SIGNAL HANDLING

OPERATING SYSTEM (LAB 13)

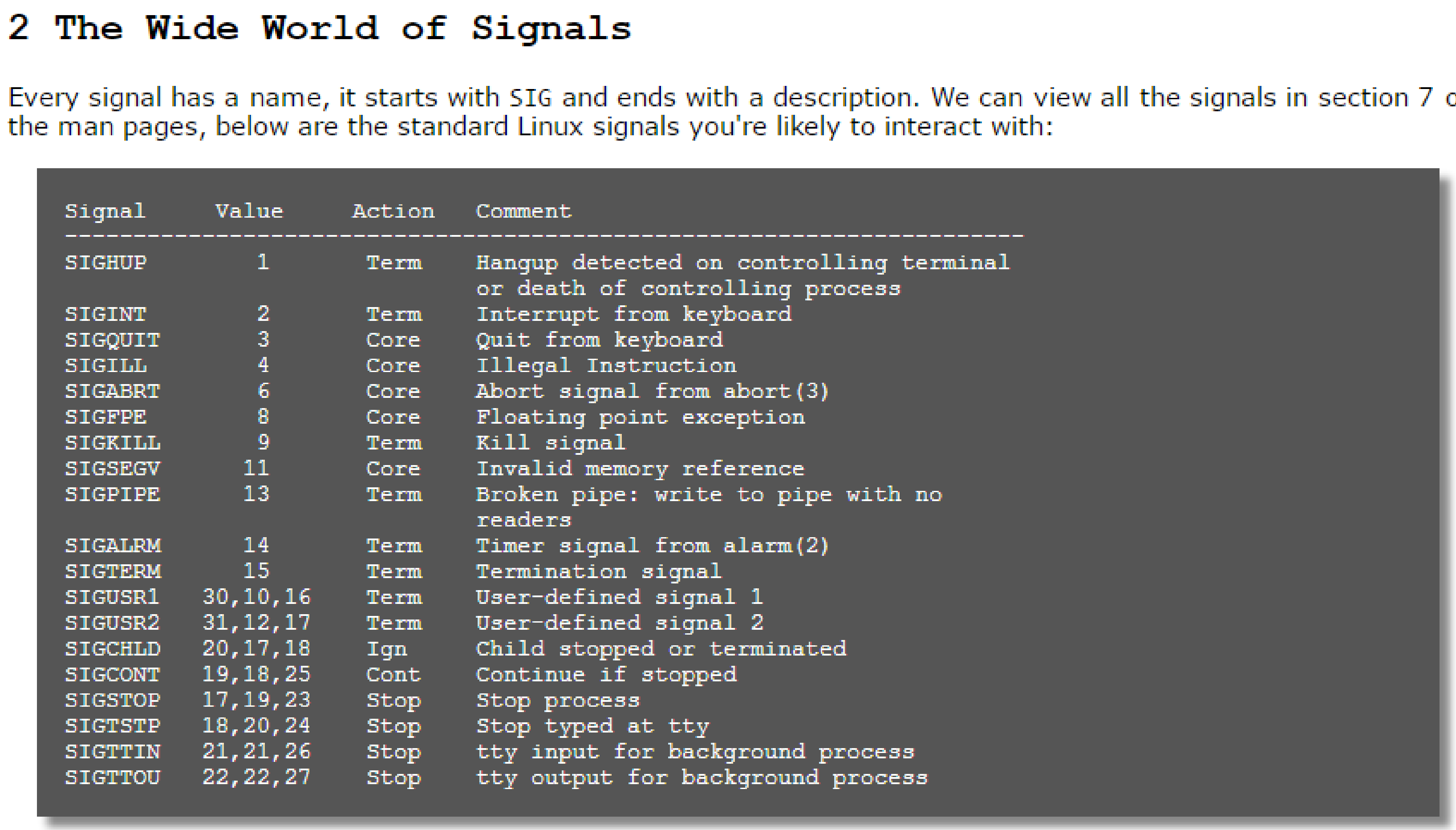
Section (A & E)

25/4/2016

SIGNAL HANDLING

- **1 What are signals and how are they used**
  - A **signal** is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and the hardware. A signal is an **interrupt** in the sense that it can change the flow of the program
  - when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

SIGNAL HANDLING

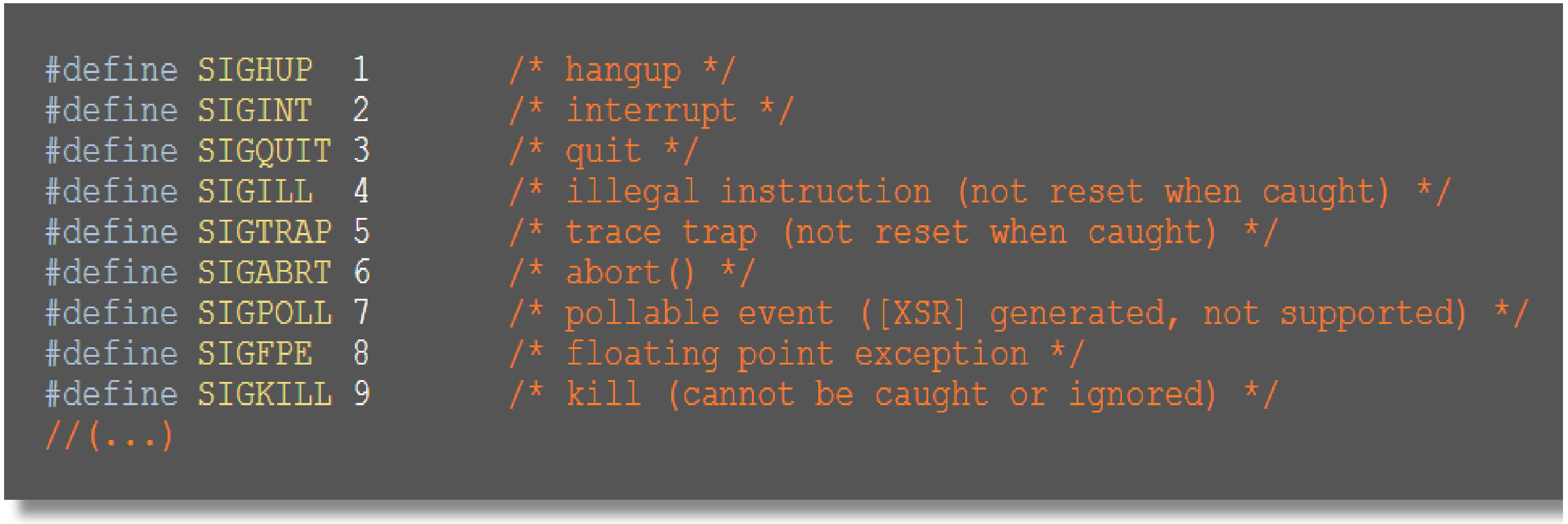


2 The Wide World of Signals

Every signal has a name, it starts with SIG and ends with a description. We can view all the signals in section 7 of the man pages, below are the standard Linux signals you're likely to interact with:

```
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 tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process
```

Signal Names and Values sys/signal.h



```
#define SIGHUP  1       /* hangup */
#define SIGINT  2       /* interrupt */
#define SIGQUIT 3       /* quit */
#define SIGILL  4       /* illegal instruction (not reset when caught) */
#define SIGTRAP 5       /* trace trap (not reset when caught) */
#define SIGABRT 6       /* abort() */
#define SIGPOLL 7       /* pollable event ([XSR] generated, not supported) */
#define SIGFPE  8       /* floating point exception */
#define SIGKILL 9       /* kill (cannot be caught or ignored) */
//(...)
```

Exercise (Command line)

1) Preparing for the kill (use two terminal)

loop.c

```
int main(){ while(1); }
```

```
killall -SIGSEGV loop
```

```
killall -SIGSTOP loop
```

```
killall -SIGCONT loop
```

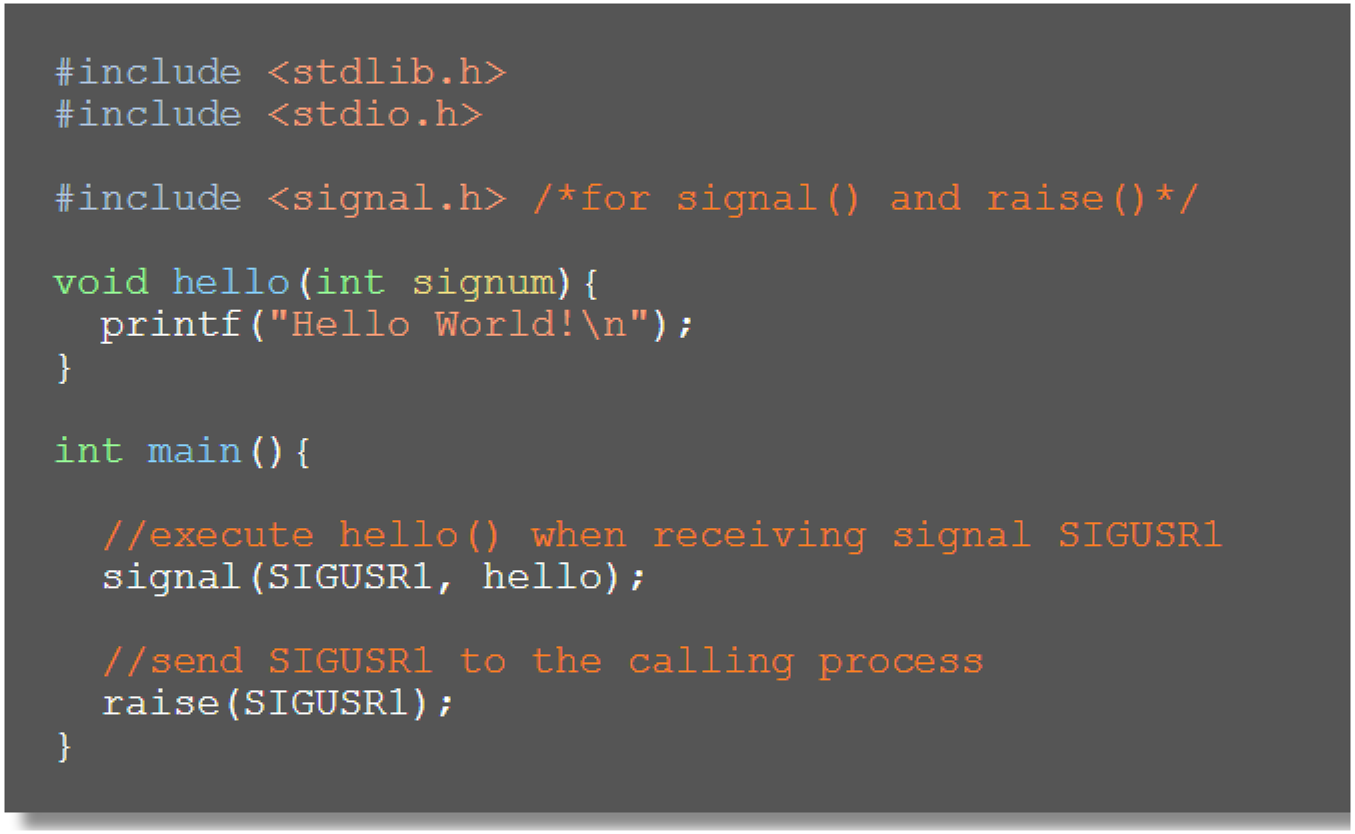
```
killall -SIGINT loop
```

-Check output using signal in command line

Handling and Generating Signals using signal API

- **Hello world of Signal Handling**
  - The primary system call for signal handling is signal(), which given a signal and function, will execute the function whenever the signal is delivered. This function is called the **signal handler** because it *handles* the signal.
    The signal() function has a strange declaration:
  - int signal(int signum, void (*handler)(int))
  - signal takes two arguments: the first argument is the signal number, such as SIGSTOP or SIGINT, and the second is a reference to a handler function

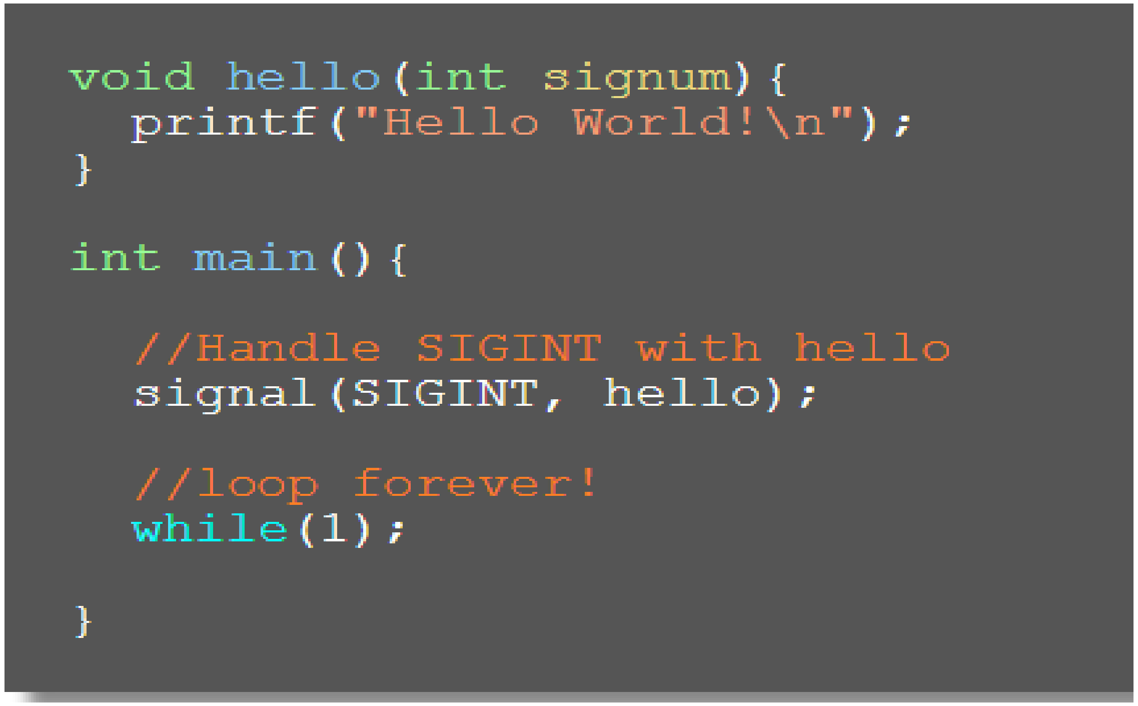
Hello world example



```c
#include <stdlib.h>
#include <stdio.h>

#include <signal.h> /*for signal() and raise()*/

void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //execute hello() when receiving signal SIGUSR1
  signal(SIGUSR1, hello);

  //send SIGUSR1 to the calling process
  raise(SIGUSR1);
}
```

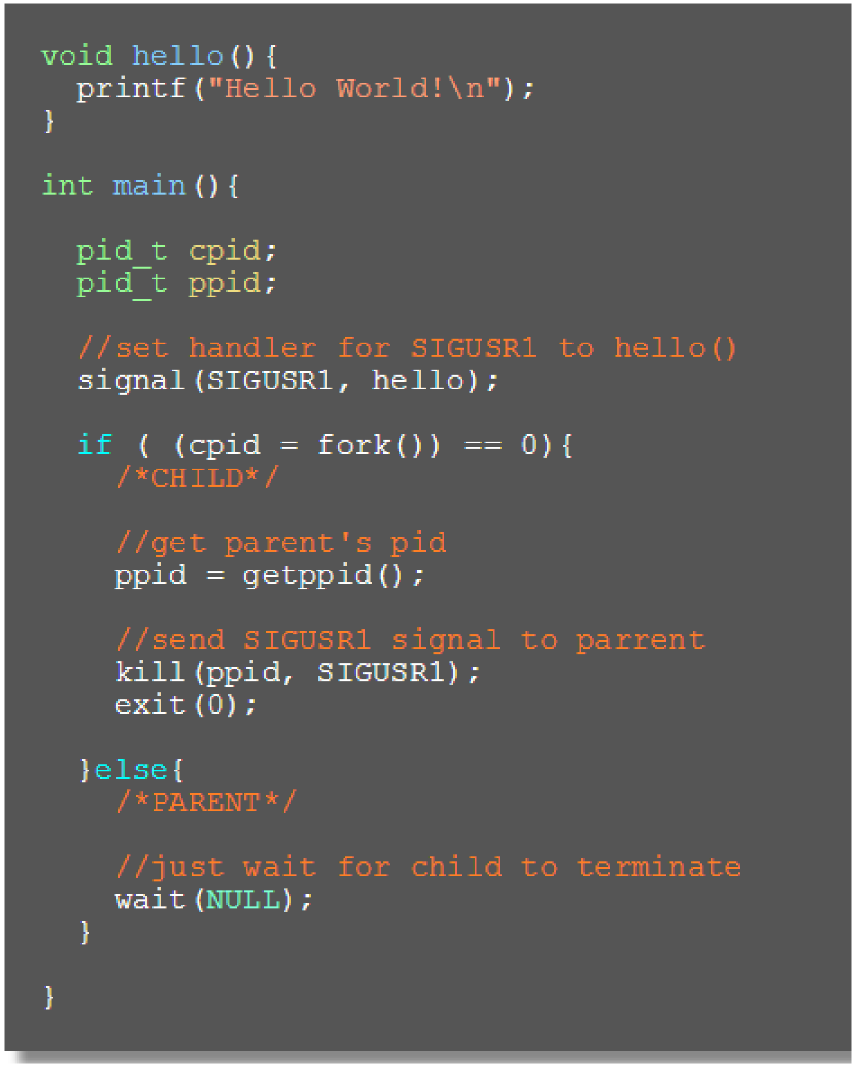
Asynchronous Execution hello world

– The execution of the signal handler is asynchronous, which means the current state of the program will be paused while the signal handler executes, and then execution will resume from the pause point, much like context switching.



```
void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //Handle SIGINT with hello
  signal(SIGINT, hello);

  //loop forever!
  while(1);

}
```

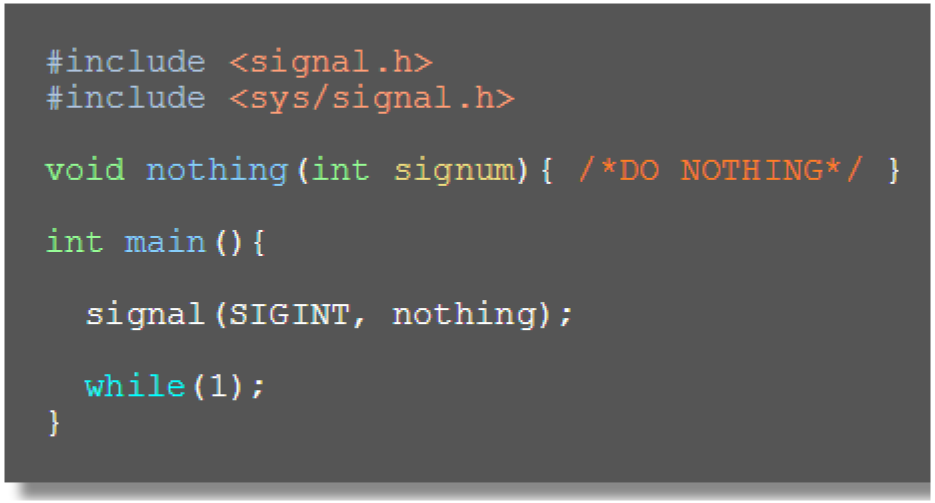
Inter Process Communication



```
void hello(){
  printf("Hello World!\n");
}

int main(){

  pid_t cpid;
  pid_t ppid;

  //set handler for SIGUSR1 to hello()
  signal(SIGUSR1, hello);

  if ( (cpid = fork()) == 0){
    /*CHILD*/

    //get parent's pid
    ppid = getppid();

    //send SIGUSR1 signal to parrent
    kill(ppid, SIGUSR1);
    exit(0);

  }else{
    /*PARENT*/

    //just wait for child to terminate
    wait(NULL);
  }

}
```

– One process can send a signal to another indicating that an action should be taken. To send a signal to a particular process, we use the kill() system call. The function declaration is below.
– int kill(pid_t pid, int signum);

Ignoring Signals

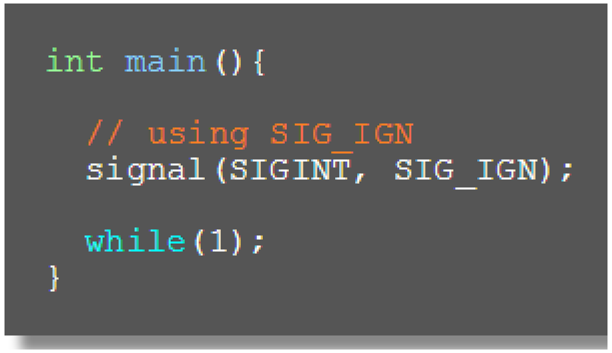


```
#include <signal.h>
#include <sys/signal.h>

void nothing(int signum){ /*DO NOTHING*/ }

int main(){

  signal(SIGINT, nothing);

  while(1);
}
```

here is a program that will ignore SIGINT by handling the signal and do nothing :

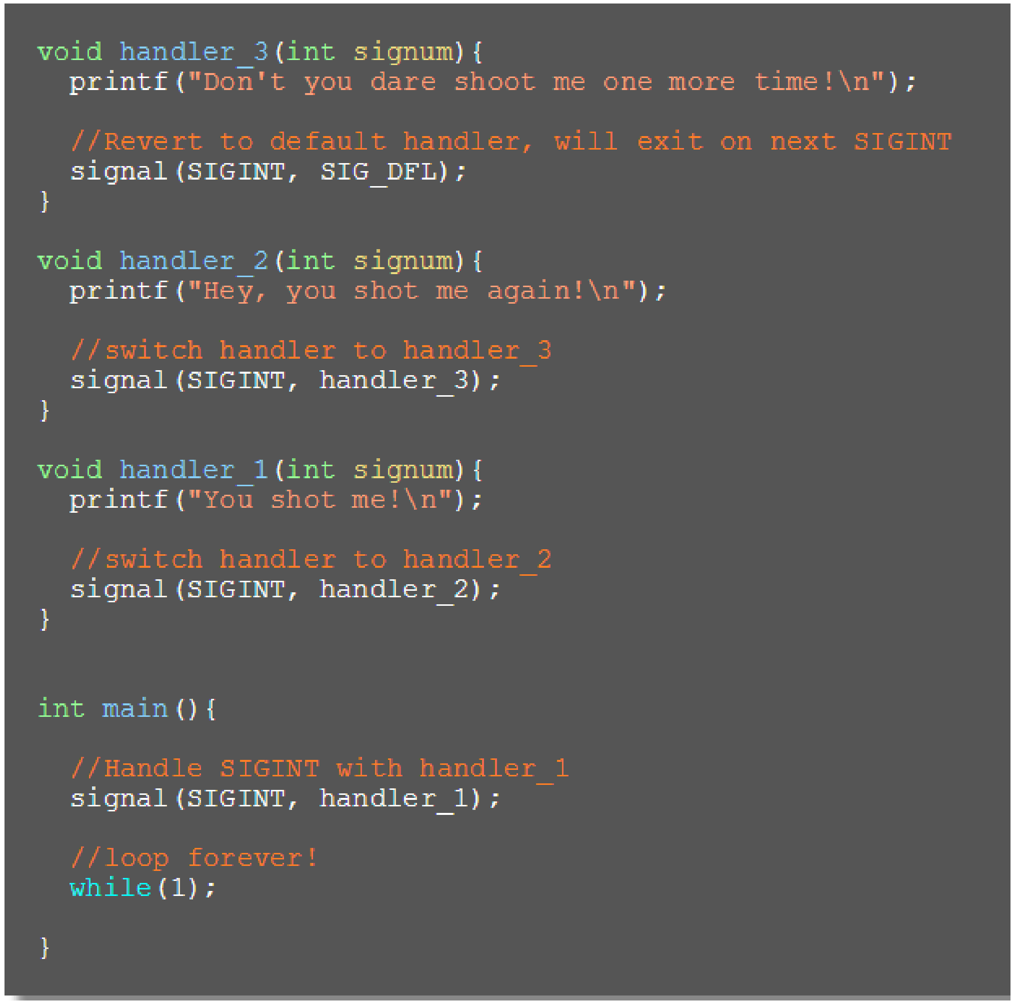
-SIG_IGN : Ignore the signal

-SIG_DFL : Replace the current signal handler with the default handler



```
int main(){

  // using SIG_IGN
  signal(SIGINT, SIG_IGN);

  while(1);
}
```

Changing and Reverting to the default handler



```
void handler_3(int signum){
  printf("Don't you dare shoot me one more time!\n");

  //Revert to default handler, will exit on next SIGINT
  signal(SIGINT, SIG_DFL);
}

void handler_2(int signum){
  printf("Hey, you shot me again!\n");

  //switch handler to handler_3
  signal(SIGINT, handler_3);
}

void handler_1(int signum){
  printf("You shot me!\n");

  //switch handler to handler_2
  signal(SIGINT, handler_2);
}


int main(){

  //Handle SIGINT with handler_1
  signal(SIGINT, handler_1);

  //loop forever!
  while(1);

}
```

1) The program first initiates handler_1() as the signal handler for SIGINT.

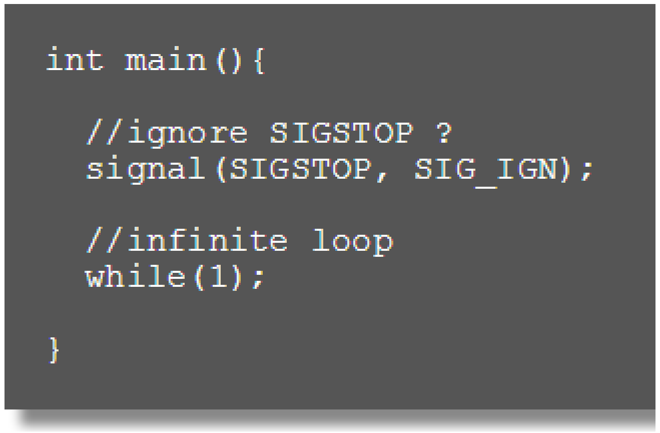
2) After the first Ctrl-c, in the signal handler, the handler is changed to handler_2(), and after the second Ctrl-c,

3) it is change again to handler_3() from handler_2().

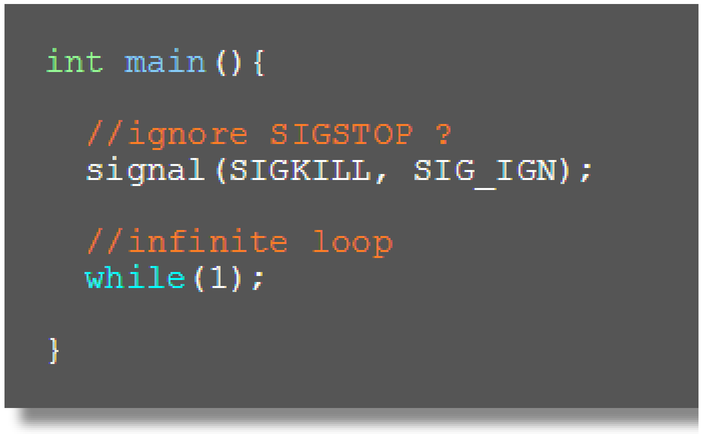
4) Finally, in handler_3() the default signal handler is reestablished, which is to terminate on SIGINT, and that is what we see in the output:

Some signals are more equal than others

– The two signals that can never be ignored or handled are: SIGKILL and SIGTSTOP. Let's look at an example:



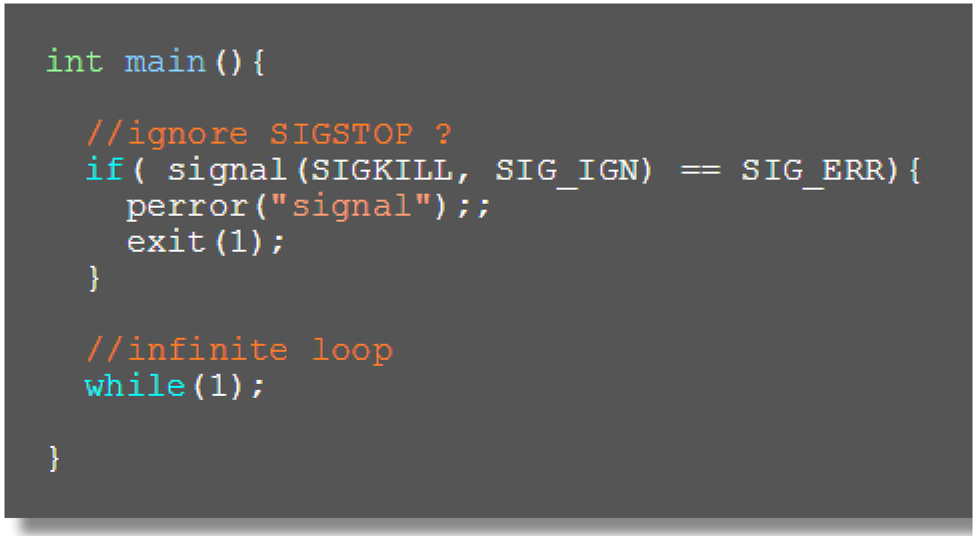
```
int main(){

  //ignore SIGSTOP ?
  signal(SIGSTOP, SIG_IGN);

  //infinite loop
  while(1);

}
```



```
int main(){

  //ignore SIGSTOP ?
  signal(SIGKILL, SIG_IGN);

  //infinite loop
  while(1);

}
```

Checking Errors of signal()

– special value is used SIG_ERR which we can compare the return value of signal(). Here, again, is the program where we try and ignore SIGKILL, but this time with proper error checking:



```
int main(){

  //ignore SIGSTOP ?
  if( signal(SIGKILL, SIG_IGN) == SIG_ERR){
    perror("signal");;
    exit(1);
  }

  //infinite loop
  while(1);

}
```

The invalid argument is SIGKILL which cannot be handled or ignored. It can only KILL!