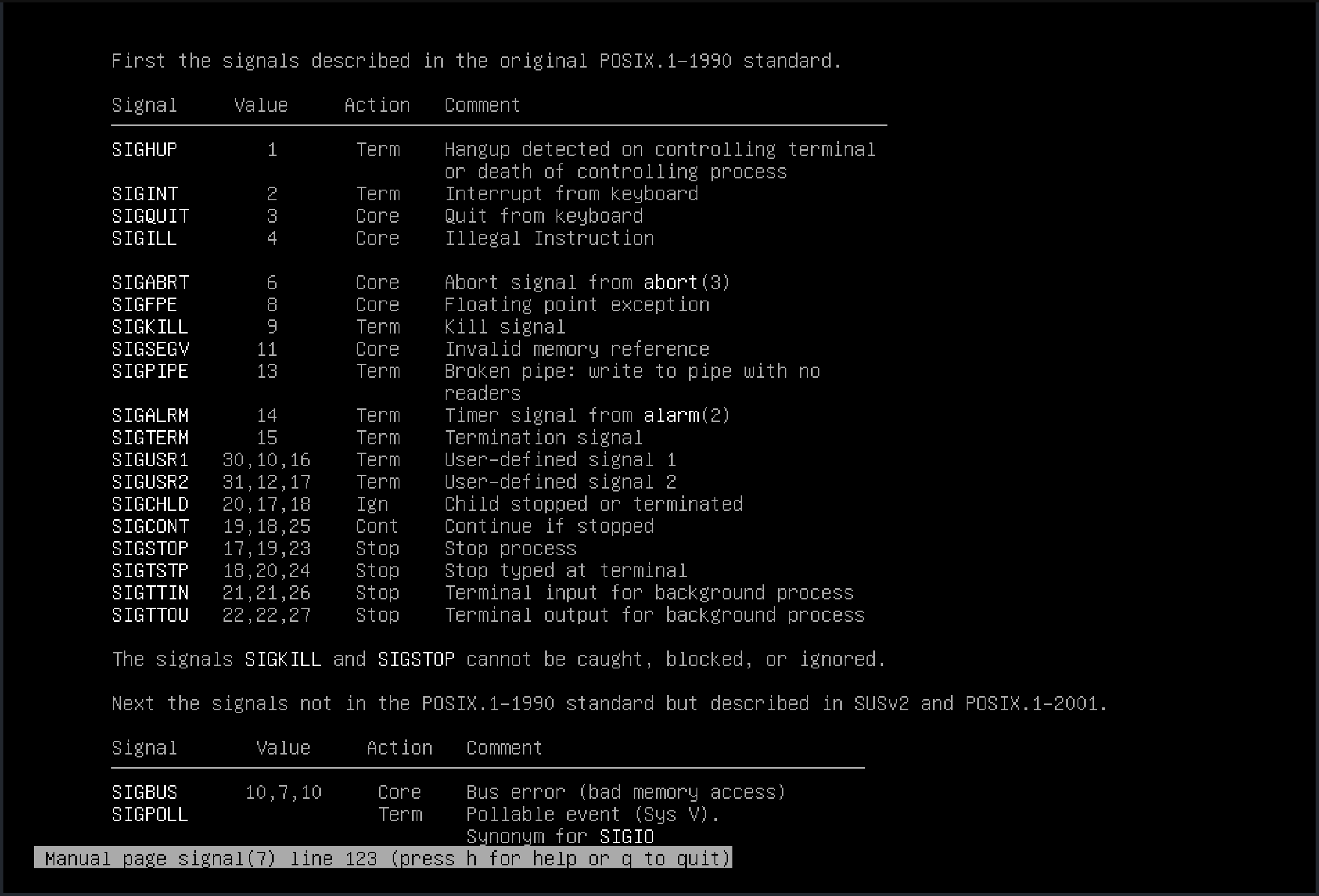
```
       A process can change the disposition of a signal using sigaction(2)  or  signal(2).   (The
       latter  is  less  portable when establishing a signal handler; see signal(2) for details.)
       Using these system calls, a process can elect one of the following behaviors to  occur  on
       delivery of the signal: perform the default action; ignore the signal; or catch the signal
       with a signal handler, a programmer-defined function that is  automatically  invoked  when
       the signal is delivered.  (By default, the signal handler is invoked on the normal process
       stack.  It is possible to arrange that the signal handler uses  an  alternate  stack;  see
       sigaltstack(2) for a discussion of how to do this and when it might be useful.)

       The  signal  disposition  is  a per-process attribute: in a multithreaded application, the
       disposition of a particular signal is the same for all threads.

       A child created via fork(2) inherits a copy of its parent's signal  dispositions.   During
       an  execve(2),  the dispositions of handled signals are reset to the default; the disposi‐
       tions of ignored signals are left unchanged.

   Sending a signal
       The following system calls and library functions allow the caller to send a signal:

       raise(3)        Sends a signal to the calling thread.

       kill(2)         Sends a signal to a specified process,  to  all  members  of  a  specified
                       process group, or to all processes on the system.

       killpg(2)       Sends a signal to all of the members of a specified process group.

       pthread_kill(3) Sends  a  signal  to  a  specified POSIX thread in the same process as the
                       caller.

       tgkill(2)       Sends a signal to a specified thread within a specific process.  (This  is
                       the system call used to implement pthread_kill(3).)

       sigqueue(3)     Sends a real-time signal with accompanying data to a specified process.

   Waiting for a signal to be caught
 Manual page signal(7) line 28 (press h for help or q to quit)
```



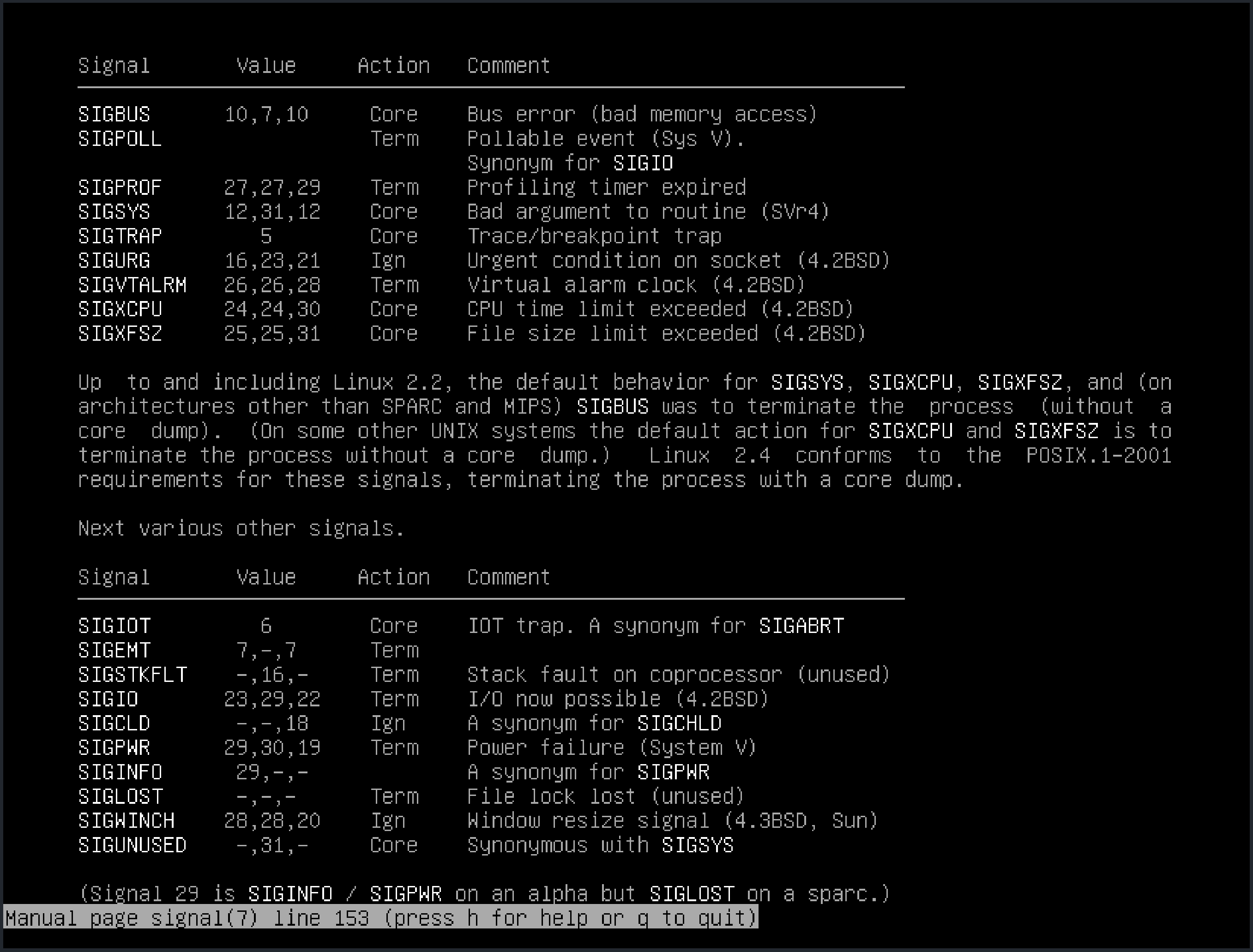
```
   Waiting for a signal to be caught
       The following system calls suspend execution of the calling process or thread until a sig‐
       nal is caught (or an unhandled signal terminates the process):

       pause(2)        Suspends execution until any signal is caught.

       sigsuspend(2)   Temporarily changes the signal mask (see  below)  and  suspends  execution
                       until one of the unmasked signals is caught.

   Synchronously accepting a signal
       Rather  than asynchronously catching a signal via a signal handler, it is possible to syn‐
       chronously accept the signal, that is, to block execution until the signal  is  delivered,
       at  which  point the kernel returns information about the signal to the caller.  There are
       two general ways to do this:

       * sigwaitinfo(2), sigtimedwait(2), and sigwait(3) suspend execution until one of the  sig‐
         nals in a specified set is delivered.  Each of these calls returns information about the
         delivered signal.

       * signalfd(2) returns a file descriptor that can be used to read information about signals
         that  are  delivered to the caller.  Each read(2) from this file descriptor blocks until
         one of the signals in the set specified in the signalfd(2)  call  is  delivered  to  the
         caller.  The buffer returned by read(2) contains a structure describing the signal.

   Signal mask and pending signals
       A  signal  may  be  blocked,  which  means that it will not be delivered until it is later
       unblocked.  Between the time when it is generated and when it is  delivered  a  signal  is
       said to be pending.

       Each  thread  in a process has an independent signal mask, which indicates the set of sig‐
       nals that the thread is currently blocking.  A thread can manipulate its signal mask using
       pthread_sigmask(3).   In  a traditional single-threaded application, sigprocmask(2) can be
       used to manipulate the signal mask.

       A child created via fork(2) inherits a copy of its parent's signal mask; the  signal  mask
 Manual page signal(7) line 62 (press h for help or q to quit)
```



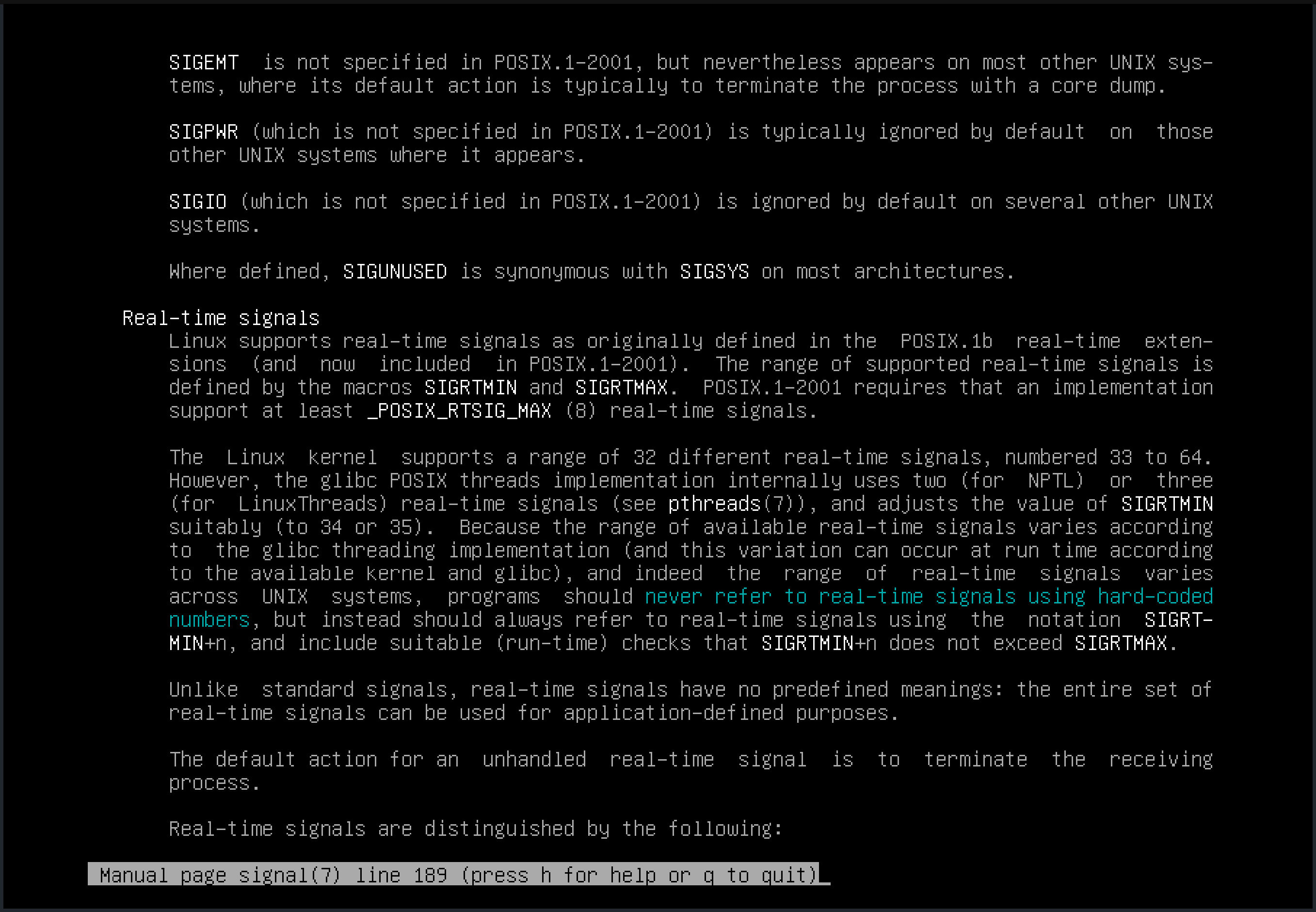
```
       is preserved across execve(2).

       A  signal  may  be  generated (and thus pending) for a process as a whole (e.g., when sent
       using kill(2)) or for a specific thread  (e.g.,  certain  signals,  such  as  SIGSEGV  and
       SIGFPE,  generated  as  a consequence of executing a specific machine-language instruction
       are thread directed, as are signals targeted at a specific thread using  pthread_kill(3)).
       A  process-directed  signal  may be delivered to any one of the threads that does not cur‐
       rently have the signal blocked.  If more than one of the threads has the signal unblocked,
       then the kernel chooses an arbitrary thread to which to deliver the signal.

       A  thread can obtain the set of signals that it currently has pending using sigpending(2).
       This set will consist of the union of the set of pending process-directed signals and  the
       set of signals pending for the calling thread.

       A  child created via fork(2) initially has an empty pending signal set; the pending signal
       set is preserved across an execve(2).

   Standard signals
       Linux supports the standard signals listed below.  Several signal  numbers  are  architec‐
       ture-dependent,  as  indicated  in the "Value" column.  (Where three values are given, the
       first one is usually valid for alpha and sparc, the middle one  for  x86,  arm,  and  most
       other architectures, and the last one for mips.  (Values for parisc are not shown; see the
       Linux kernel source for signal numbering on that architecture.)  A - denotes that a signal
       is absent on the corresponding architecture.)

       First the signals described in the original POSIX.1-1990 standard.

       Signal     Value     Action   Comment
       ──────────────────────────────────────────────────────────────────────
       SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2       Term    Interrupt from keyboard
       SIGQUIT       3       Core    Quit from keyboard
       SIGILL        4       Core    Illegal Instruction

       SIGABRT       6       Core    Abort signal from abort(3)
 Manual page signal(7) line 98 (press h for help or q to quit)
```



```
       First the signals described in the original POSIX.1-1990 standard.

       Signal     Value     Action   Comment
       ──────────────────────────────────────────────────────────────────────
       SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2       Term    Interrupt from keyboard
       SIGQUIT       3       Core    Quit from keyboard
       SIGILL        4       Core    Illegal Instruction

       SIGABRT       6       Core    Abort signal from abort(3)
       SIGFPE        8       Core    Floating point exception
       SIGKILL       9       Term    Kill signal
       SIGSEGV      11       Core    Invalid memory reference
       SIGPIPE      13       Term    Broken pipe: write to pipe with no
                                     readers
       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
       SIGTSTP   18,20,24    Stop    Stop typed at terminal
       SIGTTIN   21,21,26    Stop    Terminal input for background process
       SIGTTOU   22,22,27    Stop    Terminal output for background process

       The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

       Next the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.

       Signal       Value     Action   Comment
       ────────────────────────────────────────────────────────────────────
       SIGBUS      10,7,10     Core    Bus error (bad memory access)
       SIGPOLL                 Term    Pollable event (Sys V).
                                       Synonym for SIGIO
 Manual page signal(7) line 123 (press h for help or q to quit)
```



```
       Signal       Value     Action   Comment
       ────────────────────────────────────────────────────────────────────
       SIGBUS      10,7,10     Core    Bus error (bad memory access)
       SIGPOLL                 Term    Pollable event (Sys V).
                                       Synonym for SIGIO
       SIGPROF     27,27,29    Term    Profiling timer expired
       SIGSYS      12,31,12    Core    Bad argument to routine (SVr4)
       SIGTRAP        5        Core    Trace/breakpoint trap
       SIGURG      16,23,21    Ign     Urgent condition on socket (4.2BSD)
       SIGVTALRM   26,26,28    Term    Virtual alarm clock (4.2BSD)
       SIGXCPU     24,24,30    Core    CPU time limit exceeded (4.2BSD)
       SIGXFSZ     25,25,31    Core    File size limit exceeded (4.2BSD)

       Up  to and including Linux 2.2, the default behavior for SIGSYS, SIGXCPU, SIGXFSZ, and (on
       architectures other than SPARC and MIPS) SIGBUS was to terminate the  process  (without  a
       core  dump).  (On some other UNIX systems the default action for SIGXCPU and SIGXFSZ is to
       terminate the process without a core  dump.)   Linux  2.4  conforms  to  the  POSIX.1-2001
       requirements for these signals, terminating the process with a core dump.

       Next various other signals.

       Signal       Value     Action   Comment
       ────────────────────────────────────────────────────────────────────
       SIGIOT         6        Core    IOT trap. A synonym for SIGABRT
       SIGEMT       7,-,7      Term
       SIGSTKFLT    -,16,-     Term    Stack fault on coprocessor (unused)
       SIGIO       23,29,22    Term    I/O now possible (4.2BSD)
       SIGCLD       -,-,18     Ign     A synonym for SIGCHLD
       SIGPWR      29,30,19    Term    Power failure (System V)
       SIGINFO      29,-,-             A synonym for SIGPWR
       SIGLOST      -,-,-      Term    File lock lost (unused)
       SIGWINCH    28,28,20    Ign     Window resize signal (4.3BSD, Sun)
       SIGUNUSED    -,31,-     Core    Synonymous with SIGSYS

       (Signal 29 is SIGINFO / SIGPWR on an alpha but SIGLOST on a sparc.)
Manual page signal(7) line 153 (press h for help or q to quit)
```

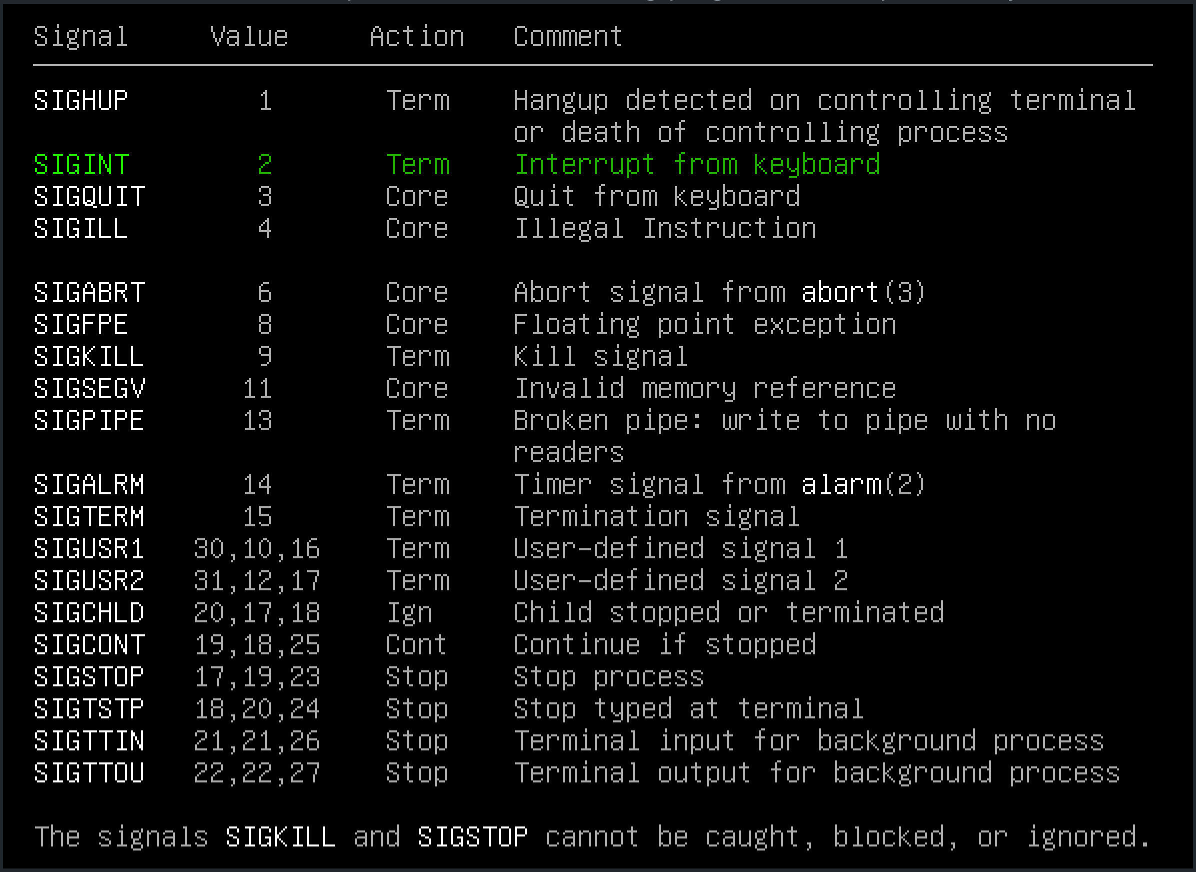


```
       SIGEMT  is not specified in POSIX.1-2001, but nevertheless appears on most other UNIX sys-
       tems, where its default action is typically to terminate the process with a core dump.

       SIGPWR (which is not specified in POSIX.1-2001) is typically ignored by default  on  those
       other UNIX systems where it appears.

       SIGIO (which is not specified in POSIX.1-2001) is ignored by default on several other UNIX
       systems.

       Where defined, SIGUNUSED is synonymous with SIGSYS on most architectures.

   Real-time signals
       Linux supports real-time signals as originally defined in the  POSIX.1b  real-time  exten-
       sions  (and  now  included  in POSIX.1-2001).  The range of supported real-time signals is
       defined by the macros SIGRTMIN and SIGRTMAX.  POSIX.1-2001 requires that an implementation
       support at least _POSIX_RTSIG_MAX (8) real-time signals.

       The  Linux  kernel  supports a range of 32 different real-time signals, numbered 33 to 64.
       However, the glibc POSIX threads implementation internally uses two (for  NPTL)  or  three
       (for  LinuxThreads) real-time signals (see pthreads(7)), and adjusts the value of SIGRTMIN
       suitably (to 34 or 35).  Because the range of available real-time signals varies according
       to  the glibc threading implementation (and this variation can occur at run time according
       to the available kernel and glibc), and indeed  the  range  of  real-time  signals  varies
       across  UNIX  systems,  programs  should never refer to real-time signals using hard-coded
       numbers, but instead should always refer to real-time signals using  the  notation  SIGRT-
       MIN+n, and include suitable (run-time) checks that SIGRTMIN+n does not exceed SIGRTMAX.

       Unlike  standard signals, real-time signals have no predefined meanings: the entire set of
       real-time signals can be used for application-defined purposes.

       The default action for an  unhandled  real-time  signal  is  to  terminate  the  receiving
       process.

       Real-time signals are distinguished by the following:

 Manual page signal(7) line 189 (press h for help or q to quit)
```

☑ Describe the usecase of different signals:

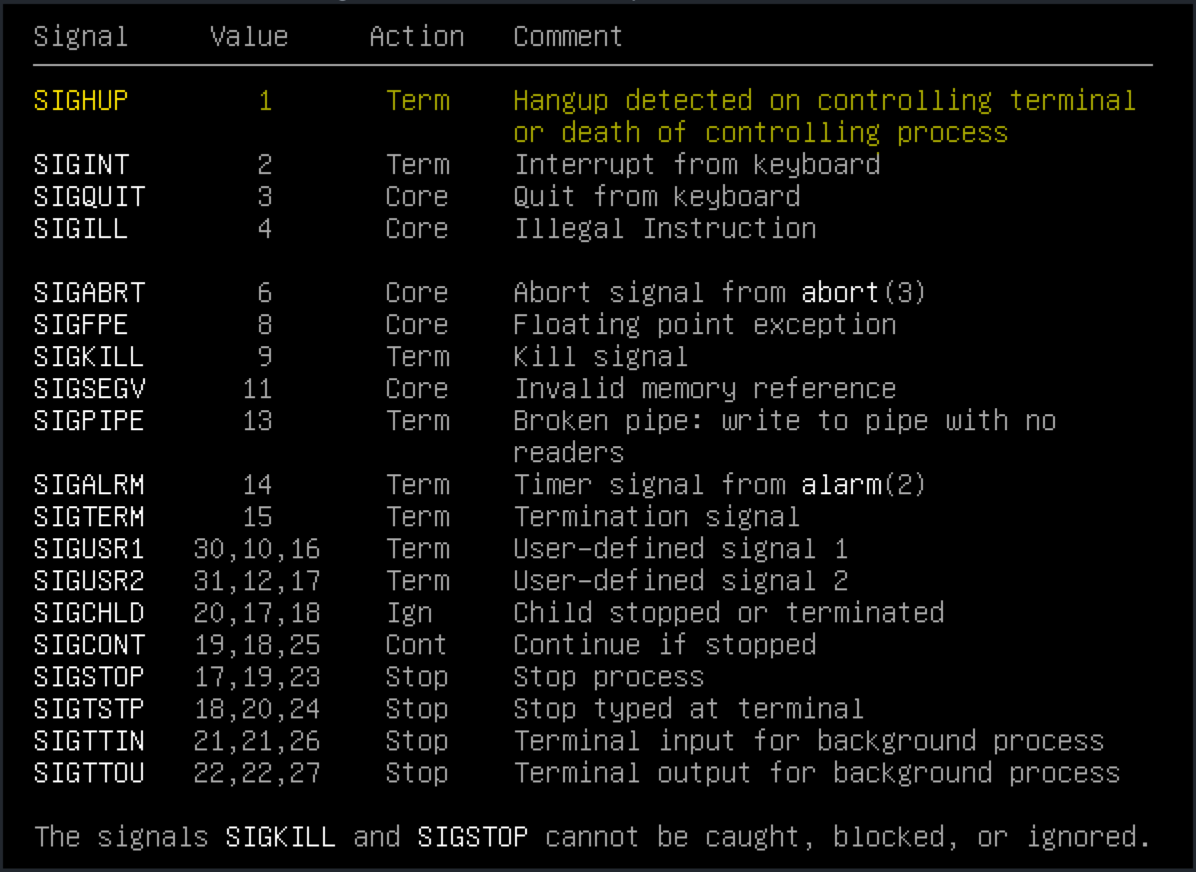
- ☑ `[FILL HERE with the description for SIGINT.]`

  `SIGINT`: SIGINT is the signal sent when we press Ctrl+C. The default action is to terminate the process. However, some programs override this action and handle it differently. So, it is a signal number 2. Is send when we press ctrl + c in a running program. (Interrupt from keyboard).



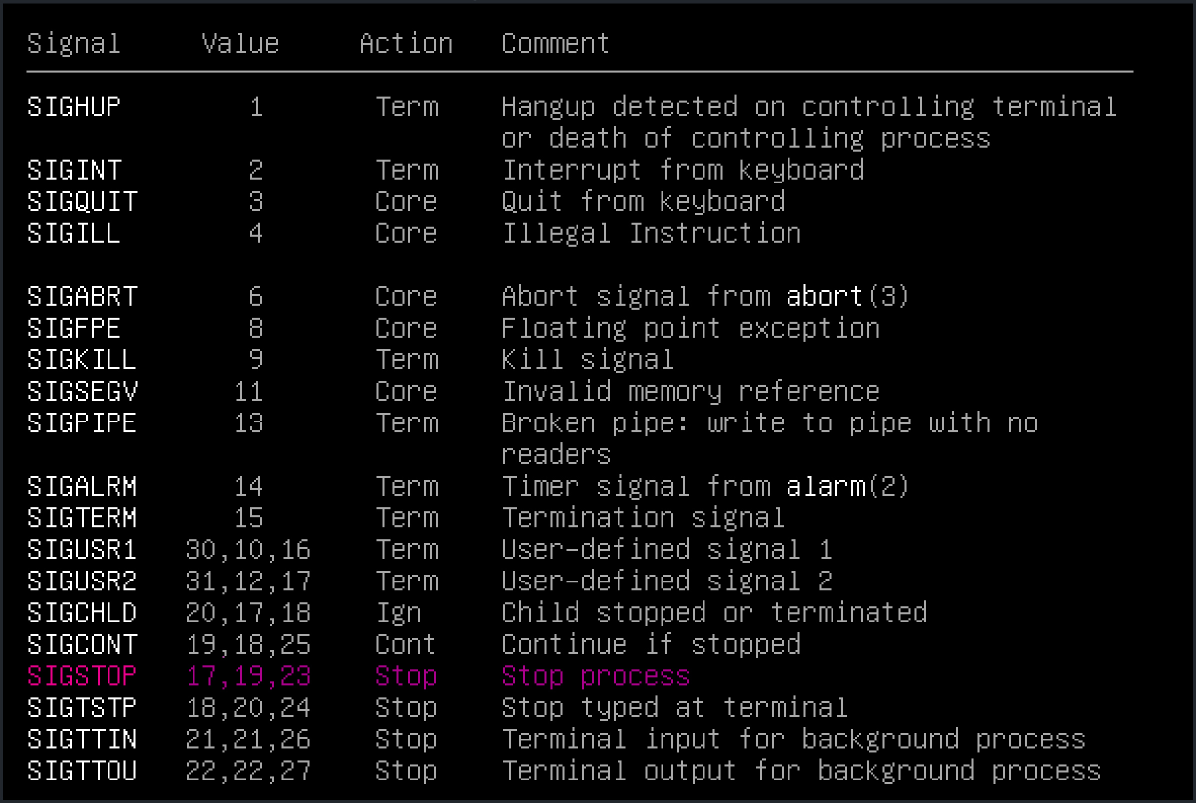
```
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction

SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
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
SIGTSTP   18,20,24    Stop    Stop typed at terminal
SIGTTIN   21,21,26    Stop    Terminal input for background process
SIGTTOU   22,22,27    Stop    Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

☑ `[FILL HERE with the description for SIGHUP.]`
`SIGHUP`: ("signal hang up") is a signal sent to a process when its controlling terminal is closed. It was originally designed to notify the process of a serial line drop. SIGHUP is a symbolic constant defined in the header file signal.h.
In other words, the SIGHUP signal was sent to a process when its controlling terminal is closed. Nowadays, it's often used to indicate that a process should reload its configuration or restart. The default action of this signal is to terminate the process.

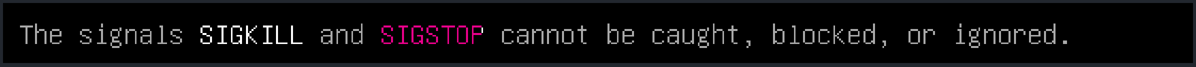


```
Signal      Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction

SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
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
SIGTSTP   18,20,24    Stop    Stop typed at terminal
SIGTTIN   21,21,26    Stop    Terminal input for background process
SIGTTOU   22,22,27    Stop    Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

- ☑ `[FILL HERE with the description for SIGSTOP.]`
  `SIGSTOP`: The SIGSTOP signal instructs a process to halt its execution temporarily. Unlike SIGINT, SIGSTOP cannot be caught, blocked, or ignored by the process. The process will remain stopped until it receives a SIGCONT signal.

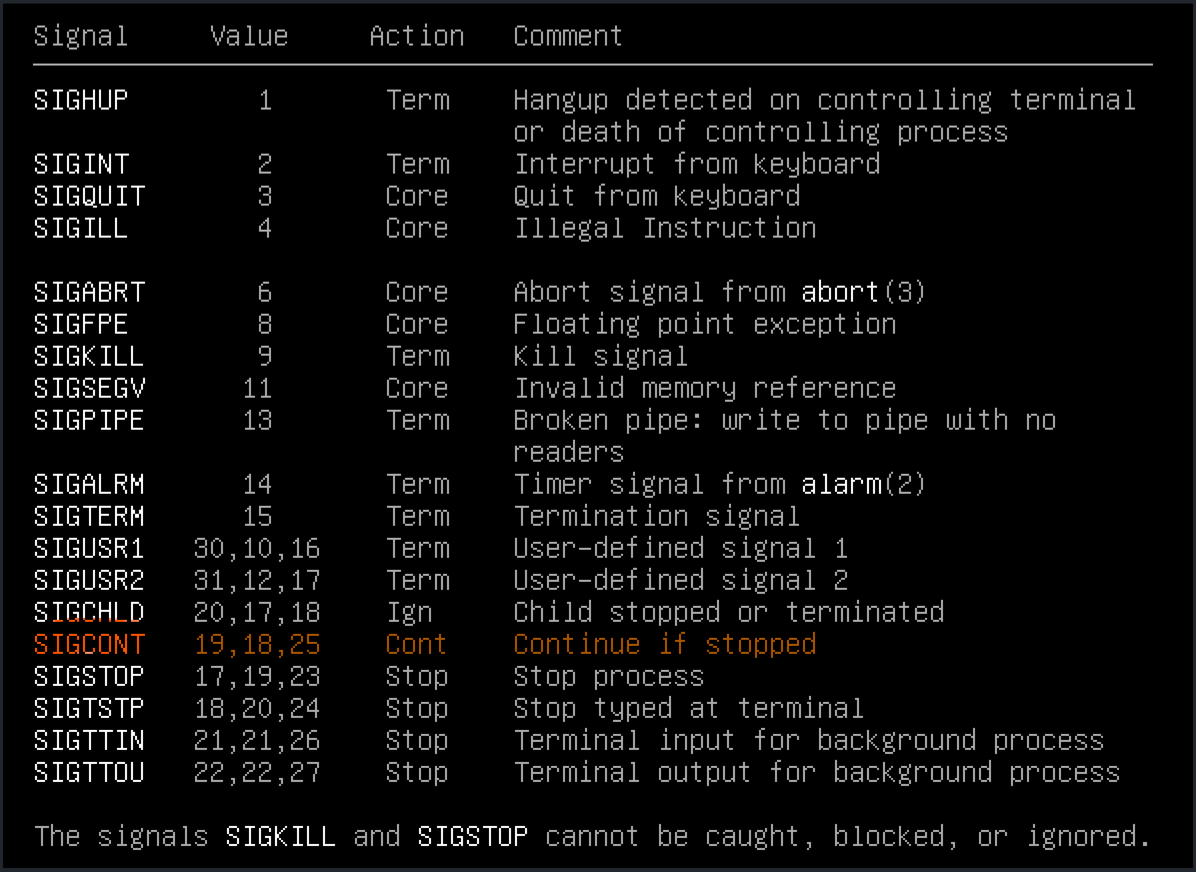


```
Signal      Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction

SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
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
SIGTSTP   18,20,24    Stop    Stop typed at terminal
SIGTTIN   21,21,26    Stop    Terminal input for background process
SIGTTOU   22,22,27    Stop    Terminal output for background process
```



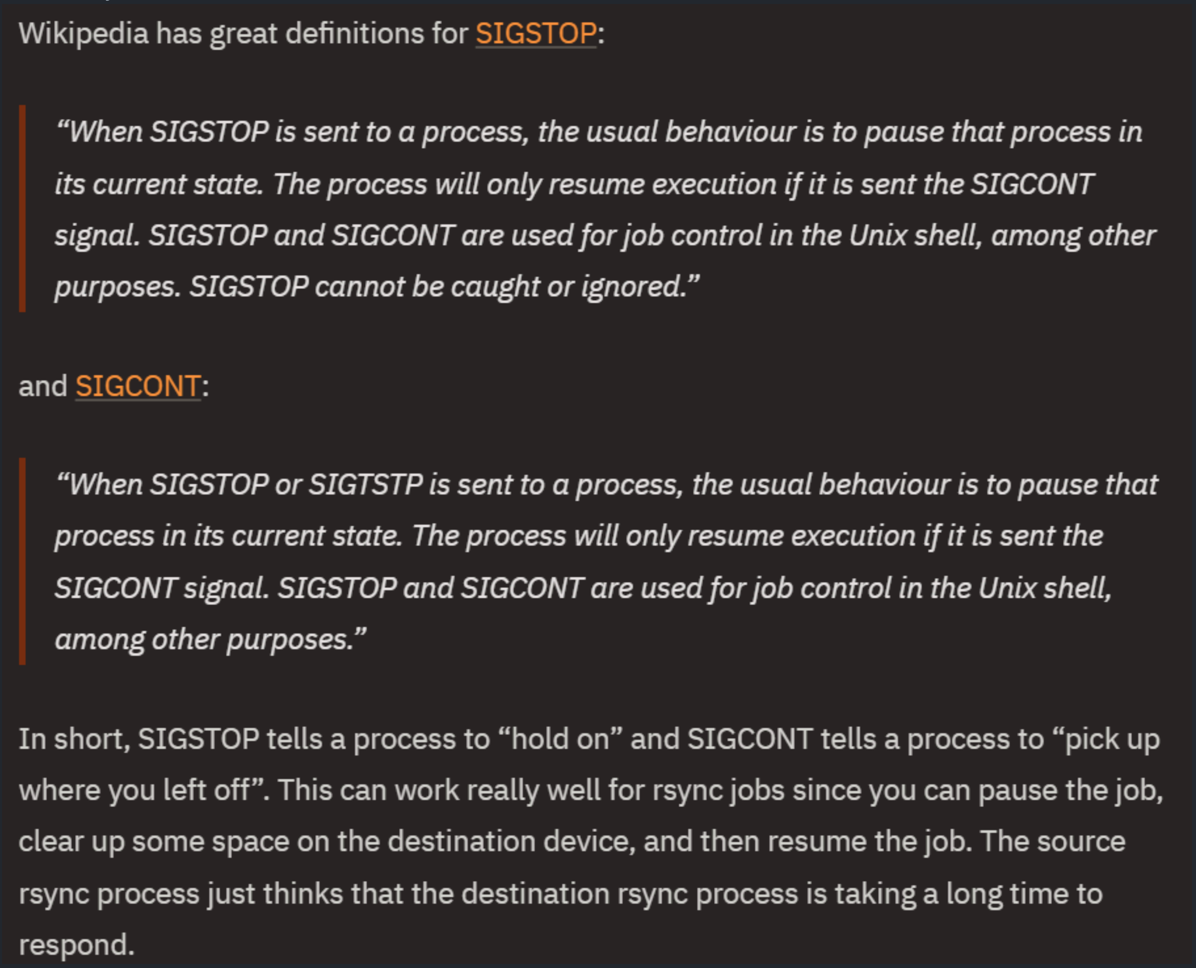
```
The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

☑ `[FILL HERE with the description for SIGCONT.]`
`SIGCONT`: The SIGCONT signal is sent to a process to instruct it to continue execution if it was previously stopped by SIGSTOP or SIGTSTP. It's commonly used to resume processes that were paused or suspended.



```
Signal      Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP         1       Term    Hangup detected on controlling terminal
                               or death of controlling process
SIGINT         2       Term    Interrupt from keyboard
SIGQUIT        3       Core    Quit from keyboard
SIGILL         4       Core    Illegal Instruction

SIGABRT        6       Core    Abort signal from abort(3)
SIGFPE         8       Core    Floating point exception
SIGKILL        9       Term    Kill signal
SIGSEGV       11       Core    Invalid memory reference
SIGPIPE       13       Term    Broken pipe: write to pipe with no
                               readers
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
SIGTSTP   18,20,24     Stop    Stop typed at terminal
SIGTTIN   21,21,26     Stop    Terminal input for background process
SIGTTOU   22,22,27     Stop    Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

Summery:



Wikipedia has great definitions for SIGSTOP:

> *“When SIGSTOP is sent to a process, the usual behaviour is to pause that process in its current state. The process will only resume execution if it is sent the SIGCONT signal. SIGSTOP and SIGCONT are used for job control in the Unix shell, among other purposes. SIGSTOP cannot be caught or ignored.”*

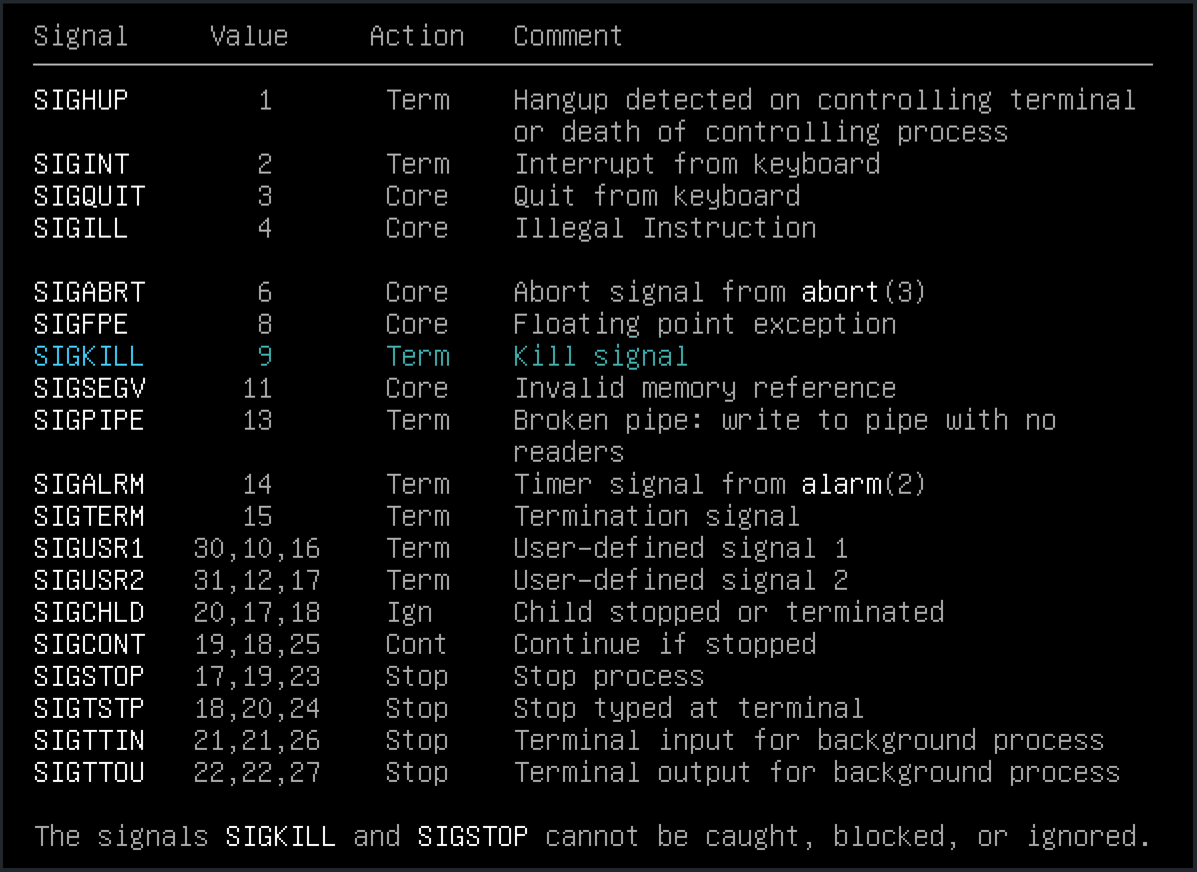
and SIGCONT:

> *“When SIGSTOP or SIGTSTP is sent to a process, the usual behaviour is to pause that process in its current state. The process will only resume execution if it is sent the SIGCONT signal. SIGSTOP and SIGCONT are used for job control in the Unix shell, among other purposes.”*

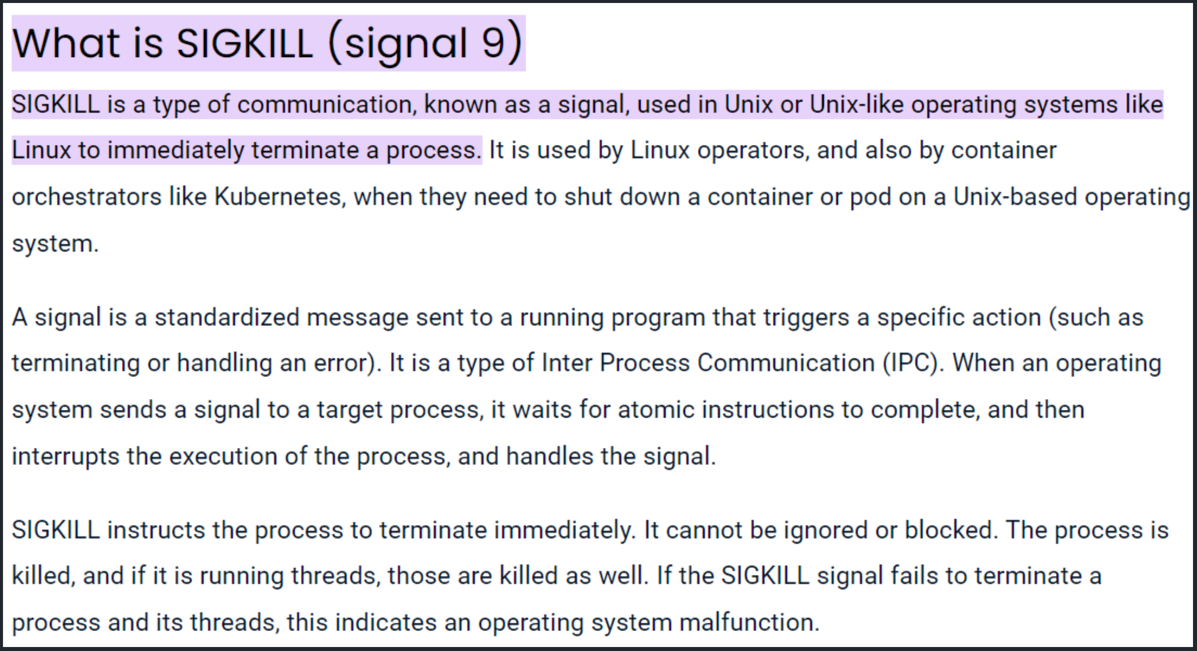
In short, SIGSTOP tells a process to “hold on” and SIGCONT tells a process to “pick up where you left off”. This can work really well for rsync jobs since you can pause the job, clear up some space on the destination device, and then resume the job. The source rsync process just thinks that the destination rsync process is taking a long time to respond.

- ☑ `[FILL HERE with the description for SIGKILL.]`
  `SIGKILL`: Kill signal. This signal cannot be caught, blocked, or ignored. It's often used as a last resort to terminate
  unresponsive or misbehaving processes that cannot be terminated through other means.

```
The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```



```
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction

SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
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
SIGTSTP   18,20,24    Stop    Stop typed at terminal
SIGTTIN   21,21,26    Stop    Terminal input for background process
SIGTTOU   22,22,27    Stop    Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```



What is SIGKILL (signal 9)

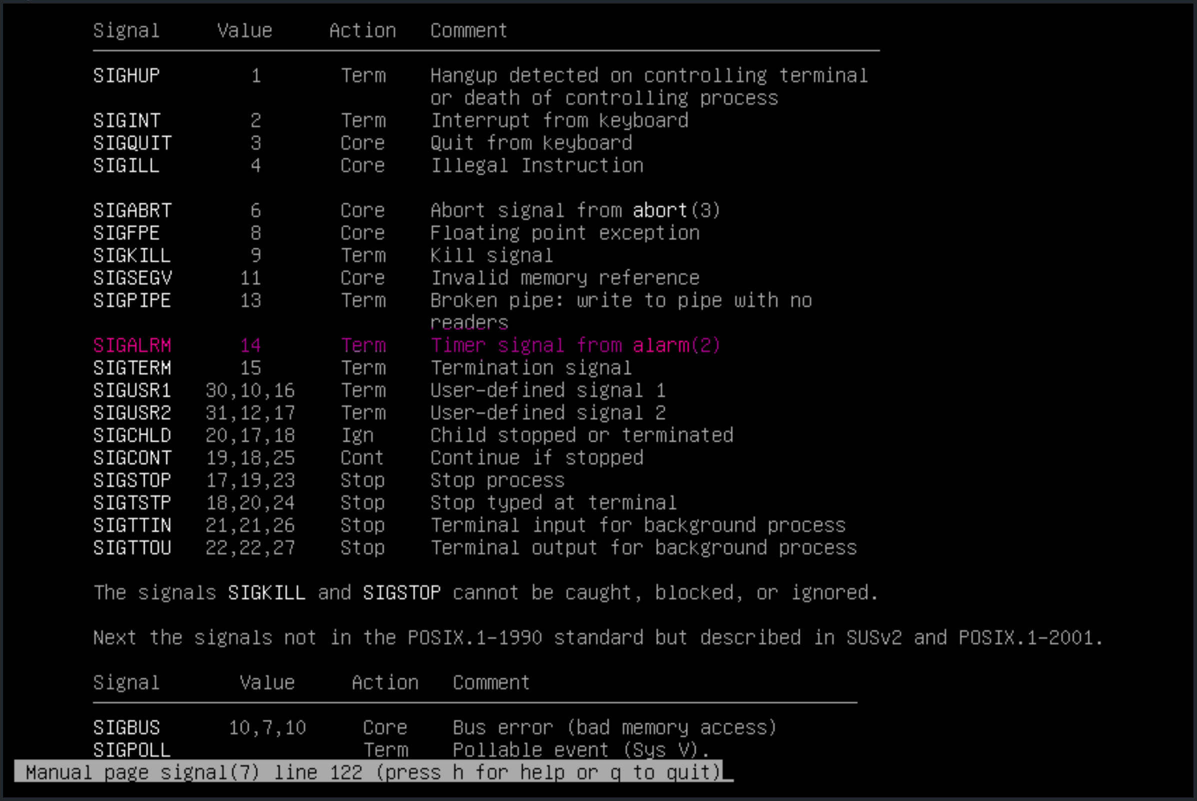
SIGKILL is a type of communication, known as a signal, used in Unix or Unix-like operating systems like Linux to immediately terminate a process. It is used by Linux operators, and also by container orchestrators like Kubernetes, when they need to shut down a container or pod on a Unix-based operating system.

A signal is a standardized message sent to a running program that triggers a specific action (such as terminating or handling an error). It is a type of Inter Process Communication (IPC). When an operating system sends a signal to a target process, it waits for atomic instructions to complete, and then interrupts the execution of the process, and handles the signal.

SIGKILL instructs the process to terminate immediately. It cannot be ignored or blocked. The process is killed, and if it is running threads, those are killed as well. If the SIGKILL signal fails to terminate a process and its threads, this indicates an operating system malfunction.

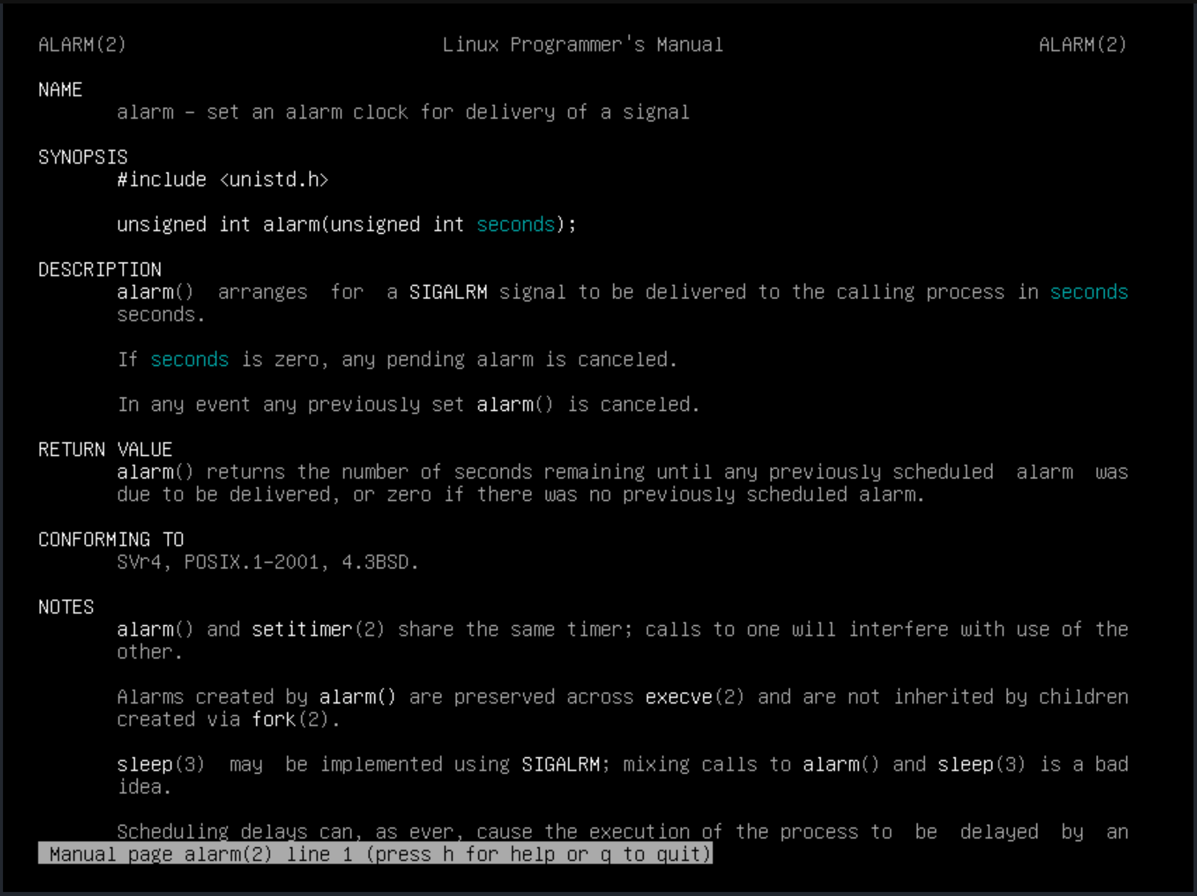
- ☑ Describe SIGALRM

- ☑ `[FILL HERE with your description.]`
  `SIGALRM`: SIGALRM is an asynchronous signal. The SIGALRM signal is raised when a time interval specified in a call to the alarm or alarmd function expires. Because SIGALRM is an asynchronous signal, the SAS/C library discovers the signal only when you call a function, when a function returns, or when you issue a call to sigchk. In other words, SIGALRM is a signal sent to a process when the real-time clock timer set by alarm() or setitimer() expires.It's often used in programming to
  handle timeouts or to perform periodic tasks.When a process receives SIGALRM, it typically interrupts its current execution and invokes a signal handler, if one has been registered for this signal.

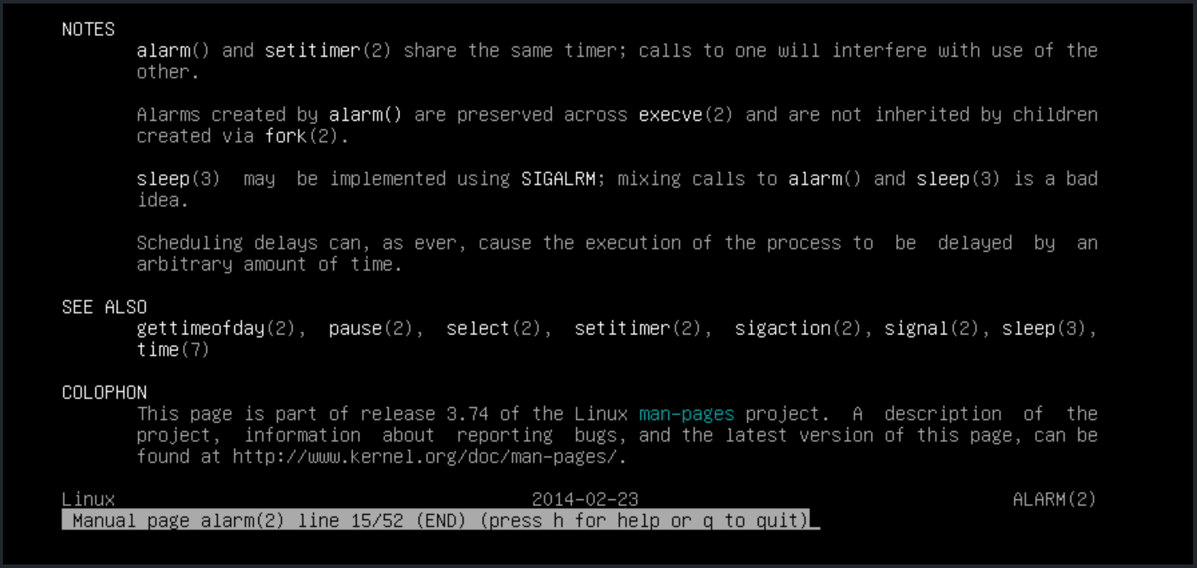


```
       Signal     Value     Action   Comment
       ──────────────────────────────────────────────────────────────────────
       SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2       Term    Interrupt from keyboard
       SIGQUIT       3       Core    Quit from keyboard
       SIGILL        4       Core    Illegal Instruction

       SIGABRT       6       Core    Abort signal from abort(3)
       SIGFPE        8       Core    Floating point exception
       SIGKILL       9       Term    Kill signal
       SIGSEGV      11       Core    Invalid memory reference
       SIGPIPE      13       Term    Broken pipe: write to pipe with no
                                     readers
       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
       SIGTSTP   18,20,24    Stop    Stop typed at terminal
       SIGTTIN   21,21,26    Stop    Terminal input for background process
       SIGTTOU   22,22,27    Stop    Terminal output for background process

       The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

       Next the signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.

       Signal       Value     Action   Comment
       ────────────────────────────────────────────────────────────────────
       SIGBUS      10,7,10     Core    Bus error (bad memory access)
       SIGPOLL                 Term    Pollable event (Sys V).
 Manual page signal(7) line 122 (press h for help or q to quit)
```

  With `man alarm`:



```
ALARM(2)                      Linux Programmer's Manual                     ALARM(2)

NAME
       alarm - set an alarm clock for delivery of a signal

SYNOPSIS
       #include <unistd.h>

       unsigned int alarm(unsigned int seconds);

DESCRIPTION
       alarm()  arranges  for  a SIGALRM signal to be delivered to the calling process in seconds
       seconds.

       If seconds is zero, any pending alarm is canceled.

       In any event any previously set alarm() is canceled.

RETURN VALUE
       alarm() returns the number of seconds remaining until any previously scheduled  alarm  was
       due to be delivered, or zero if there was no previously scheduled alarm.

CONFORMING TO
       SVr4, POSIX.1-2001, 4.3BSD.

NOTES
       alarm() and setitimer(2) share the same timer; calls to one will interfere with use of the
       other.

       Alarms created by alarm() are preserved across execve(2) and are not inherited by children
       created via fork(2).

       sleep(3)  may  be implemented using SIGALRM; mixing calls to alarm() and sleep(3) is a bad
       idea.

       Scheduling delays can, as ever, cause the execution of the process to  be  delayed  by  an
 Manual page alarm(2) line 1 (press h for help or q to quit)
```

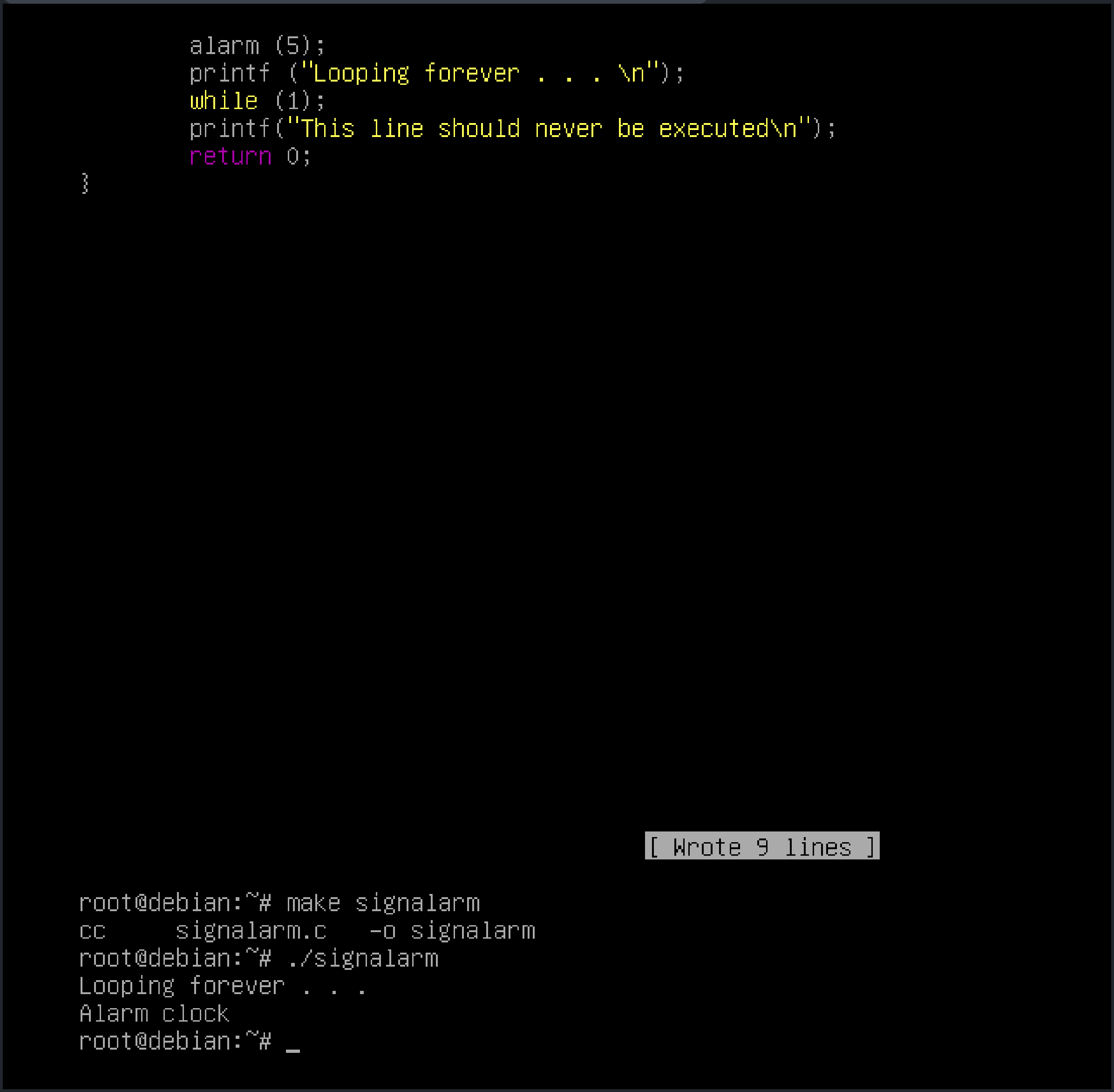


```
NOTES
       alarm() and setitimer(2) share the same timer; calls to one will interfere with use of the
       other.

       Alarms created by alarm() are preserved across execve(2) and are not inherited by children
       created via fork(2).

       sleep(3)  may  be implemented using SIGALRM; mixing calls to alarm() and sleep(3) is a bad
       idea.

       Scheduling delays can, as ever, cause the execution of the process to  be  delayed  by  an
       arbitrary amount of time.

SEE ALSO
       gettimeofday(2),  pause(2),  select(2),  setitimer(2),  sigaction(2), signal(2), sleep(3),
       time(7)

COLOPHON
       This page is part of release 3.74 of the Linux man-pages project.  A  description  of  the
       project,  information  about  reporting  bugs, and the latest version of this page, can be
       found at http://www.kernel.org/doc/man-pages/.

Linux                             2014-02-23                                    ALARM(2)
 Manual page alarm(2) line 15/52 (END) (press h for help or q to quit)
```

- ☑ Investigate the given code

  - ☑ `[FILL HERE with your description.]`
    This program will terminate itself in 5 seconds. At first, it will setup an alarm to be fired up in 5 seconds. Then it will print Looping forever . . . and then it will enter an infinite loop. While the loop is executing, the SIGALRM will be delivered to program. This causes the application to be terminated because we don't have any signal handler for it.

☑ `[FILL HERE with screenshot of program execution]`

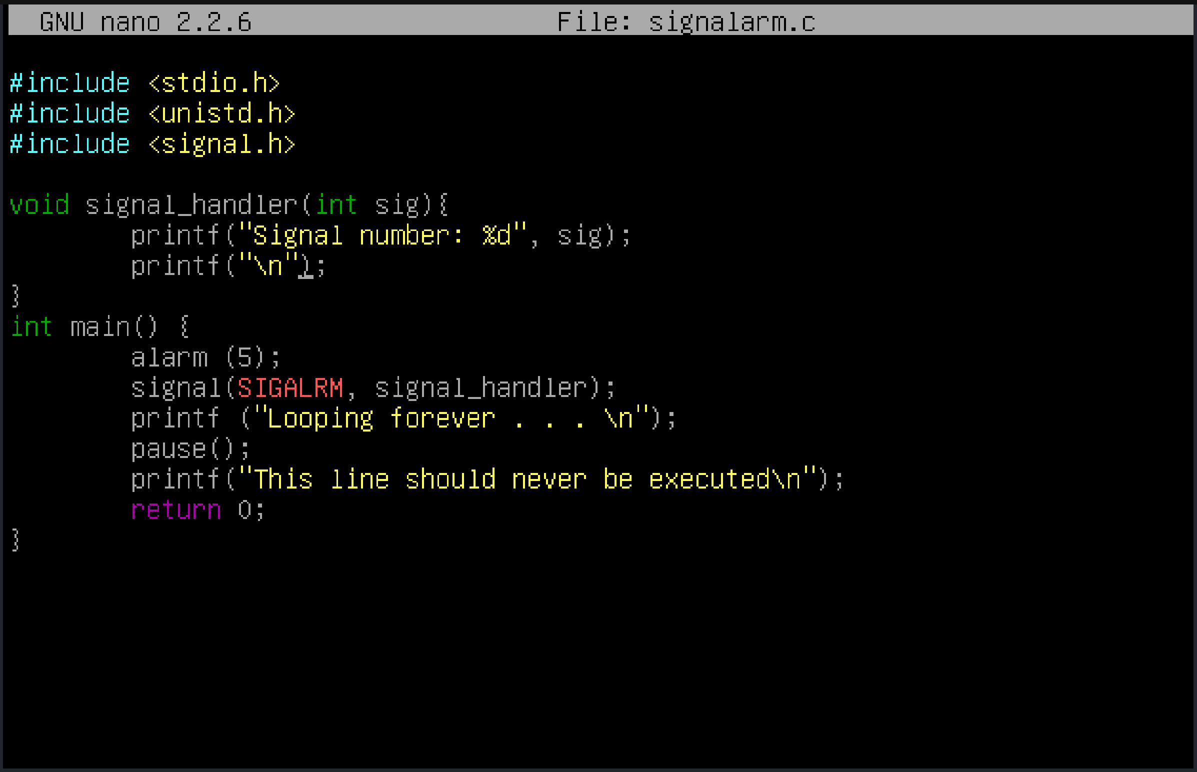


```
                alarm (5);
                printf ("Looping forever . . . \n");
                while (1);
                printf("This line should never be executed\n");
                return 0;
        }

                                                        [ Wrote 9 lines ]

        root@debian:~# make signalarm
        cc      signalarm.c   -o signalarm
        root@debian:~# ./signalarm
        Looping forever . . .
        Alarm clock
        root@debian:~# _
```

☑ Modify the given program by handling SIGALRM

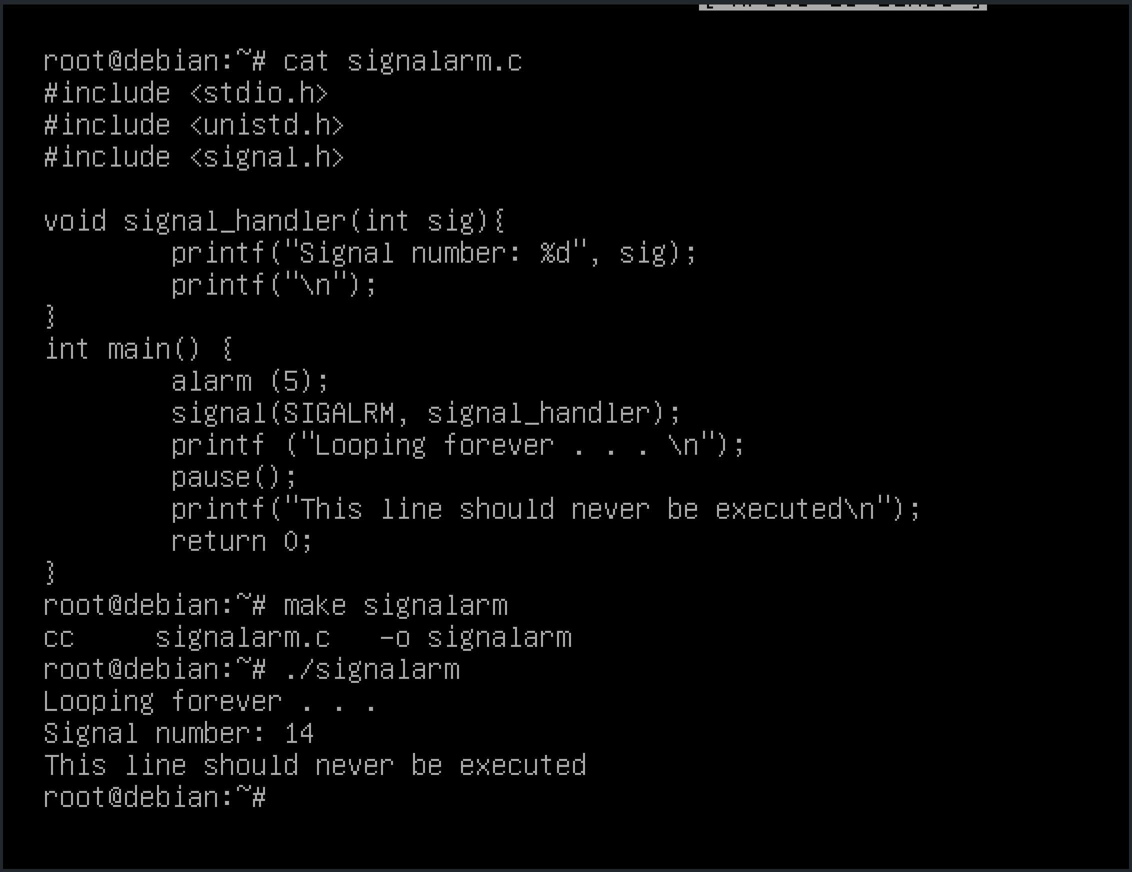
☑ `[FILL HERE with your source code.]`

The code is here:



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

void signal_handler(int sig){
        printf("Signal number: %d", sig);
        printf("\n");
}
int main() {
        alarm (5);
        signal(SIGALRM, signal_handler);
        printf ("Looping forever . . . \n");
        pause();
        printf("This line should never be executed\n");
        return 0;
}
```

The result:

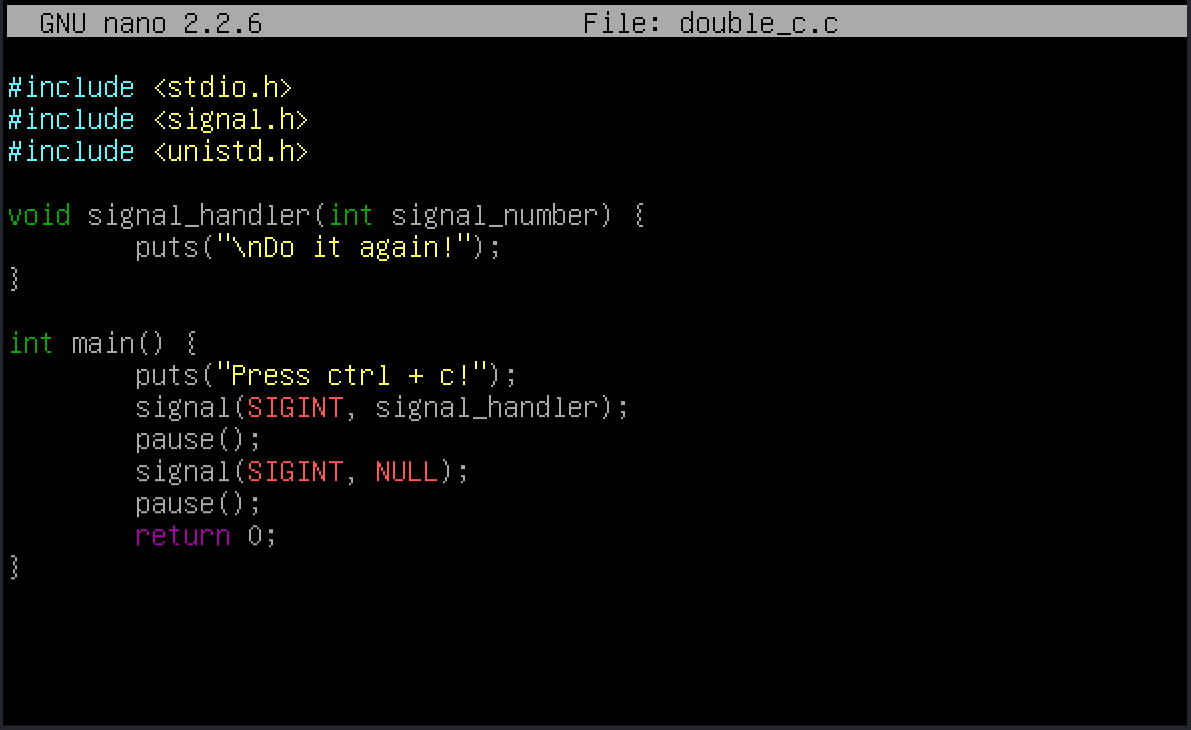


```
root@debian:~# cat signalarm.c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void signal_handler(int sig){
        printf("Signal number: %d", sig);
        printf("\n");
}
int main() {
        alarm (5);
        signal(SIGALRM, signal_handler);
        printf ("Looping forever . . . \n");
        pause();
        printf("This line should never be executed\n");
        return 0;
}
root@debian:~# make signalarm
cc     signalarm.c   -o signalarm
root@debian:~# ./signalarm
Looping forever . . .
Signal number: 14
This line should never be executed
root@debian:~#
```

- ☑ Write a program that handles Ctrl + C

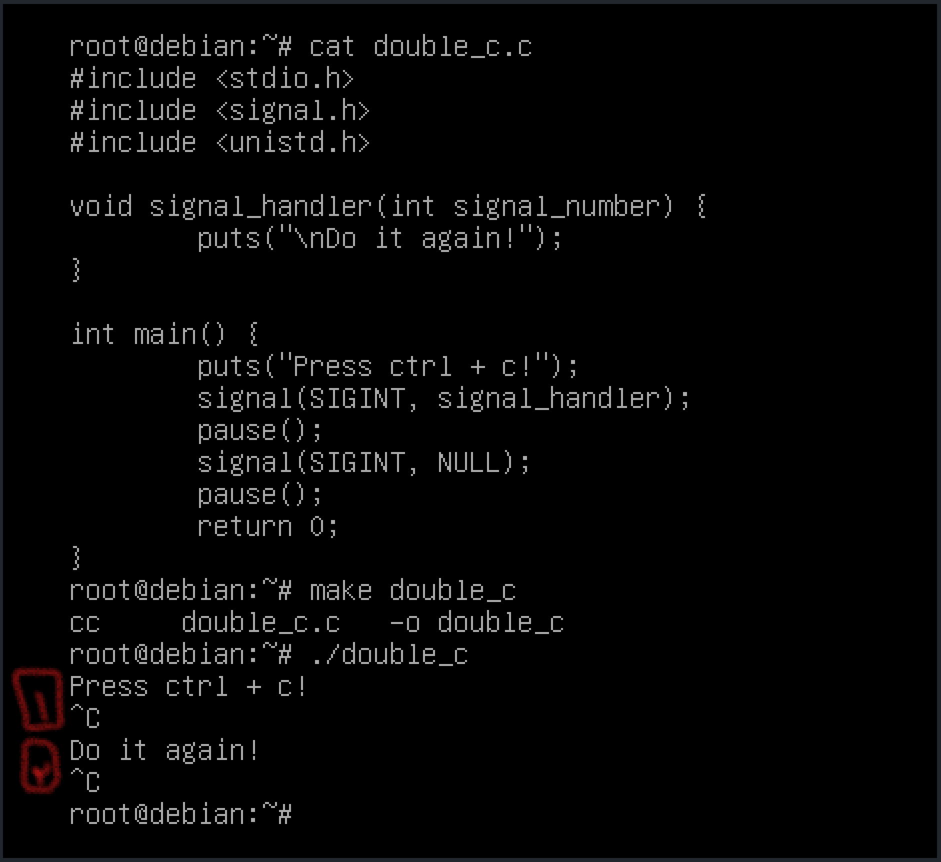
  - ☑ `[FILL HERE with your source code.]`

    The code is here:



```
  GNU nano 2.2.6                    File: double_c.c

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

void signal_handler(int signal_number) {
        puts("\nDo it again!");
}

int main() {
        puts("Press ctrl + c!");
        signal(SIGINT, signal_handler);
        pause();
        signal(SIGINT, NULL);
        pause();
        return 0;
}
```

The result:



```
root@debian:~# cat double_c.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int signal_number) {
        puts("\nDo it again!");
}

int main() {
        puts("Press ctrl + c!");
        signal(SIGINT, signal_handler);
        pause();
        signal(SIGINT, NULL);
        pause();
        return 0;
}
root@debian:~# make double_c
cc     double_c.c   -o double_c
root@debian:~# ./double_c
Press ctrl + c!
^C
Do it again!
^C
root@debian:~#
```

Amirreza81 added the documentation label 5 days ago

Amirreza81 assigned AMshoka 5 days ago

Assignees

AMshoka

Labels

documentation

Projects

None yet

Milestone

No milestone

Development

Create a branch for this issue or link a pull request.

2 participants

Pin issue