CS 3320
Operating Systems

System Calls

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Outline

- What is a system call?
  - Kernel space vs user space
  - System call vs library call
  - What service can system calls provide?
  - System call naming, input, output
- How to add a new system call
  - Example
  - Project 1

User space vs. Kernel space

- **Kernel space** is strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers.
- **User space** is the area where application software and some drivers execute.

User mode	**User applications**	For example, bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, etc.			
	Low-level system components:	**System daemons**: *systemd*, *runit*, *logind*, *networkd*, *PulseAudio*, ...	**Windowing system**: *X11*, *Wayland*, *SurfaceFlinger* (Android)	**Other libraries:** *GTK+*, *Qt*, *EFL*, *SDL*, *SFML*, *FLTK*, *GNUstep*, etc.	**Graphics**: *Mesa*, *AMD Catalyst*, ...
	C standard library	`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, ... (up to 2000 subroutines) *glibc* aims to be POSIX/SUS-compatible, *musl* and *uClibc* target embedded systems, *bionic* written for Android, etc.			
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)			
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*			
Hardware (CPU, main memory, data storage devices, etc.)					

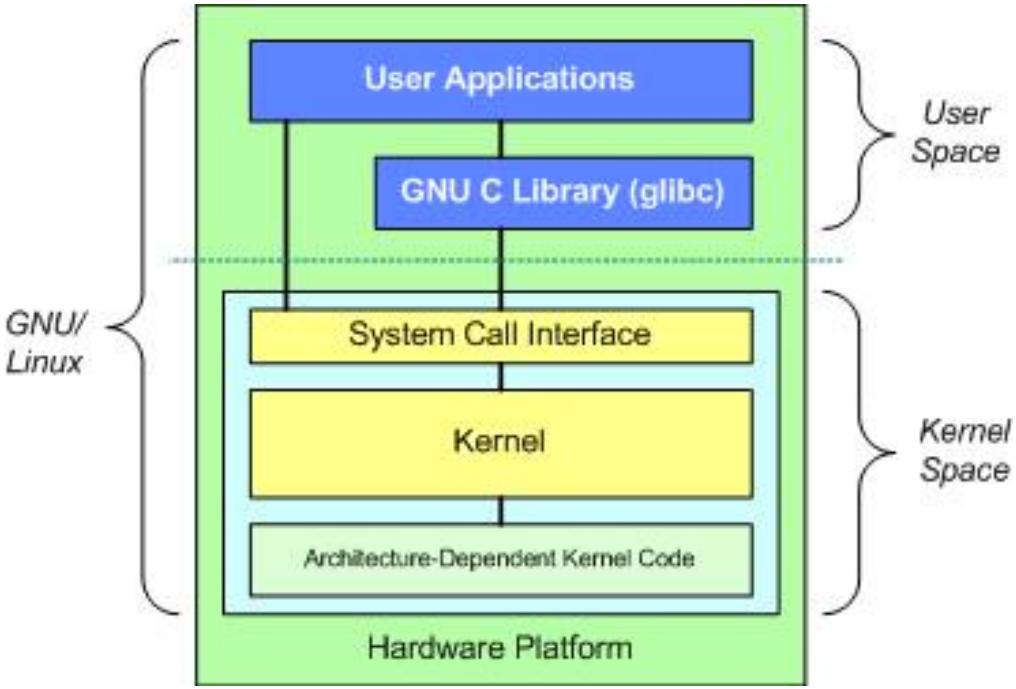
User mode vs. Kernel mode

- The difference between kernel and user mode?
  - The CPU can execute any instruction in its instruction set and use any feature of the hardware when executing in kernel mode.
  - However, it can execute only a subset of instructions and use only subset of features when executing in the user mode.

- The purpose of having these two modes
  - Purpose: protection – to protect critical resources (e.g., privileged instructions, memory, I/O devices) from being misused by user programs.

Interactions between user and kernel spaces

- For applications, in order to perform privileged operations, it must transit into OS through well defined interfaces
  - System call

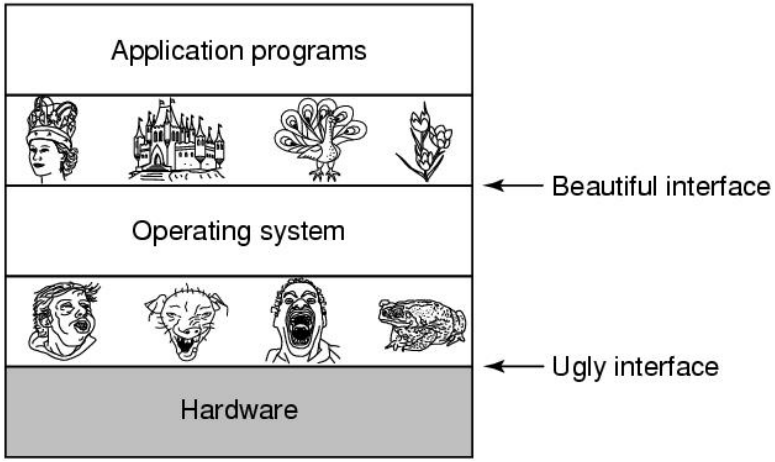



```
printf("%d", helloworld);
        ↓
write(buffer, count) ;
        ↓
os->write(buf, count, pos);
        ↓
```

System calls

- A type of special "protected procedure calls" allowing user-level processes request services from the kernel.

- System calls provide:
  - An abstraction layer between processes and hardware, allowing the kernel to provide access control
  - A virtualization of the underlying system
  - A well-defined interface for system services



System calls vs. Library functions

- What are the similarities and differences between ***system calls*** and ***library functions*** (e.g., libc functions)?

libc functions

https://www.gnu.org/software/libc/manual/html_node/Function-Index.html

system calls

https://filippo.io/linux-syscall-table/

System calls vs. Library functions

- Similarity
  - Both appear to be APIs that can be called by programs to obtain a service
    - E.g., open,
    - https://elixir.bootlin.com/linux/latest/source/tools/include/nolibc/nolibc.h#L2038
    - E.g., strlen
    - https://www.gnu.org/software/libc/manual/html_node/String-Length.html#index-strlen

System calls vs. Library functions

```c
/* strlen example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char szInput[256];
  printf ("Enter a sentence: ");
  gets (szInput);
  printf ("The sentence entered is %u characters long.\n",(unsigned)strlen(szInput));
  return 0;
}
```

Output:

```
Enter sentence: just testing
The sentence entered is 12 characters long.
```

libc functions:

<string.h> - - -> strlen() : all in user space

System calls vs. Library functions

```c
// C program to illustrate
// open system call
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
    // if file does not have in directory
    // then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);

    printf("fd = %d/n", fd);

    if (fd ==-1)
    {
        // print which type of error have in a code
        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"
        perror("Program");
    }
    return 0;
}
```

https://elixir.bootlin.com/linux/latest/source/fs/open.c#L1074

```c
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
        struct open_flags op;
        int fd = build_open_flags(flags, mode, &op);
        struct filename *tmp;

        if (fd)
                return fd;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;

        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

System calls:

<fcntl.h> - - -> open()

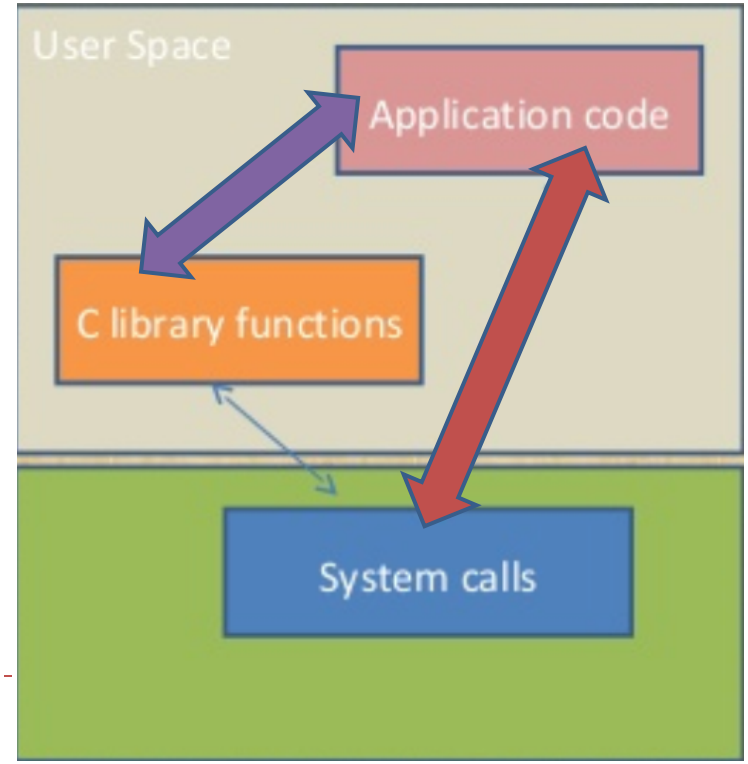
- - -> do_sys_open() // wrapper system call

System calls vs. Library functions

- Difference
  - Library functions execute in the user space
  - System calls execute in the kernel space

strlen() (<string.h>) ? → all in user space

open() (<fcntl.h>)? → do_sys_open()

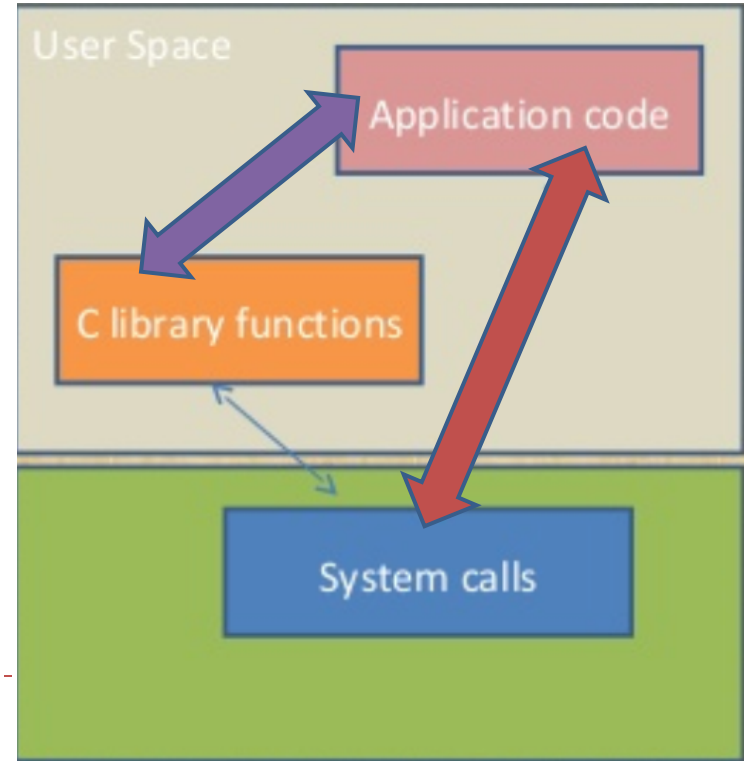


System calls vs. Library functions

- Difference
  - Fast, no context switch
  - Slow, high cost, kernel/user context switch

strlen() (<string.h>) ? → all in user space

open() (<fcntl.h>)? → do_sys_open()



Services Provided by System Calls

- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection, e.g., encrypt
- Networking, etc.

Examples

```
#include<stdio.h>
#include<dos.h>
int main()
{
   struct date dt;

   getdate(&dt);

   printf("Operating system's current date is %d-%d-%d\n"
   ,dt.da_day,dt.da_mon,dt.da_year);

   return 0;
}
```

OUTPUT:

Operating system's current date is 12-01-2012

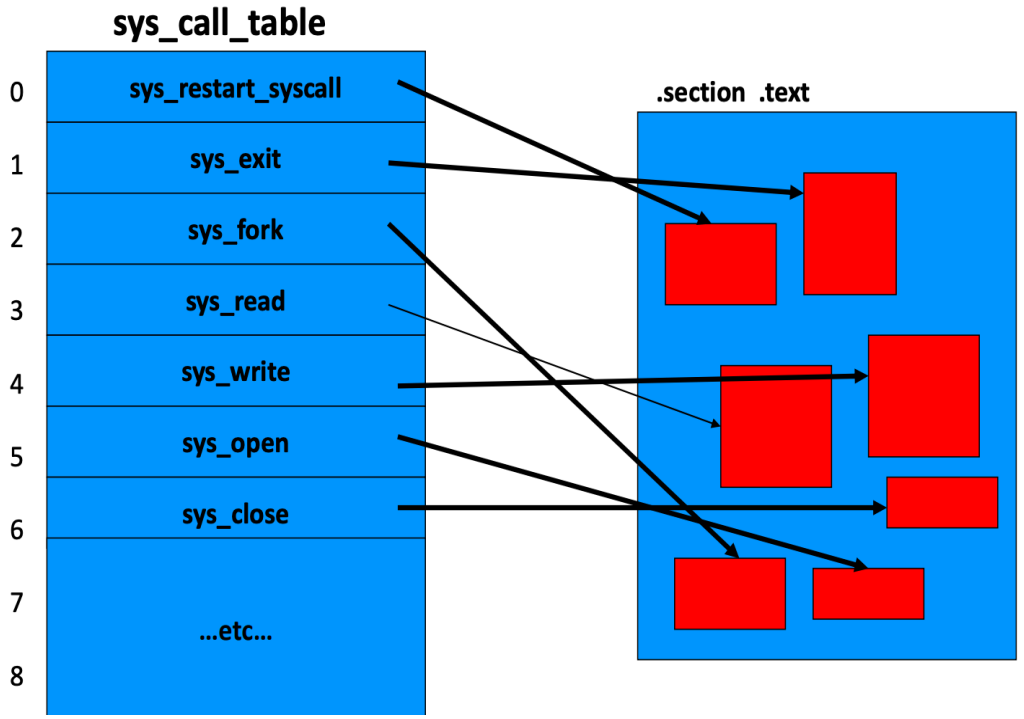
```
#include <stdio.h>
#include <time.h>
int main ()
{
    time_t seconds;
    seconds = time (NULL);

    printf ("Number of hours since 1970 Jan 1st " \
            "is %ld \n", seconds/3600);
    return 0;
}
```

OUTPUT:

Number of hours since 1970 Jan 1st is 374528

System call table

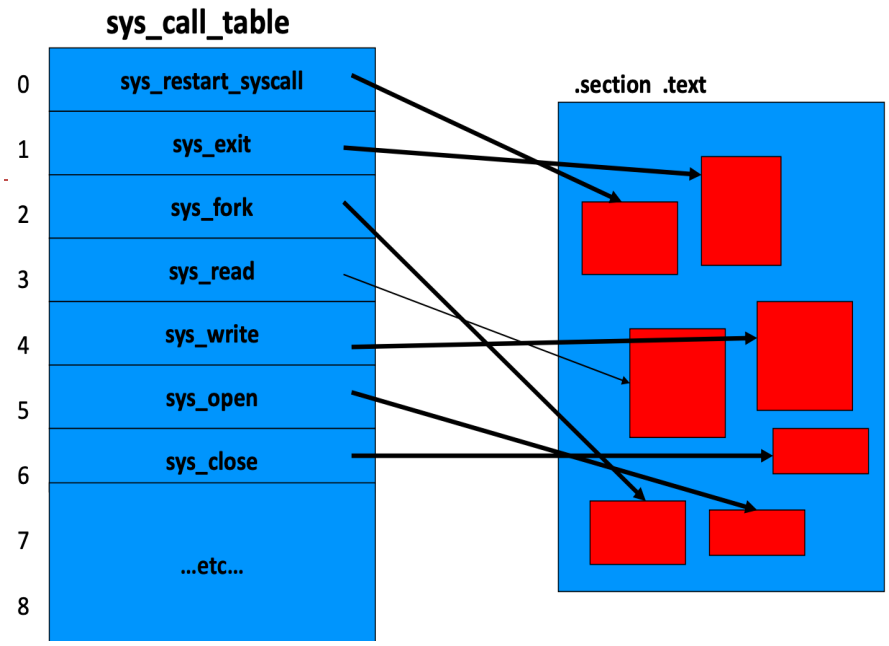
- There are approximately 350 system calls in Linux.
- An array of function-pointers (identified by the ID number)
- This array is named 'sys_call_table[]' in Linux https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscall_64.c

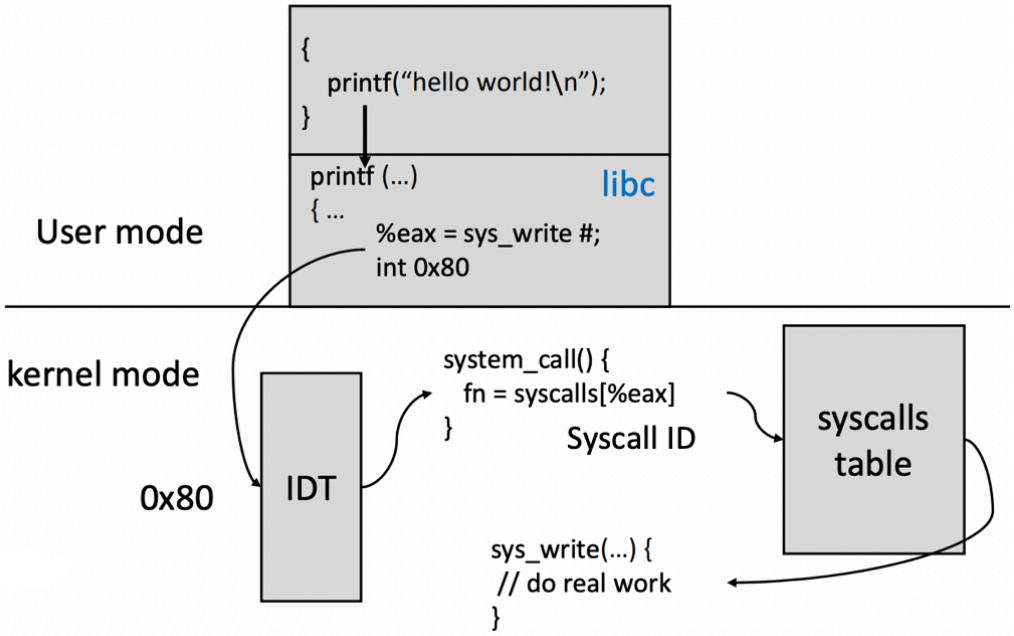


The 'jump-table' idea

System call table

- Any specific system-call is selected by its ID-number (i.e., the system call number, which is placed into register %eax)





Syscall Naming Convention

- Usually a library function “foo()” will do some work and then call a system call (“sys_foo()”)
- In Linux, all system calls begin with “sys_”
- Often “sys_abc()” just does some simple error checking and then calls a worker function named “do_abc()”

https://elixir.bootlin.com/linux/v4.14/source/arch/x86/entry/syscalls/syscall_64.tbl

open:

https://elixir.bootlin.com/linux/v4.14/source/fs/open.c#L1072

do_sys_open:

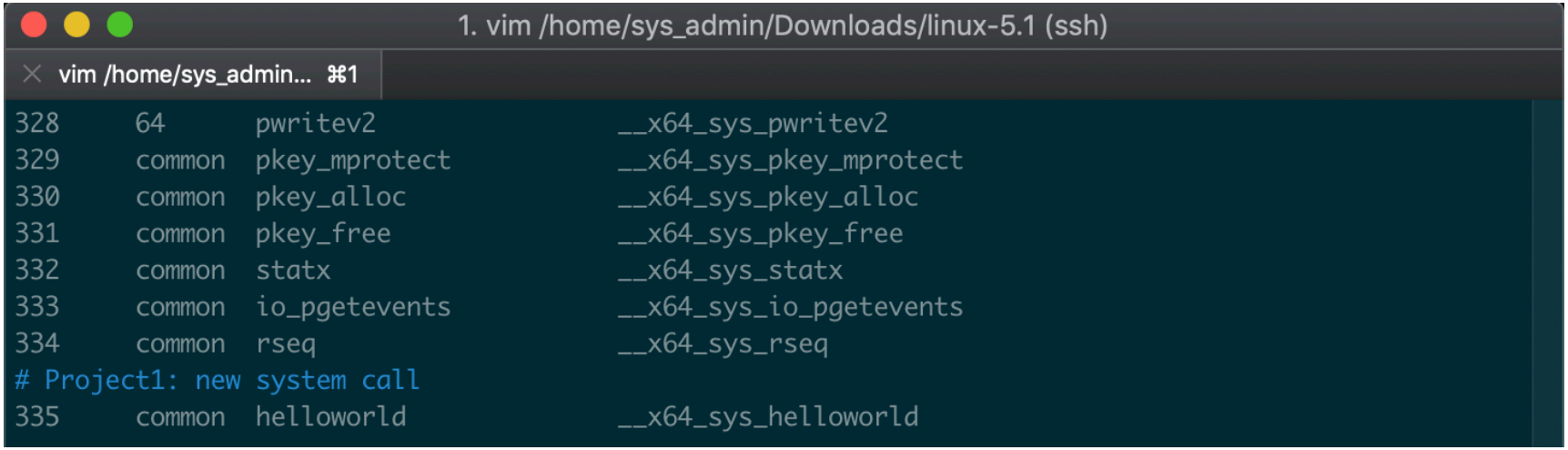
https://elixir.bootlin.com/linux/v4.14/source/fs/open.c#L1044

Define your system call

- Step 1: register your system call
- Step 2: declare your system call in the header file
- Step 3: implement your system call
- Step 4: write user level app to call it

Step 1: register your system call

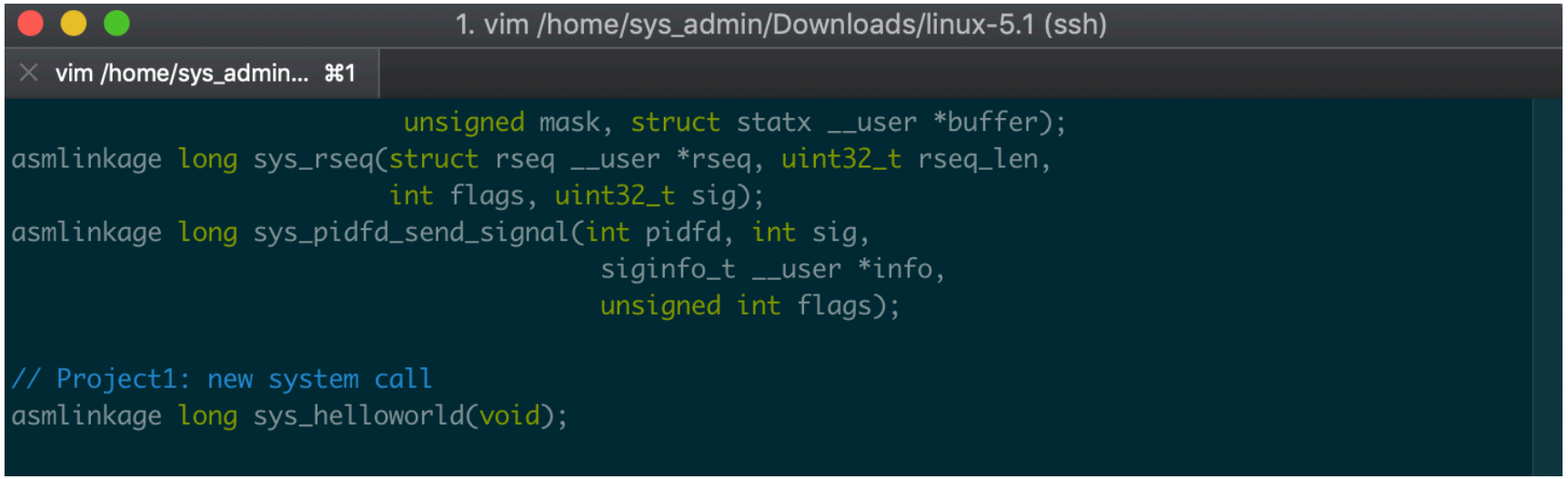
arch/x86/entry/syscalls/syscall_64.tbl



https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscalls/syscall_64.tbl#L346

Step 2: declare your system call in the header file

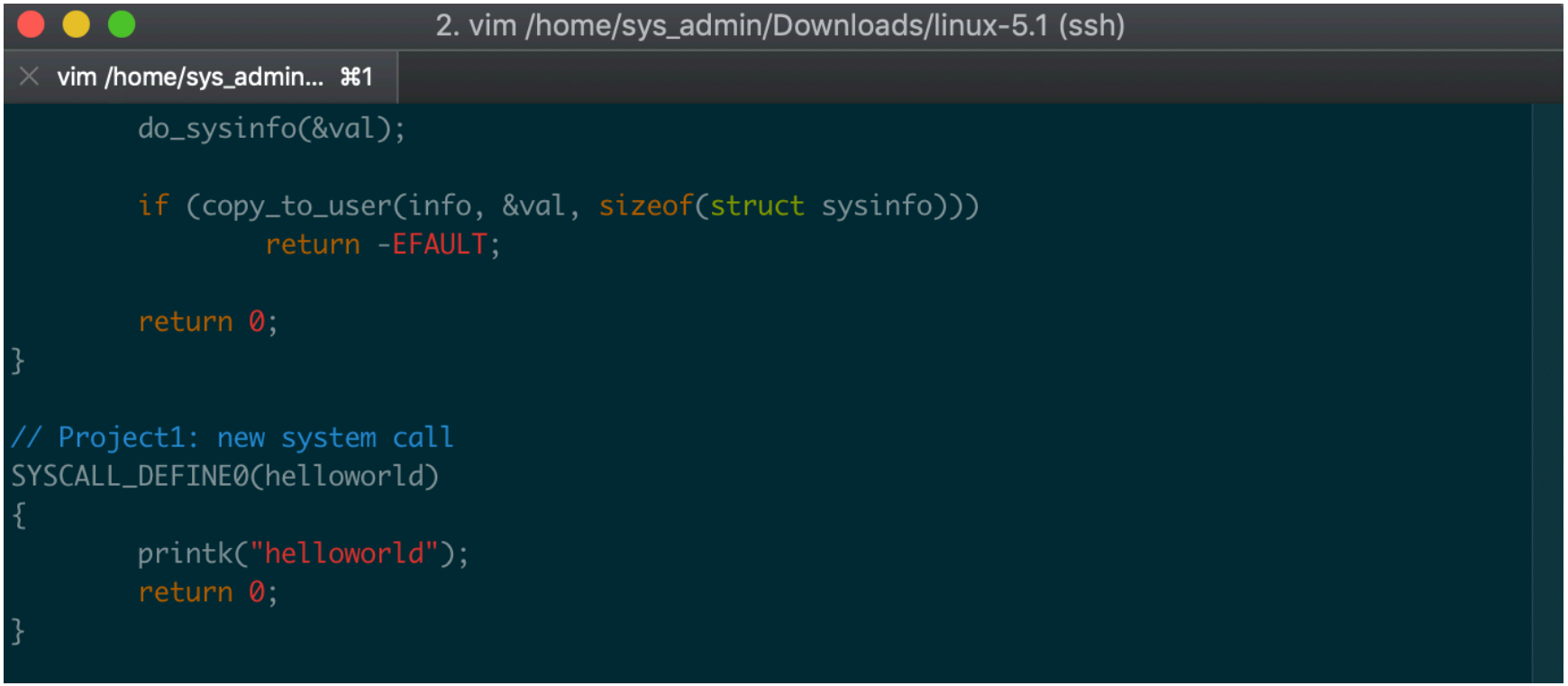
include/linux/syscalls.h



https://elixir.bootlin.com/linux/v5.0/source/include/linux/syscalls.h

Step 3: implement your system call

kernel/sys.c



https://elixir.bootlin.com/linux/v5.0/source/kernel/sys.c#L402

Step 4: write user level app to call it

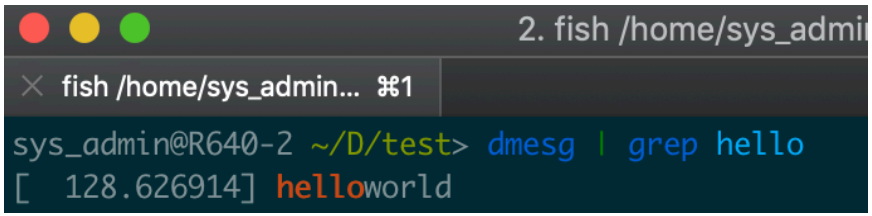
test_syscall.c：

```
****************************************
#include <linux/unistd .h>
#include <sys/syscall .h>
#include <sys/types .h>
#include <stdio .h>
#define __NR_helloworld 335

int main(int argc, char *argv[])
{
    syscall (__NR_helloworld) ;
    return 0 ;
}
****************************************

//If syscall needs parameter, then：
//syscall (__NR_helloworld，a，b，c) ;
```

Compile and execute：

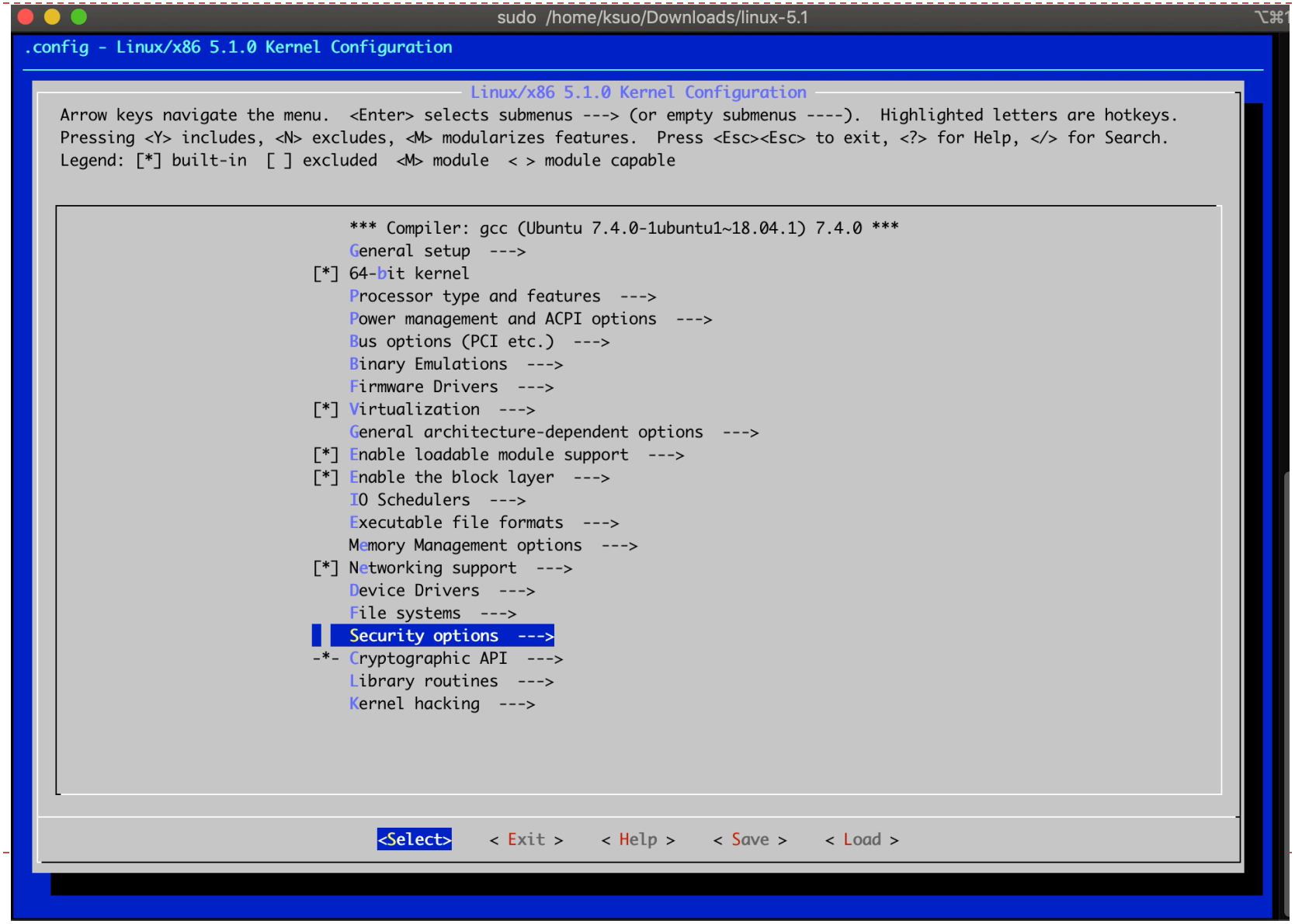
```
$ gcc test_syscall.c -o test_syscall
$ ./test_syscall
```

The test program will call the new system call and output a helloworld message at the tail of the output of dmesg (system log).



Project 1: menuconfig

https://youtu.be/UyOGF4UOoR0



Compile the kernel

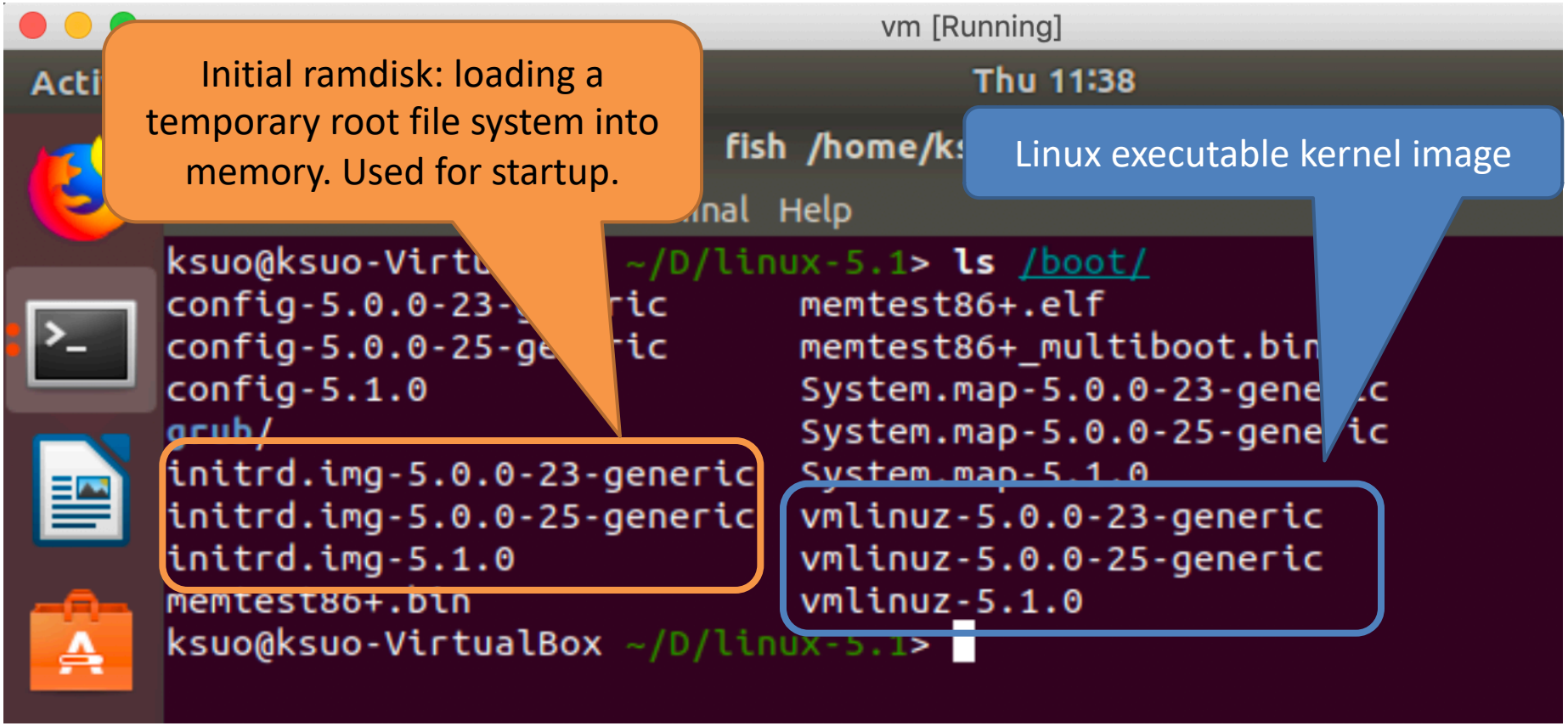
Commands:

$ sudo make; sudo make modules; sudo make modules_install; sudo make install

```
ksuo@ksuo-VirtualBox ~/D/linux-5.1>
sudo make; sudo make modules; sudo make modules_install; sudo make install
```

Where is the new kernel?

- $ ls /boot/



How to boot to the new kernel ?

If you are using Ubuntu: change the grub configuration file:

$ sudo vim /etc/default/grub

Make the following changes:

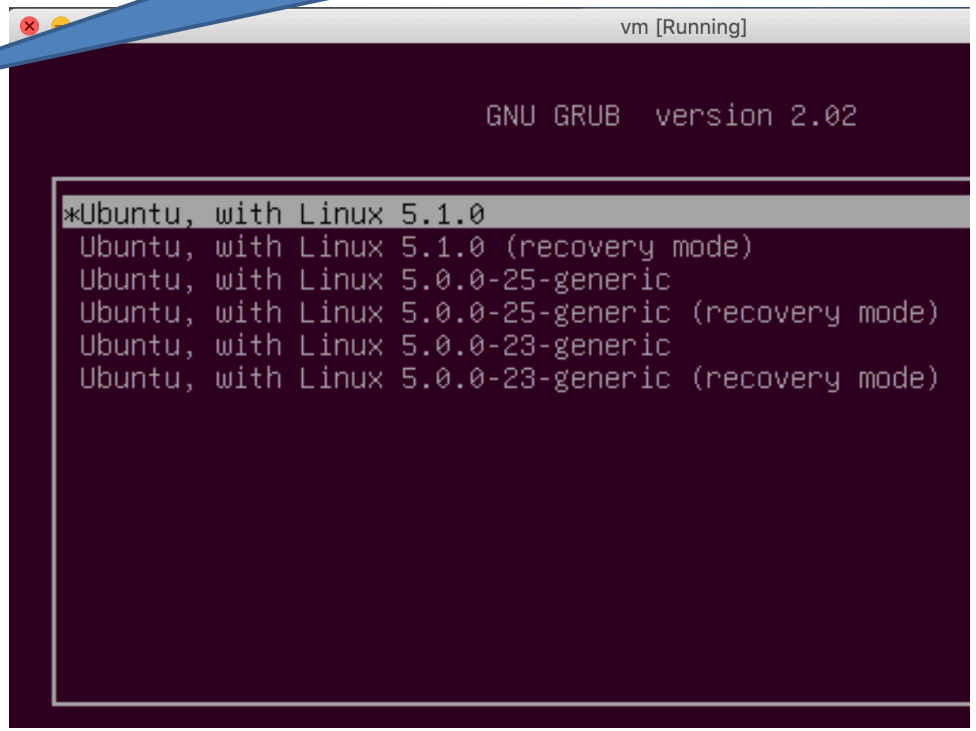
GRUB_DEFAULT=0

GRUB_TIMEOUT=10

Then, update the grub entry:

$ sudo update-grub2

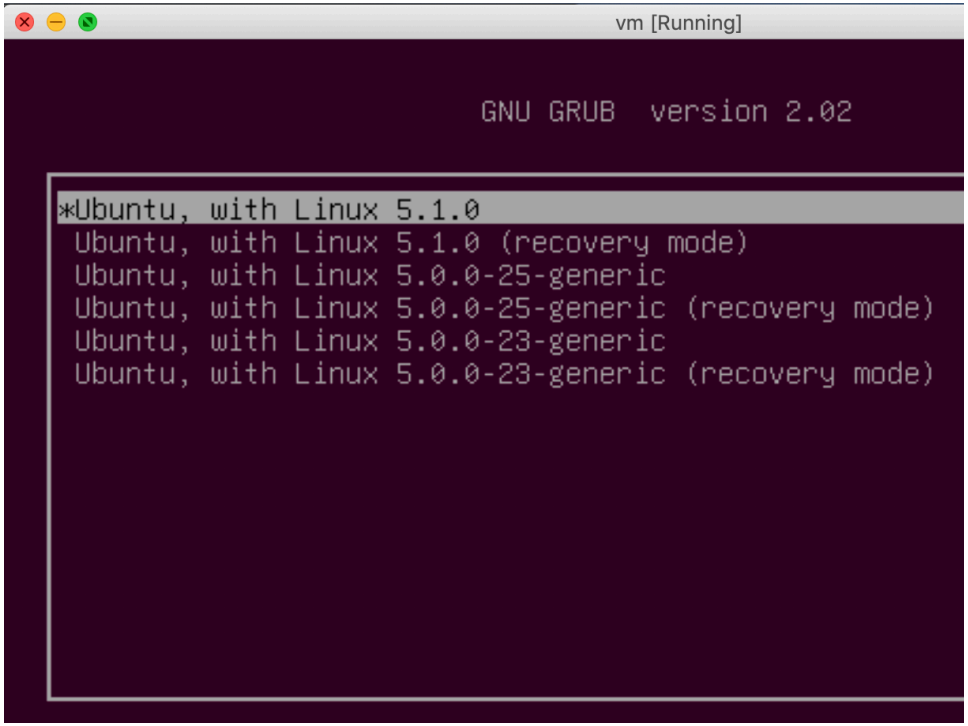
The OS boots by using the first kernel by default. You have 10 seconds to choose.



What if my kernel crashed?

- Your kernel could crash because you might bring in some kernel bugs

- In the menu, choose the old kernel to boot the system
- Fix your bug in the source code
- Compile and reboot



Edit a file with vim

- step 1: $ vim file
- step 2: press i, enter insert mode; move the cursor to position and edit the context
- step 3: after editing, press ESC to exit the insert mode to normal mode
- step 4: press :wq to save what you edited and quit. If you do not want to save, press :q!

More about vim

- A quick start guide for beginners to the Vim text editor
  - https://eastmanreference.com/a-quick-start-guide-for-beginners-to-the-vim-text-editor
- Vim basics:
  - https://www.howtoforge.com/vim-basics
- Learn the Basic Vim Commands [Beginners Guide]
  - https://www.youtube.com/watch?time_continue=265&v=ZEGqkam-3Ic