CS 332/532 Systems Programming

Lecture 24

Signals

Professor : Mahmut Unan – UAB CS

Agenda

- Signals

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
    - Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
    - Performing some default action
    - Executing a signal-handler function
    - Ignoring the signal

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Table 6.2

UNIX Signals

(Table can be found on page 288 in textbook)

Real-time (RT) Signals

- Linux includes all of the concurrency mechanisms found in other UNIX systems
- Linux also supports real-time (RT) signals
- RT signals differ from standard UNIX signals in three primary ways:
    - Signal delivery in priority order is supported
    - Multiple signals can be queued
    - With standard signals, no value or message can be sent to the target process – it is only a notification
    - With RT signals it is possible to send a value along with the signal

Semaphores

- User level:
    - Linux provides a semaphore interface corresponding to that in UNIX SVR4

– Internally:
    - Implemented as functions within the kernel and are more efficient than user-visable semaphores

– Three types of kernel semaphores:
    - Binary semaphores
    - Counting semaphores
    - Reader-writer semaphores

Traditional Semaphores	
void sema_init(struct semaphore *sem, int count)	Initializes the dynamically created semaphore to the given count
void init_MUTEX(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
void init_MUTEX_LOCKED(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 0 (initially locked)
void down(struct semaphore *sem)	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
int down_interruptible(struct semaphore *sem)	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received
int down_trylock(struct semaphore *sem)	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
void up(struct semaphore *sem)	Releases the given semaphore
Reader-Writer Semaphores	
void init_rwsem(struct rw_semaphore, *rwsem)	Initializes the dynamically created semaphore with a count of 1
void down_read(struct rw_semaphore, *rwsem)	Down operation for readers
void up_read(struct rw_semaphore, *rwsem)	Up operation for readers
void down_write(struct rw_semaphore, *rwsem)	Down operation for writers
void up_write(struct rw_semaphore, *rwsem)	Up operation for writers

Table 6.5

Linux Semaphores

(Table can be found on page 293 in textbook)

Table 6.6

Linux Memory Barrier Operations

rmb()	Prevents loads from being reordered across the barrier
wmb()	Prevents stores from being reordered across the barrier
mb()	Prevents loads and stores from being reordered across the barrier
Barrier()	Prevents the compiler from reordering loads or stores across the barrier
smp_rmb()	On SMP, provides a rmb() and on UP provides a barrier()
smp_wmb()	On SMP, provides a wmb() and on UP provides a barrier()
smp_mb()	On SMP, provides a mb() and on UP provides a barrier()

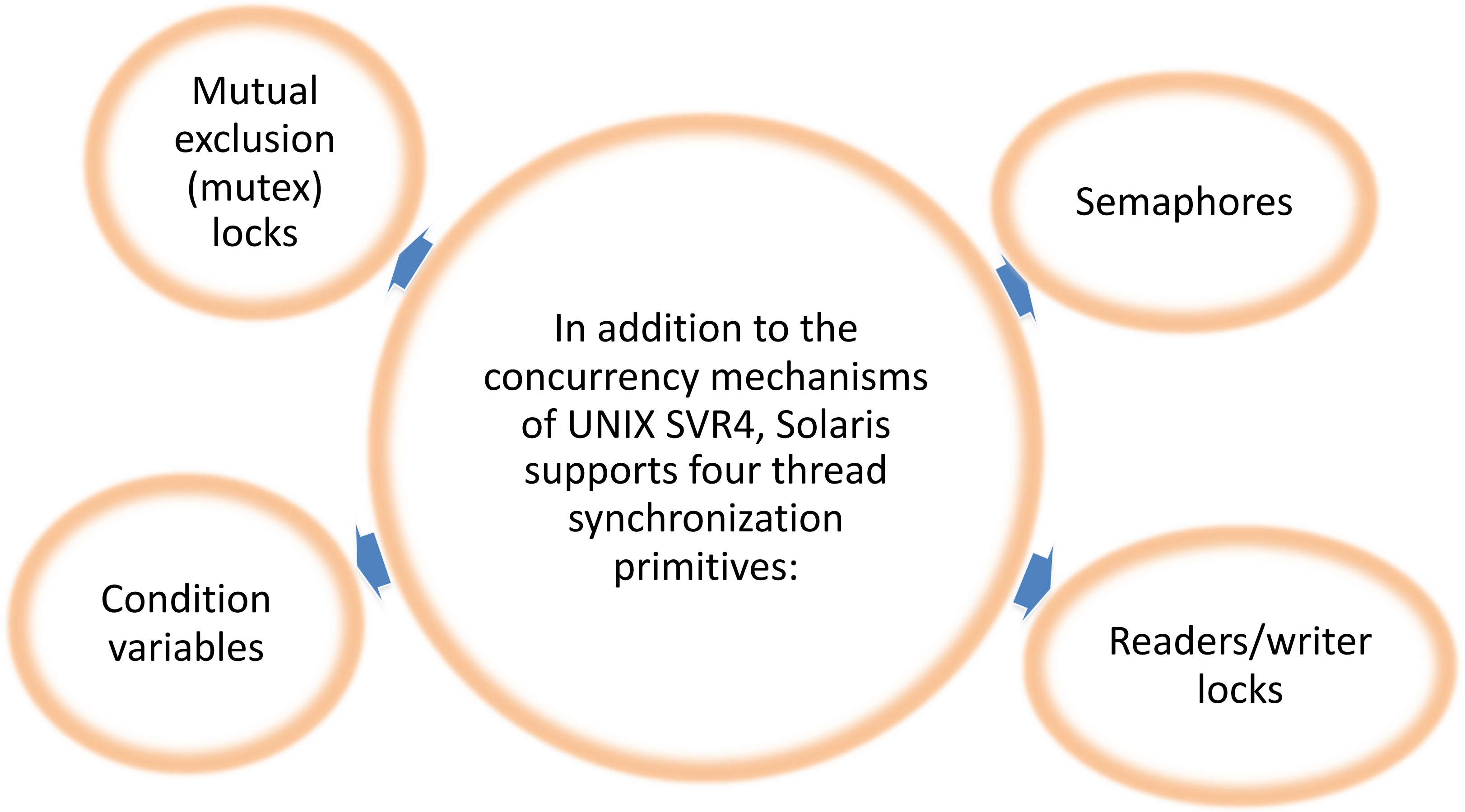
SMP = symmetric multiprocessor
UP = uniprocessor

(Table can be found on page 294 in textbook)

Read-Copy-Update (RCU)

- The RCU mechanism is an advanced lightweight synchronization mechanism which was integrated into the Linux kernel in 2002
- The RCU is used widely in the Linux kernel
- RCU is also used by other operating systems
- There is a userspace RCU library called liburcu
- The shared resources that the RCU mechanism protects must be accessed via a pointer
- The RCU mechanism provides access for multiple readers and writers to a shared resource

Synchronization Primitives


In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:
Mutual exclusion (mutex) locks
Semaphores
Readers/writer locks
Condition variables

Mutual Exclusion (MUTEX) Lock

- Used to ensure only one thread at a time can access the resource protected by the mutex
- The thread that locks the mutex must be the one that unlocks it
- A thread attempts to acquire a mutex lock by executing the `mutex_enter` primitive
- Default blocking policy is a spinlock
- An interrupt-based blocking mechanism is optional

Semaphores

Solaris provides classic counting semaphores with the following primitives:

- sema_p() Decrements the semaphore, potentially blocking the thread
- sema_v() Increments the semaphore, potentially unblocking a waiting thread
- sema_tryp() Decrements the semaphore if blocking is not required

Readers/Writer Locks

- Allows multiple threads to have simultaneous read-only access to an object protected by the lock
- Allows a single thread to access the object for writing at one time, while excluding all readers
    - When lock is acquired for writing it takes on the status of `write lock`
    - If one or more readers have acquired the lock its status is `read lock`

Condition Variables

A condition variable is used to wait until a particular condition is true

Condition variables must be used in conjunction with a mutex lock

Signals

- Signals are software interrupts that provide a mechanism to deal with asynchronous events (e.g., a user pressing Control-C to interrupt a program that the shell is currently executing).
- A signal is a notification to a process that an event has occurred that requires the attention of the process (this typically interrupts the normal flow of execution of the interrupted process).
- Signals can also be used as a synchronization technique or even as a simple form of interprocess communication.
- Signals could be generated by hardware interrupts, the program itself, other programs, the OS kernel, or by the user.

- All these signals have a unique symbolic name and starts with SIG. The standard signals (also called POSIX reliable signals) are defined in the header file *<signal.h>*.
- Here are some examples:
- `SIGINT:` Interrupt a process from keyboard (e.g., pressing Control-C). The process is terminated.
- `SIGQUIT:` Interrupt a process to quit from keyboard (e.g., pressing Control-/). The process is terminated and a core file is generated.
- `SIGTSTP:` Interrupt a process to stop from keyboard (e.g., pressing Control-Z). The process is stopped from executing.
- `SIGUSR1` and `SIGUSR2`: These are user-defined signals, for use in application programs.

- ***NOTE***: Please see Section 10.2 and Figure 10.1 in the textbook for a complete list of UNIX System signals. You can also find more details using *man 7 signal* on any CS UNIX system (*man signal* on Mac).
- After a signal is generated, it is delivered to a process to perform some action in response to this signal.
- Since signals are asynchronous events, a process has to decide ahead of time how to respond when the particular signal is delivered.
- There are three different options possible when a signal is delivered to a process:

- Perform the default action. Most signals have a default action associated with them, if the process does not change the default behavior then the default action specific to that particular signal will occur.
  - The predefined default signal is specified as `SIG_DFL`.
- Ignore the signal. If a signal is ignored, then the default action is performed.
  - The predefined ignore signal handler is specified as `SIG_IGN`.
- Catch and Handle the signal. It is also possible to override the default action and invoke a specific user-defined task when a signal is received.
  - This is usually performed by invoking a signal handler using *`signal()`* or *`sigaction()`* system calls.
- However, the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

signal()

- signal - ANSI C signal handling
-
  ```
  typedef void (*sighandler_t)(int);
  ```

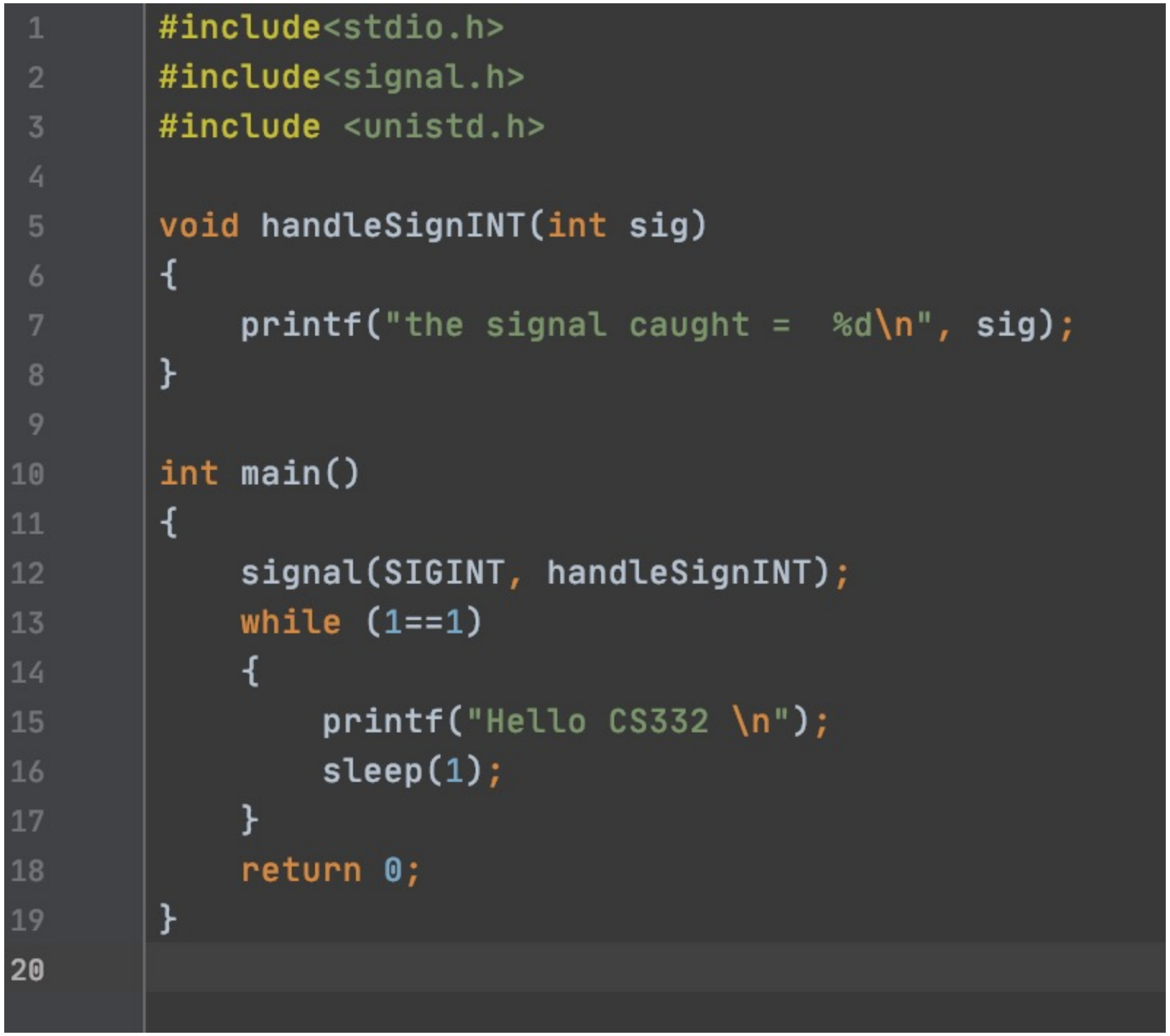
-
  ```
  sighandler_t signal(int signum, sighandler_t
  handler);
  ```

- **signal**() sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

Sending Signals Using The Keyboard

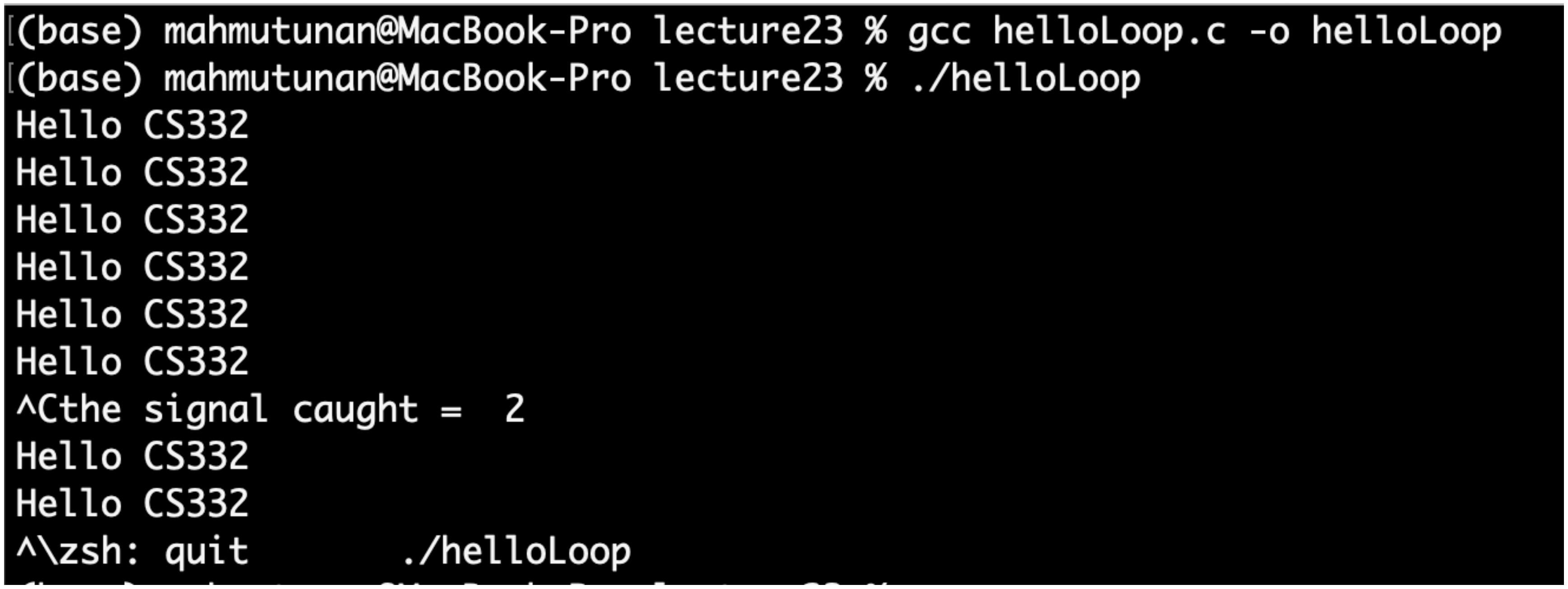
- Ctrl-C to send an INT signal (SIGINT) to the running process.
  - This signal causes the process to immediately terminate.
- Ctrl-Z to send a TSTP signal (SIGTSTP) to the running process.
  - This signal causes the process to suspend execution.
- Ctrl-\ to send a ABRT signal (SIGABRT) to the running process.
  - This signal causes the process to immediately terminate.
  - Ctrl-\ doing the same as Ctrl-C but it gives us some better flexibility.

infinite loop

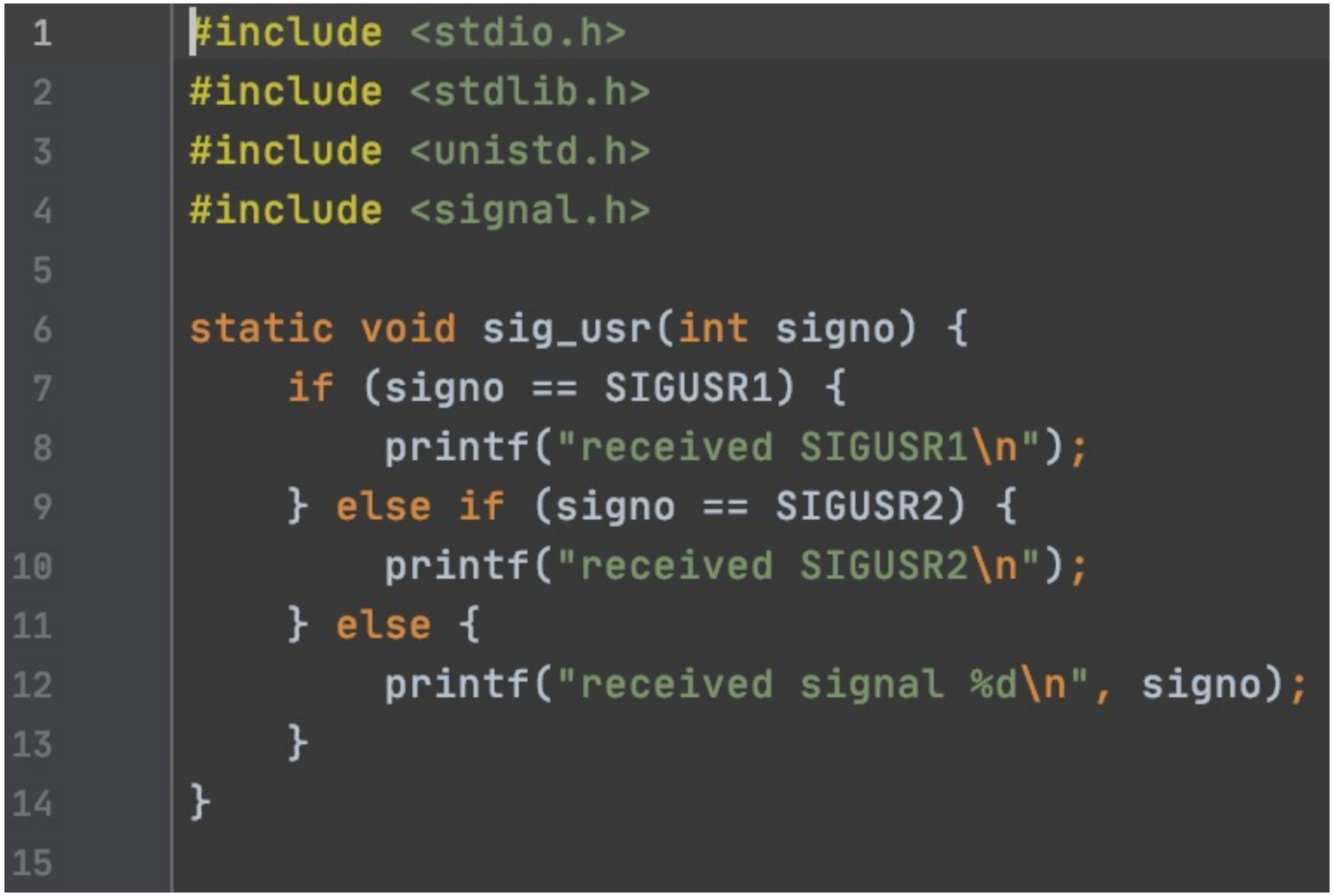


```c
#include<stdio.h>
#include<signal.h>
#include <unistd.h>

void handleSignINT(int sig)
{
    printf("the signal caught =  %d\n", sig);
}

int main()
{
    signal(SIGINT, handleSignINT);
    while (1==1)
    {
        printf("Hello CS332 \n");
        sleep(1);
    }
    return 0;
}
```

compile & run

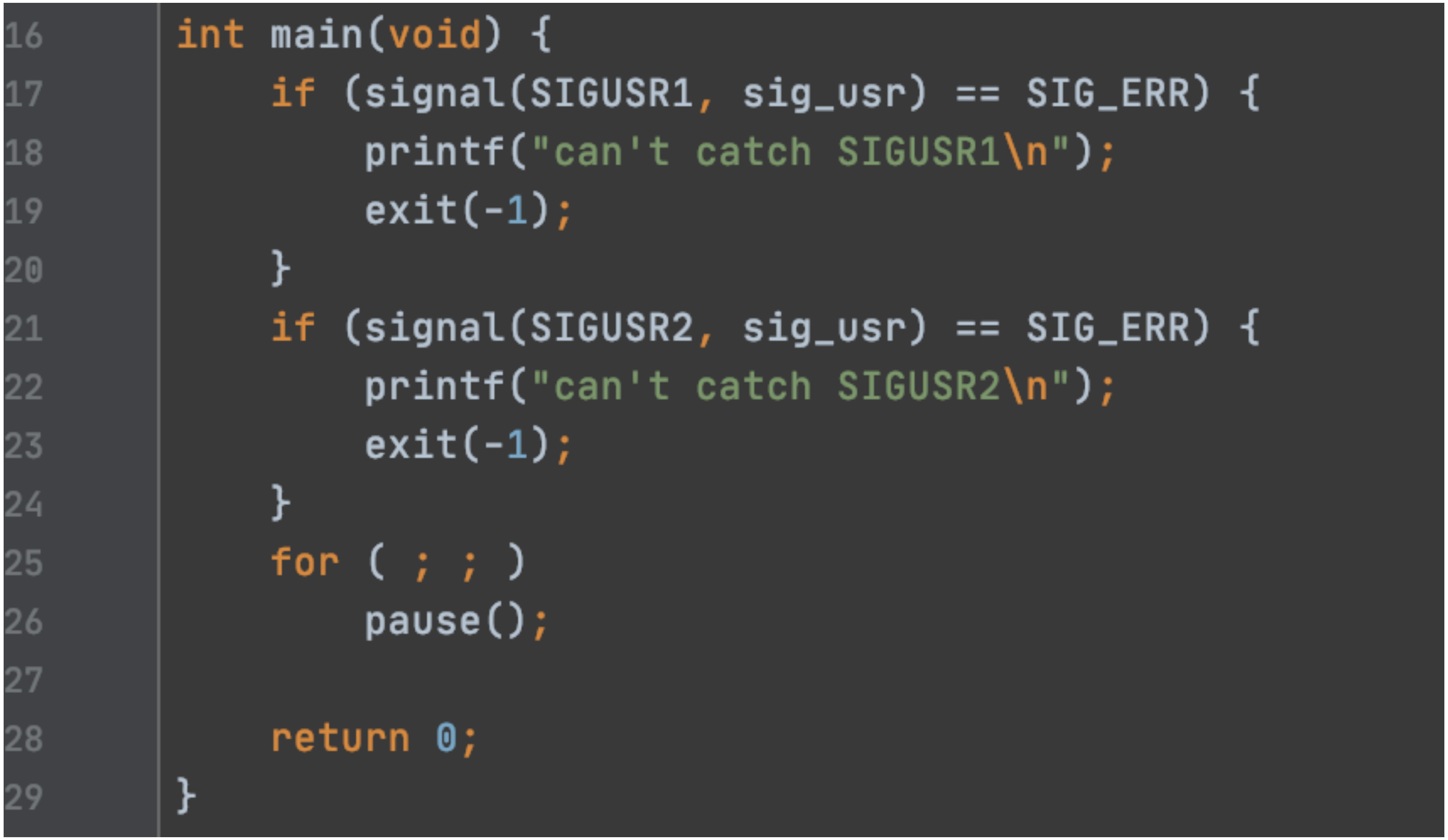

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc helloLoop.c -o helloLoop
(base) mahmutunan@MacBook-Pro lecture23 % ./helloLoop
Hello CS332
Hello CS332
Hello CS332
Hello CS332
Hello CS332
Hello CS332
^Cthe signal caught =  2
Hello CS332
Hello CS332
^\zsh: quit       ./helloLoop
```

- Now, let's write a program to handle a simple signal that catches either of the two user-defined signals and prints the signal number (see Figure 10.2 in the textbook).
- 1. Create a user-defined function (signal handler) that will be invoked when a signal is received:

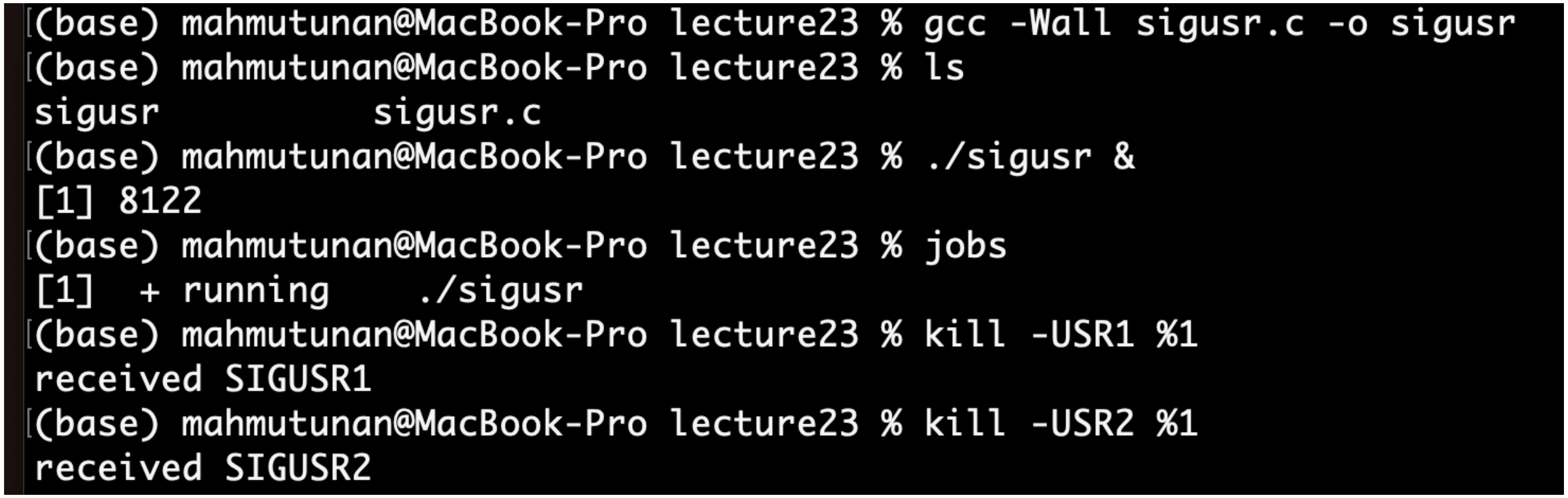


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

static void sig_usr(int signo) {
    if (signo == SIGUSR1) {
        printf("received SIGUSR1\n");
    } else if (signo == SIGUSR2) {
        printf("received SIGUSR2\n");
    } else {
        printf("received signal %d\n", signo);
    }
}

```

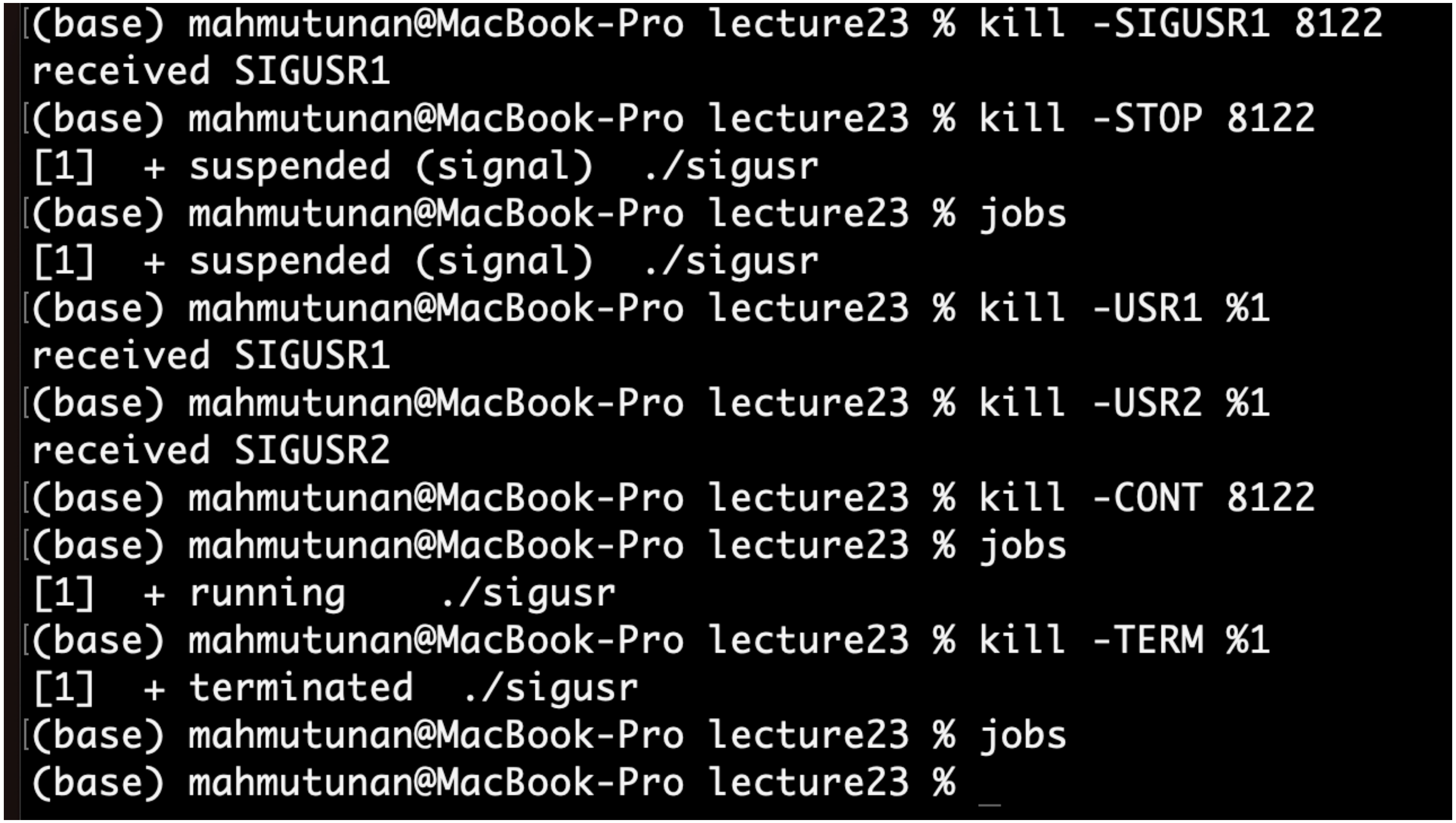
- 2. Now let’s call above user-defined signal.



```
int main(void) {
    if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR1\n");
        exit(-1);
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR2\n");
        exit(-1);
    }
    for ( ; ; )
        pause();

    return 0;
}
```



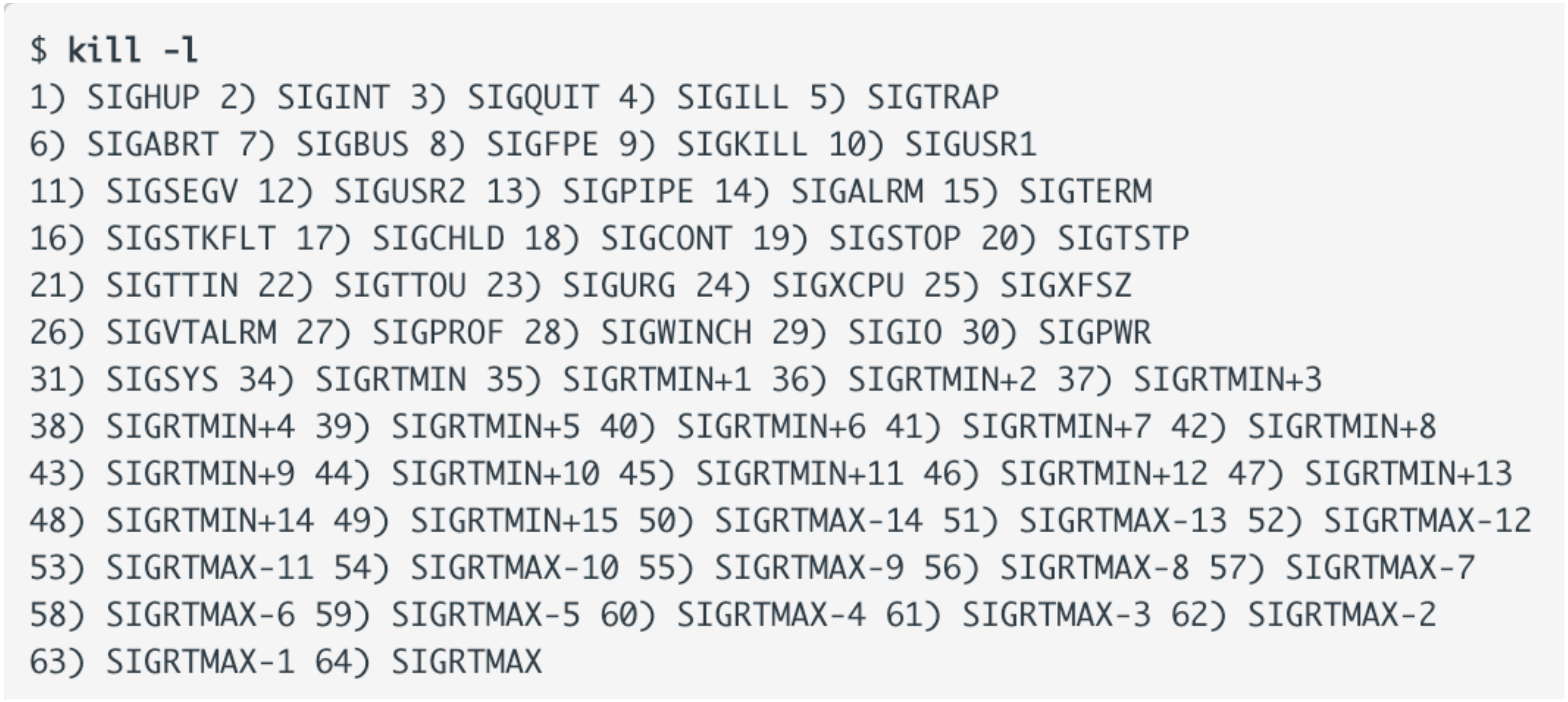
```
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigusr.c -o sigusr
(base) mahmutunan@MacBook-Pro lecture23 % ls
sigusr          sigusr.c
(base) mahmutunan@MacBook-Pro lecture23 % ./sigusr &
[1] 8122
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + running    ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
```



```
(base) mahmutunan@MacBook-Pro lecture23 % kill -SIGUSR1 8122
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -STOP 8122
[1]  + suspended (signal)  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + suspended (signal)  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
(base) mahmutunan@MacBook-Pro lecture23 % kill -CONT 8122
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + running    ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -TERM %1
[1]  + terminated  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % jobs
(base) mahmutunan@MacBook-Pro lecture23 % _
```

- In the example we used the *kill* command that we used in the previous lecture to generate the signal.
- While we used the kill command in the previous lecture to terminate a process using the TERM signal (the default signal if no signal is specified), the kill command could be used to generate any other signal that is supported by the kernel.

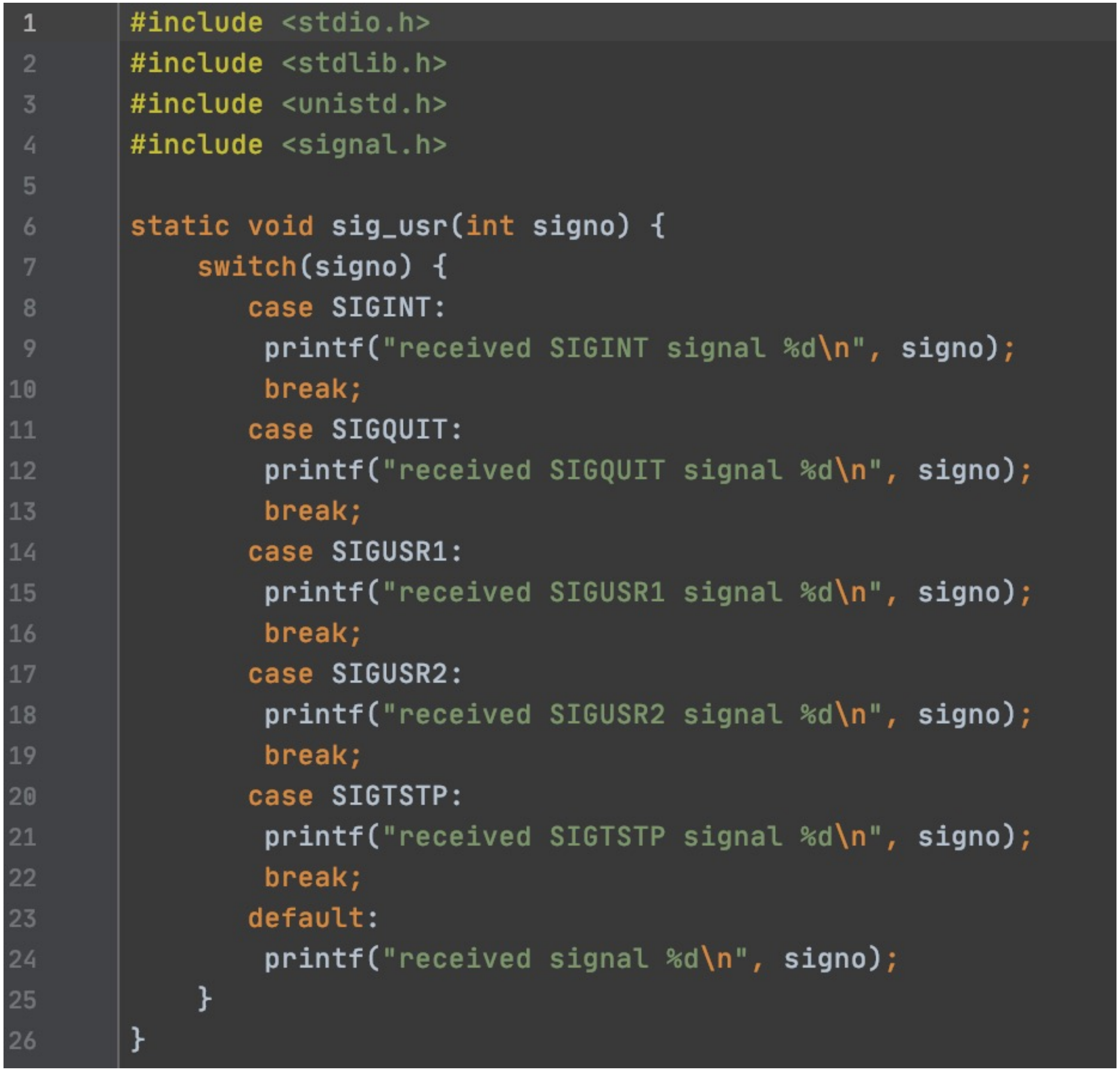
- You can list all signals that can be generated using *kill -l*. For example, on the CS Linux systems you see the following output:



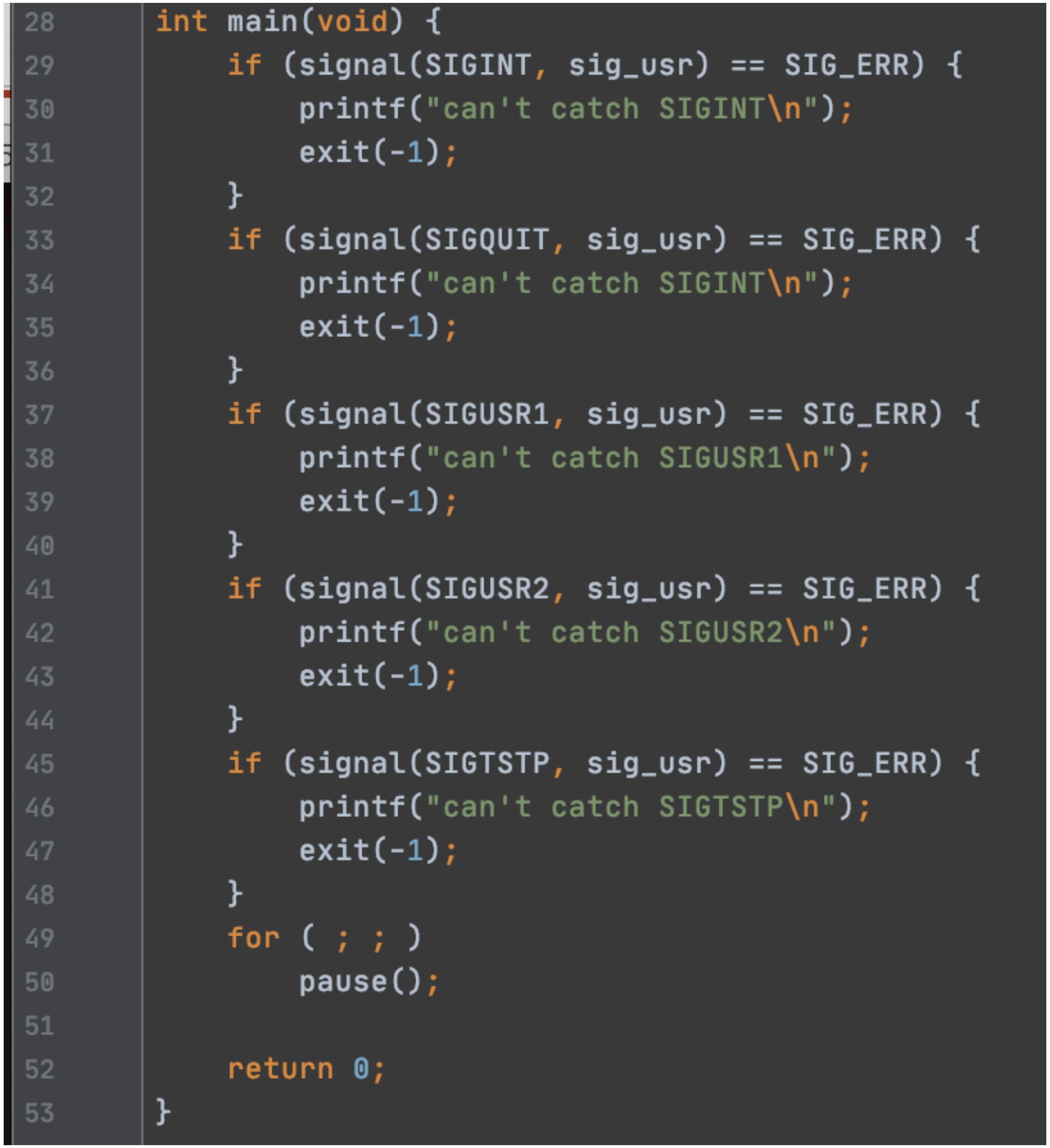
```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Exercise 2

- We will now extend the exercise 1 to handle other signal generated from keyboard such as Control-C (SIGINT), Control-Z (SIGTSTP), and Control-\ (SIGQUIT). In this example we will use a single signal handler to handle all these signals using a switch statement (instead of separate signal handlers).

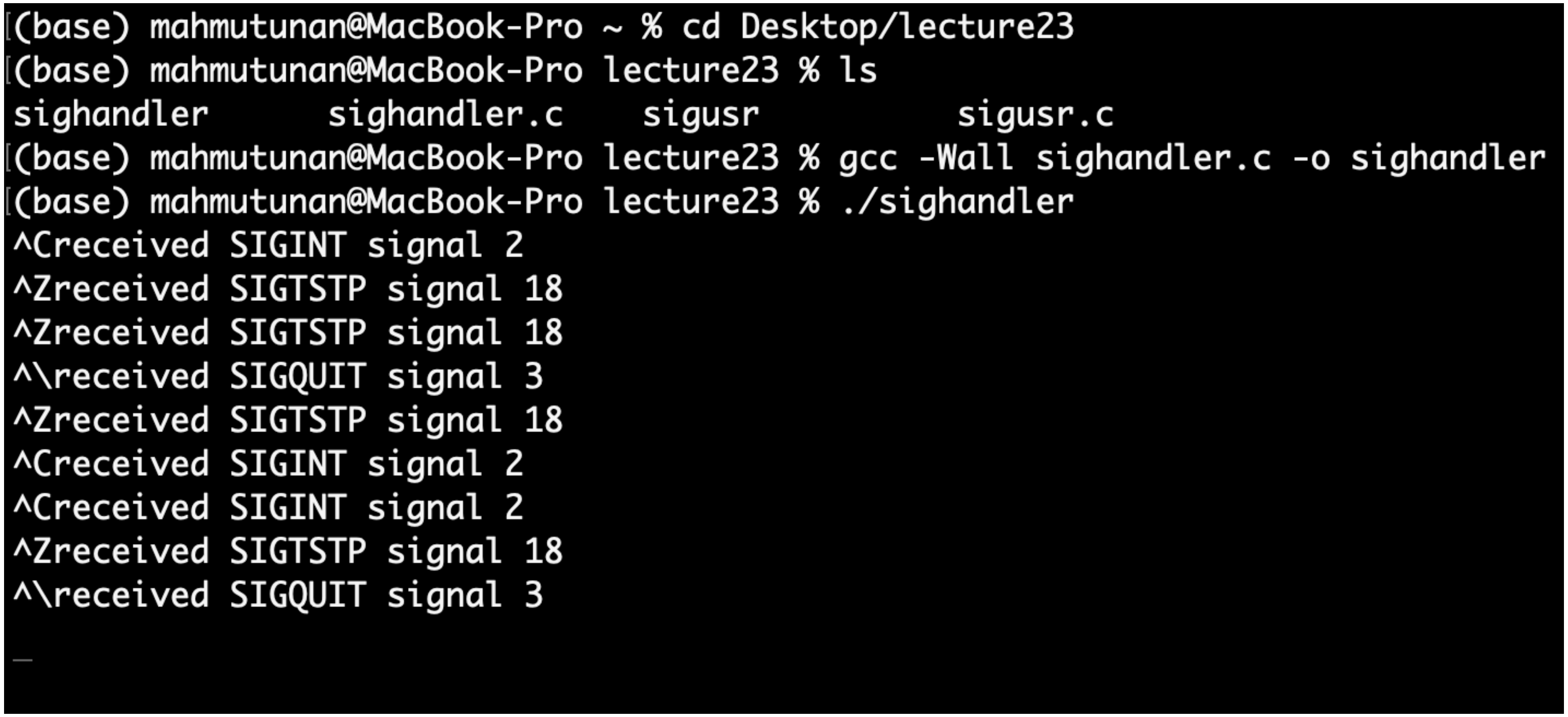


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

static void sig_usr(int signo) {
    switch(signo) {
       case SIGINT:
        printf("received SIGINT signal %d\n", signo);
        break;
       case SIGQUIT:
        printf("received SIGQUIT signal %d\n", signo);
        break;
       case SIGUSR1:
        printf("received SIGUSR1 signal %d\n", signo);
        break;
       case SIGUSR2:
        printf("received SIGUSR2 signal %d\n", signo);
        break;
       case SIGTSTP:
        printf("received SIGTSTP signal %d\n", signo);
        break;
       default:
        printf("received signal %d\n", signo);
    }
}
```



```c
int main(void) {
    if (signal(SIGINT, sig_usr) == SIG_ERR) {
        printf("can't catch SIGINT\n");
        exit(-1);
    }
    if (signal(SIGQUIT, sig_usr) == SIG_ERR) {
        printf("can't catch SIGINT\n");
        exit(-1);
    }
    if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR1\n");
        exit(-1);
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR2\n");
        exit(-1);
    }
    if (signal(SIGTSTP, sig_usr) == SIG_ERR) {
        printf("can't catch SIGTSTP\n");
        exit(-1);
    }
    for ( ; ; )
        pause();

    return 0;
}
```

compile & run



```
[(base) mahmutunan@MacBook-Pro ~ % cd Desktop/lecture23
[(base) mahmutunan@MacBook-Pro lecture23 % ls
sighandler      sighandler.c    sigusr          sigusr.c
[(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sighandler.c -o sighandler
[(base) mahmutunan@MacBook-Pro lecture23 % ./sighandler
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^Zreceived SIGTSTP signal 18
^\received SIGQUIT signal 3
^Zreceived SIGTSTP signal 18
^Creceived SIGINT signal 2
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^\received SIGQUIT signal 3

_
```

infinite loop

- Notice that we have replaced the default signal handlers to kill this job from the keyboard and the program is in an infinite loop with pause. As a result we cannot terminate this program from the keyboard. We have to login to the specific machine this process is running and kill this process using either *kill* or *top* commands. You can follow the examples from the first part and kill this process.