

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



# User mode vs. Kernel mode

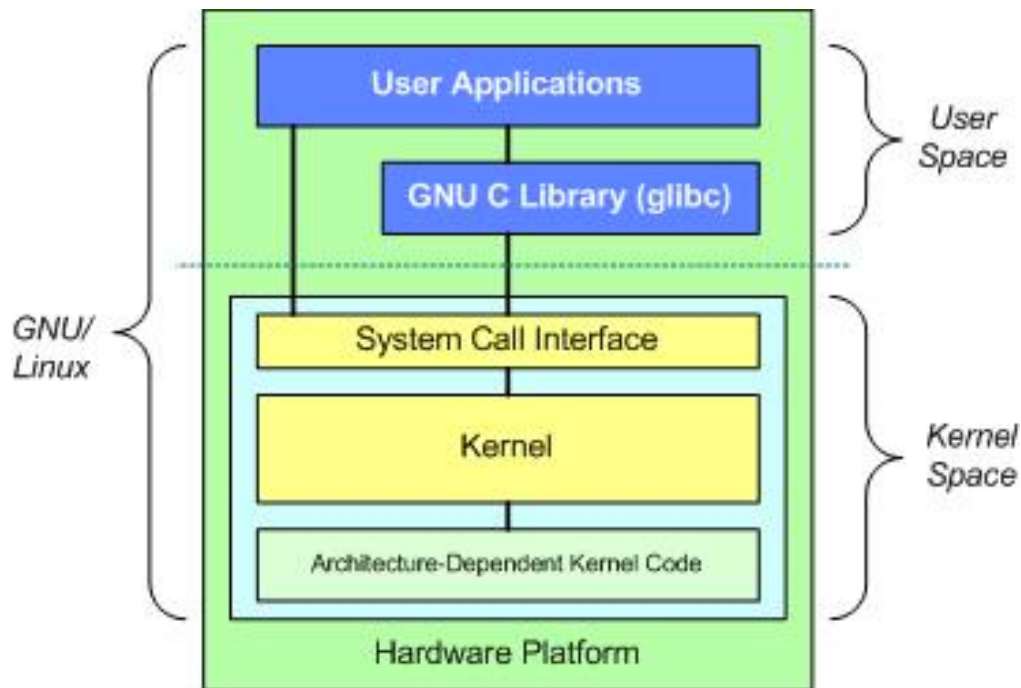
---

- 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) ;
```

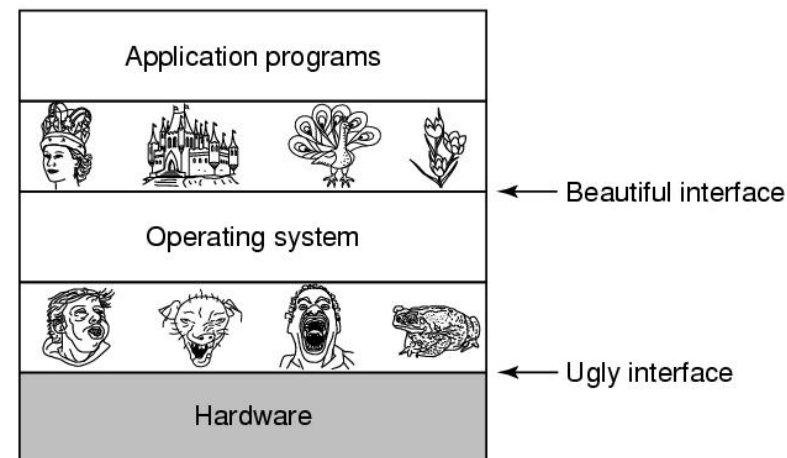
```
os->write(buf, count, pos);
```



# 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](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](https://www.gnu.org/software/libc/manual/html_node/String-Length.html#index-strlen)





# System calls vs. Library functions

```
1 /* strlen example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char szInput[256];
8     printf ("Enter a sentence: ");
9     gets (szInput);
10    printf ("The sentence entered is %u characters long.\n", (unsigned)strlen(szInput));
11    return 0;
12 }
```

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 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;
}
```

System calls:

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

- - -> do\_sys\_open() // wrapper system call

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

```
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 vs. Library functions

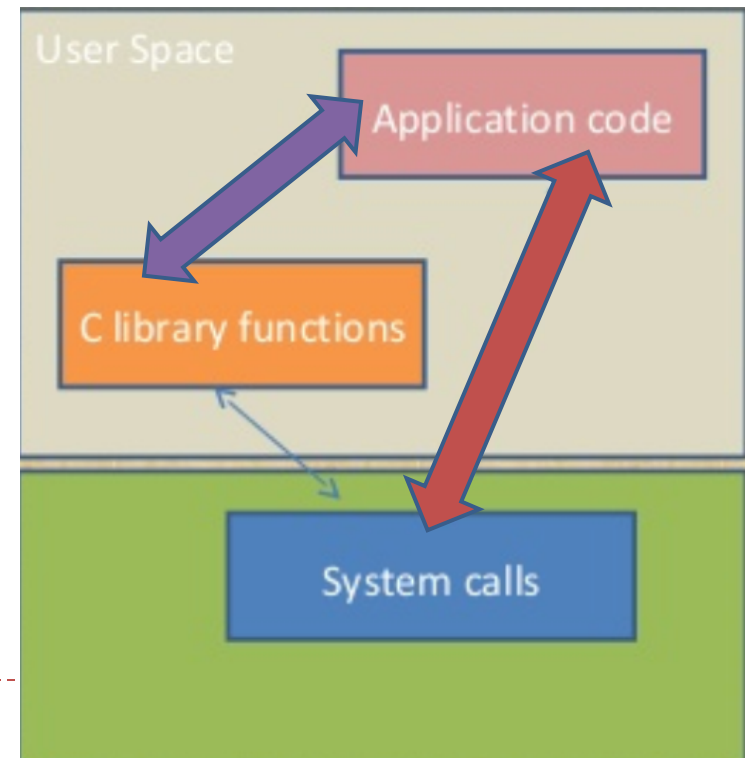
---

- 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

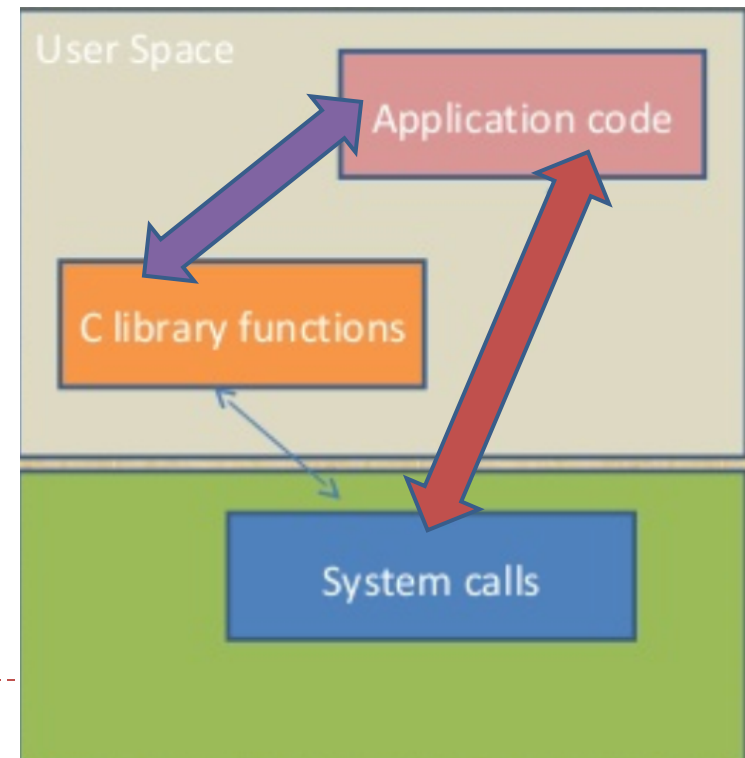
---

- 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;
}
```

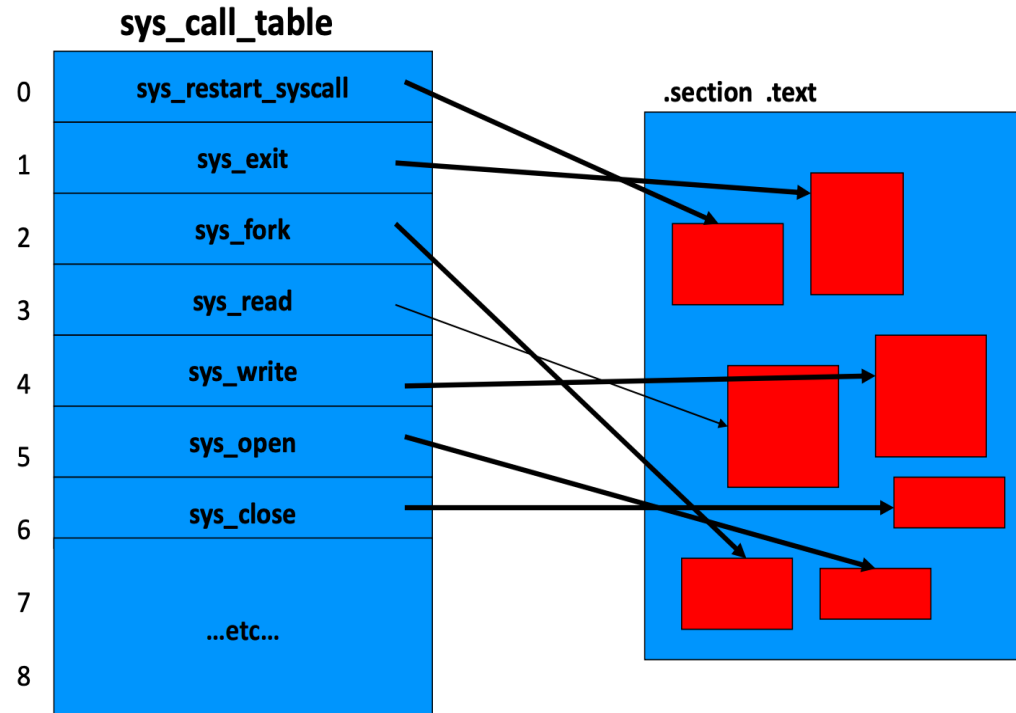
OUTPUT:

Number of hours since 1970 Jan 1st is 374528



# System call table

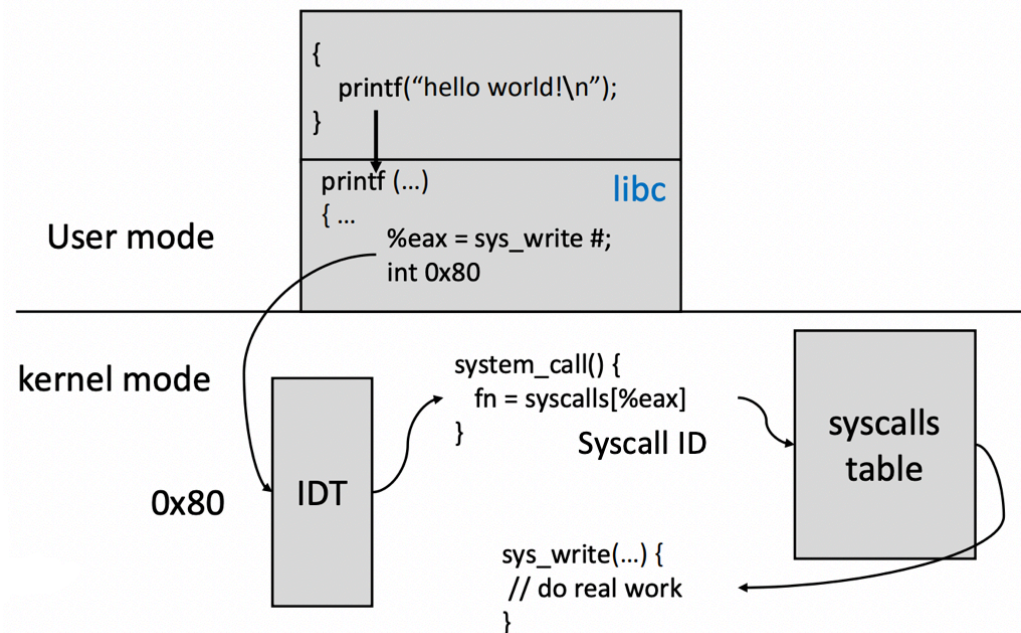
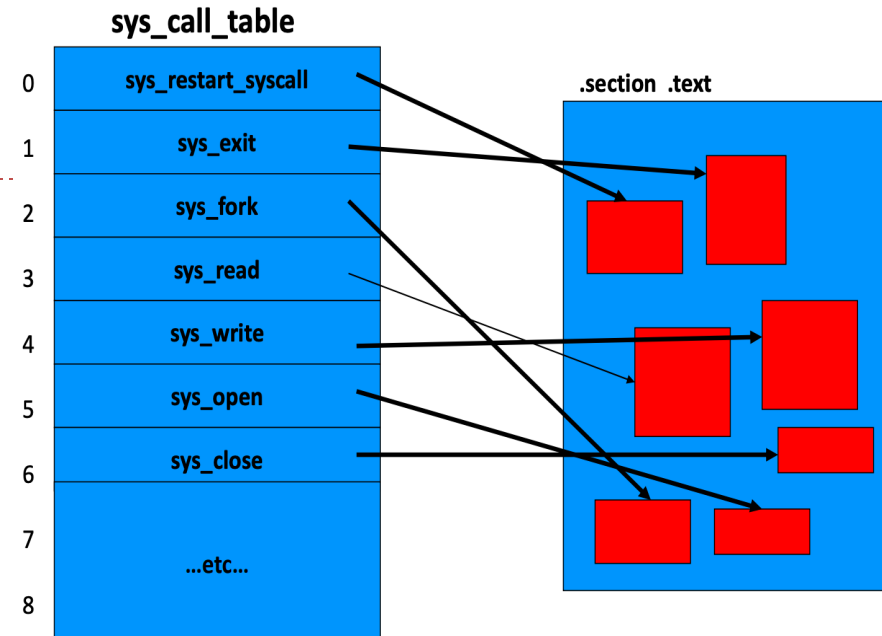
- There are approximately 350 system calls in Linux.
- An array of function-pointers (identified by the ID number)
- This array is named 'sys\_call\_table[]' in Linux  
[https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscall\\_64.c](https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscall_64.c)



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](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

---

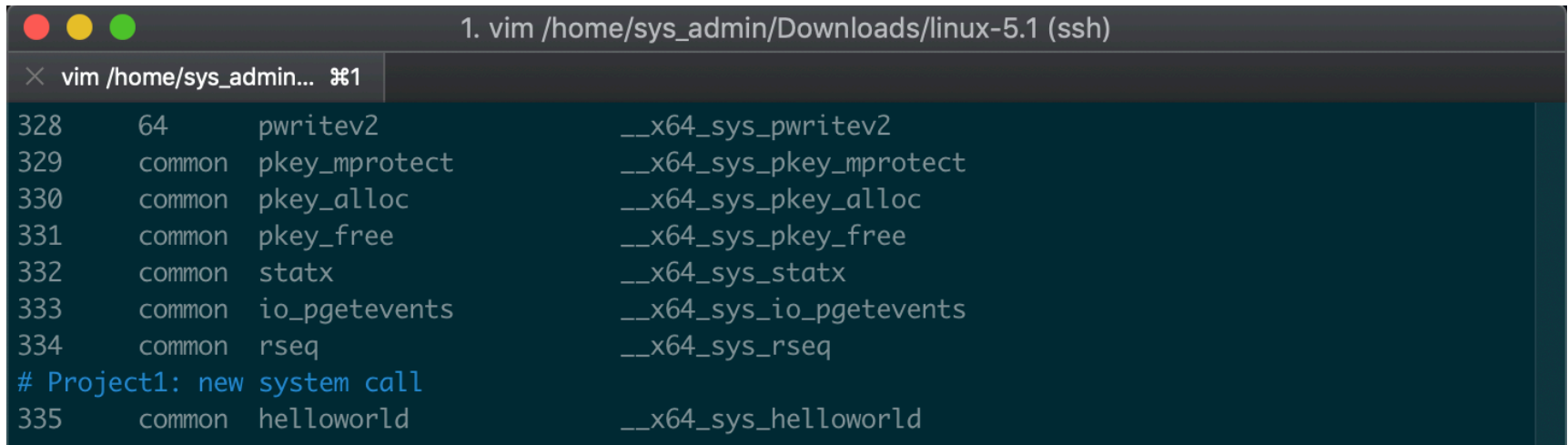
- Step 1: register your system call
- Step 2: declare your system call in the header file
- Step 3: implement your system call
- Step 4: write user level app to call it



# Step 1: register your system call

---

arch/x86/entry/syscalls/syscall\_64.tbl



```
1. vim /home/sys_admin/Downloads/linux-5.1 (ssh)
vim /home/sys_admin... 381
328    64      pwritev2      __x64_sys_pwritev2
329    common  pkey_mprotect __x64_sys_pkey_mprotect
330    common  pkey_alloc    __x64_sys_pkey_alloc
331    common  pkey_free     __x64_sys_pkey_free
332    common  statx         __x64_sys_statx
333    common  io_pgetevents __x64_sys_io_pgetevents
334    common  rseq          __x64_sys_rseq
# Project1: new system call
335    common  helloworld    __x64_sys_helloworld
```

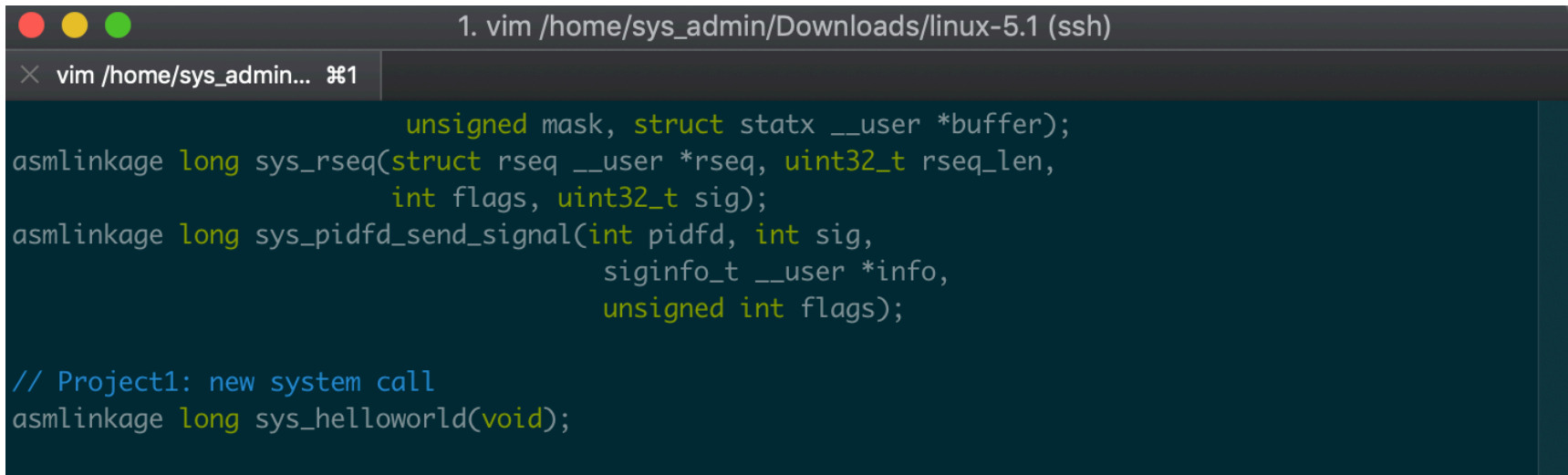
[https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscalls/syscall\\_64.tbl#L346](https://elixir.bootlin.com/linux/v5.0/source/arch/x86/entry/syscalls/syscall_64.tbl#L346)



## Step 2: declare your system call in the header file

---

include/linux/syscalls.h



The screenshot shows a vim editor window with the title bar "1. vim /home/sys\_admin/Downloads/linux-5.1 (ssh)". The editor buffer shows the following code:

```
        unsigned mask, struct statx __user *buffer);
asmlinkage long sys_rseq(struct rseq __user *rseq, uint32_t rseq_len,
                        int flags, uint32_t sig);
asmlinkage long sys_pidfd_send_signal(int pidfd, int sig,
                                     siginfo_t __user *info,
                                     unsigned int flags);

// Project1: new system call
asmlinkage long sys_helloworld(void);
```

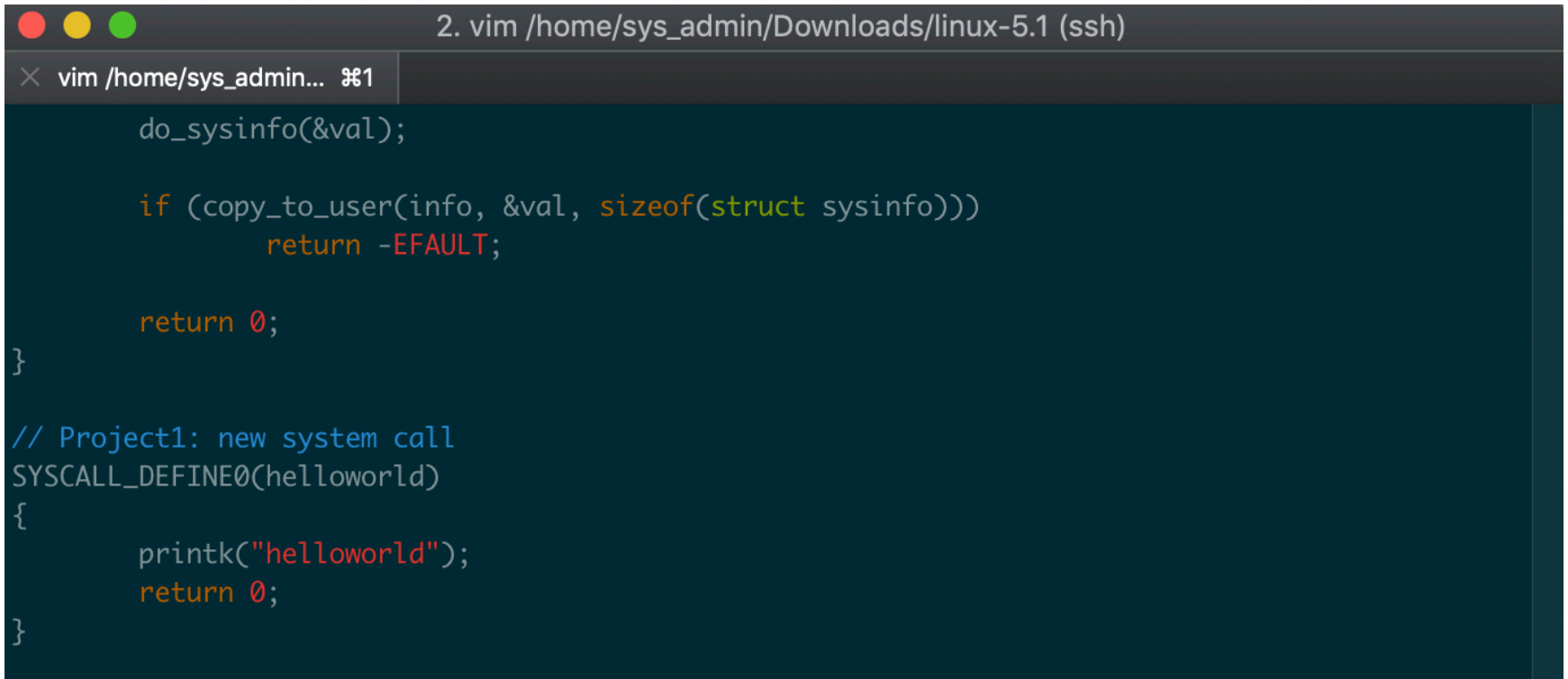
<https://elixir.bootlin.com/linux/v5.0/source/include/linux/syscalls.h>



# Step 3: implement your system call

---

kernel/sys.c



```
2. vim /home/sys_admin/Downloads/linux-5.1 (ssh)
vim /home/sys_admin... 1
do_sysinfo(&val);

if (copy_to_user(info, &val, sizeof(struct sysinfo)))
    return -EFAULT;

return 0;
}

// Project1: new system call
SYSCALL_DEFINE0(helloworld)
{
    printk("helloworld");
    return 0;
}
```

<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:
```

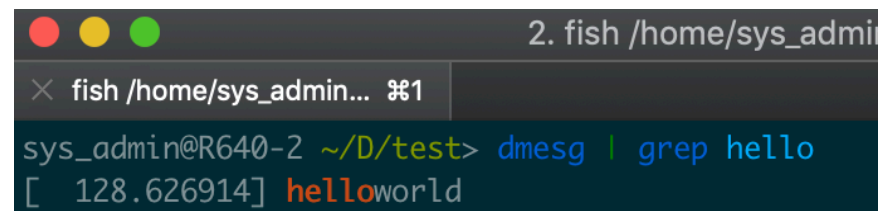
```
//syscall (__NR_helloworld, a, b, c);
```

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).

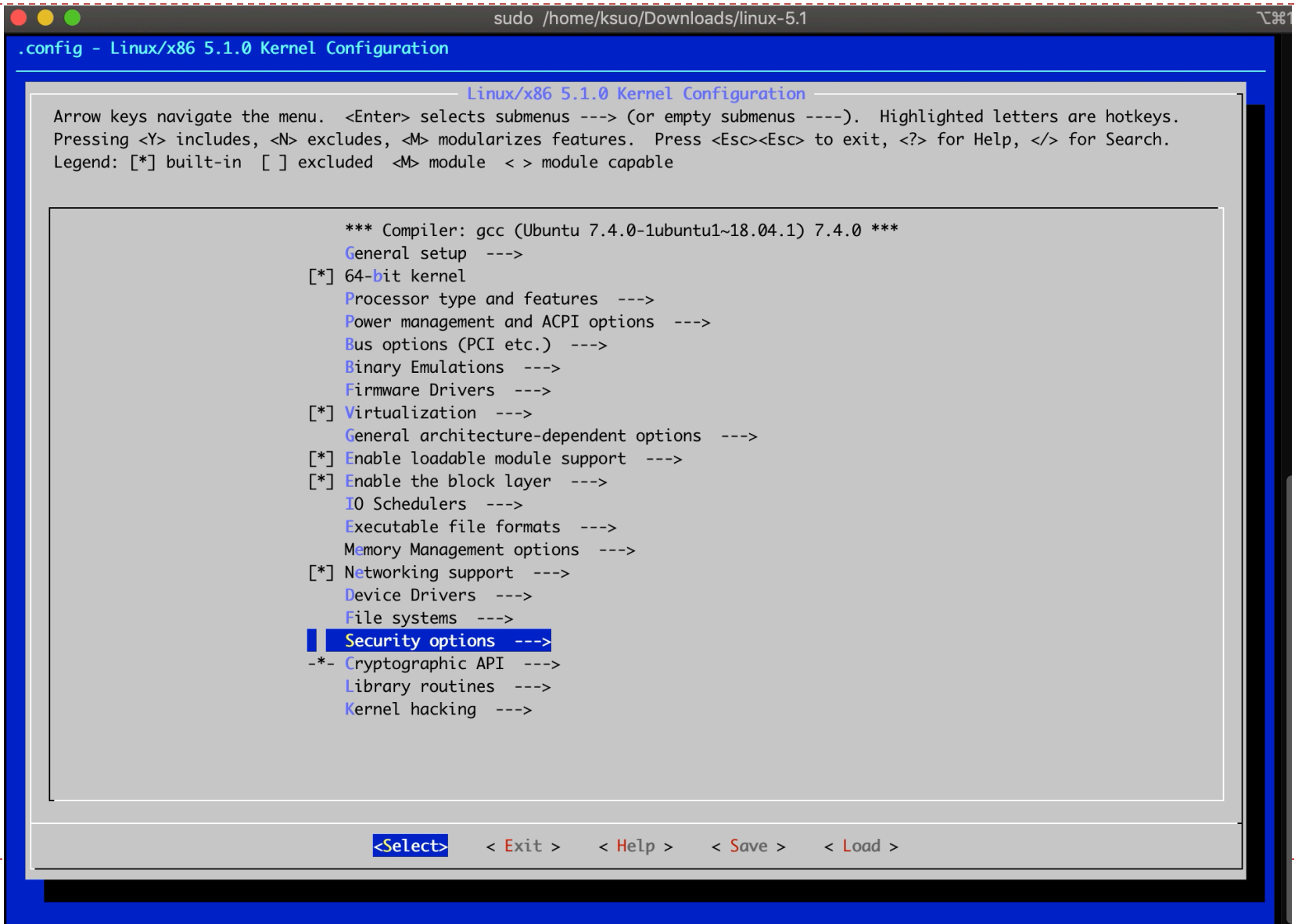


```
2. fish /home/sys_admin...
fish /home/sys_admin... %1
sys_admin@R640-2 ~/D/test> dmesg | grep hello
[ 128.626914] helloworld
```



# Project 1: menuconfig

<https://youtu.be/UyOGF4UOoR0>



```
sudo /home/ksuo/Downloads/linux-5.1
.config - Linux/x86 5.1.0 Kernel Configuration

Linux/x86 5.1.0 Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys.
Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search.
Legend: [*] built-in [ ] excluded <M> module < > module capable

*** Compiler: gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0 ***
General setup --->
[*] 64-bit kernel
Processor type and features --->
Power management and ACPI options --->
Bus options (PCI etc.) --->
Binary Emulations --->
Firmware Drivers --->
[*] Virtualization --->
General architecture-dependent options --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
IO Schedulers --->
Executable file formats --->
Memory Management options --->
[*] Networking support --->
Device Drivers --->
File systems --->
Security options --->
-- Cryptographic API --->
Library routines --->
Kernel hacking --->

<Select> <Exit> <Help> <Save> <Load>
```

# 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/`

Initial ramdisk: loading a temporary root file system into memory. Used for startup.

Linux executable kernel image

```
config-5.0.0-23-generic
config-5.0.0-25-generic
config-5.1.0
grub/
initrd.img-5.0.0-23-generic
initrd.img-5.0.0-25-generic
initrd.img-5.1.0
memtest86+.bin
```

```
vmlinuz-5.0.0-23-generic
vmlinuz-5.0.0-25-generic
vmlinuz-5.1.0
```

# How to boot to the new kernel ?

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

```
$ sudo vim /etc/default/grub
```

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

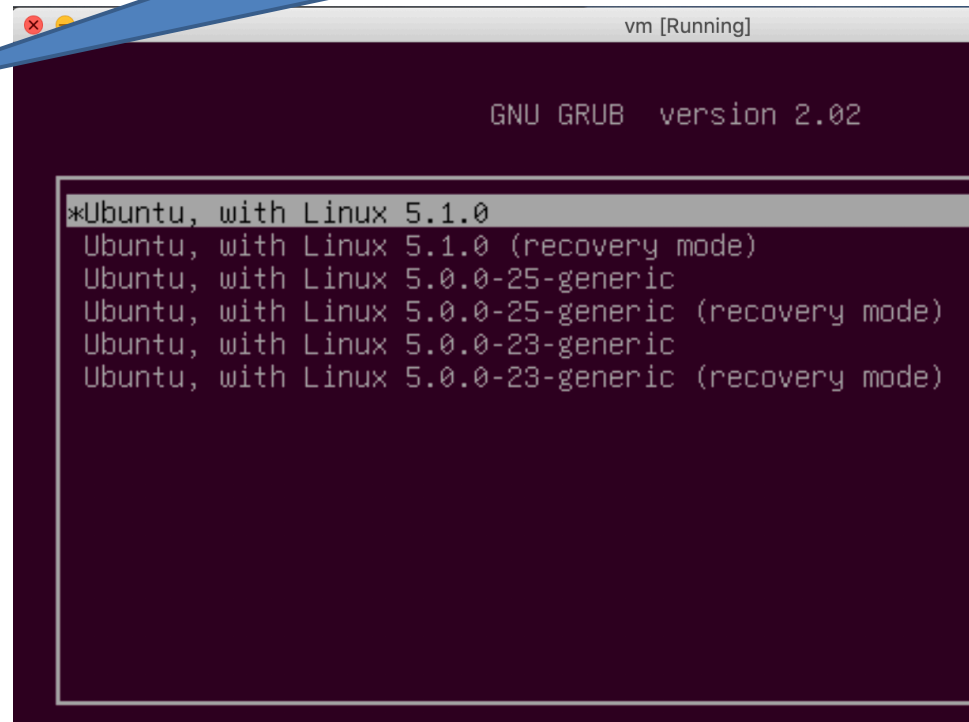
Make the following changes:

```
GRUB_DEFAULT=0
```

```
GRUB_TIMEOUT=10
```

Then, update the grub entry:

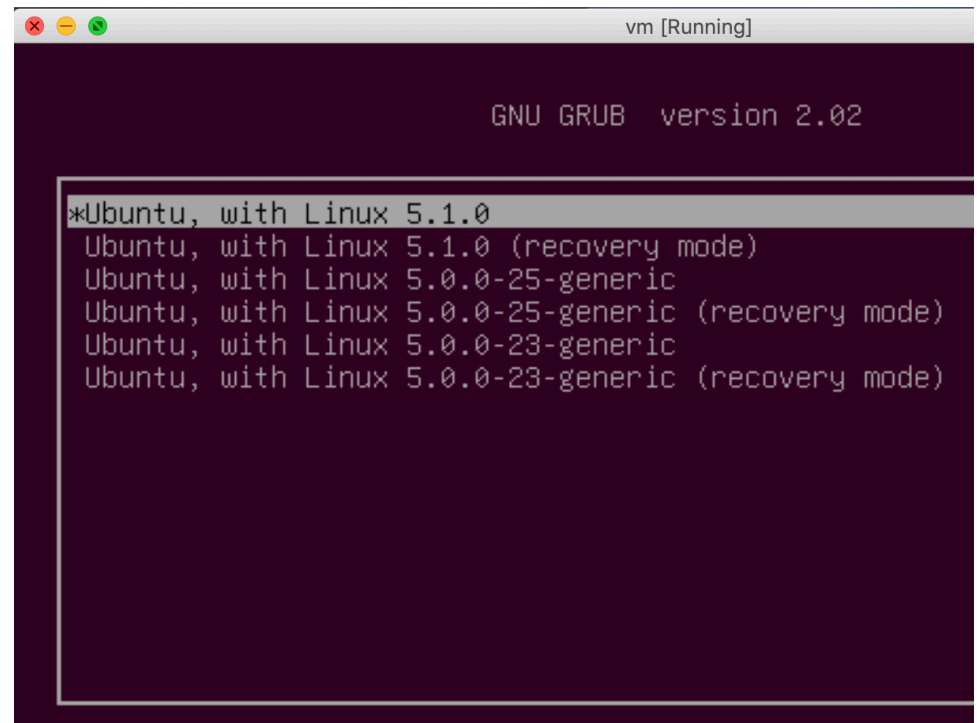
```
$ sudo update-grub2
```



# 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-3lc](https://www.youtube.com/watch?time_continue=265&v=ZEGqkam-3lc)

