Disclaimer: This material is a study notes, so for any technical issues please contact me to resolve it periodically.

Linux Kernel Overview

Contents

1 What is the Linux kernel ?

Q: Where to find the Linux Kernel ?
A: You can find it under \boot, you will find ”vmlinuz-xxx” (FYI: vmlinuz the z means compressed image, vmlinux is not compressed !)

1.1 Linux Important concepts

- each application has a specific use, and programs should not do more than what it's supposed to do. (one program = one function only), and you can reach the overall functionality by letting the applications cooperate together.

- Everything is a file ! You need to know that Linux deals with everything as a file to read/write from and into it. and you have seven types of a files as following:

  1. - : regular file
  2. d : directory
  3. c : character device file
  4. b : block device file
  5. s : local socket file
  6. p : named pipe
  7. l : symbolic link

  And you can get the type of a certain file by using the *ls -ld ¡file name¿*

Listing 1: HW info

```
$ touch file
$ ls −ld file
# Normal Device file
```

```
−rw−rw−r−− 1 root root 0 Jan 10 12:52 file

$ ls −ld /dev/null
# Character device file
crw−−−−−−− 1 root root 10, 165 Jan 4 10:13 /dev/null

$ ls −ld /dev/sda
# Block device file
brw−rw−−−− 1 root root 8, 0 Jan 4 10:12 /dev/sda

$ ls −ld /dev/log
# Local Socket Device file
srw−rw−rw− 1 root root 0 Jan 4 10:13 /dev/log
```

2 Understanding the *syscalls*

They are some functions implemented by the kernel for the user space to request operations from the kernel. there are around 400 Linux *syscalls*.
You can find the list of them in **"include/uapi/asm-generic/unistd.h"** in the Linux kernel source tree. in this file, you will find some constants which every constant is a system call id.

When you use a system call to ask the kernel to do something for you, the parameters will stored in a registers and the kernel is invoked (through a software interrupt), then it determines which call call is invoked, and call it. according to the system call return value you can continue your application, but if the system call returned -1, and you should handle that error as per your application. let's see an example:

Listing 2: HW info

```
$ man 2 read # read the manual section 2 for the read
# output of the command
READ(2) Linux Programmer's Manual READ(2)
NAME
    read − read from a file descriptor
SYNOPSIS
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
DESCRIPTION

 read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

 ...


RETRUN VALUE
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this
```

as you can see for the read *syscall*, it returns the number of bytes read or otherwise if there is an error, it returns -1.

2.1 Defining your own *syscall*

1. Add a constant for the syscall APIs in **"/arch/x86/include/asm/unistd_32.h"** to have a new entry for the syscall

Listing 3: Add the syscall constant "/arch/x86/include/asm/unistd_32.h"

```
#define __NR_newSysCall  338

and make sure to increase the (NR_syscalls = NR_syscalls + 1)

//before
#define NR_syscalls 338
//after
#define NR_syscalls 339
```

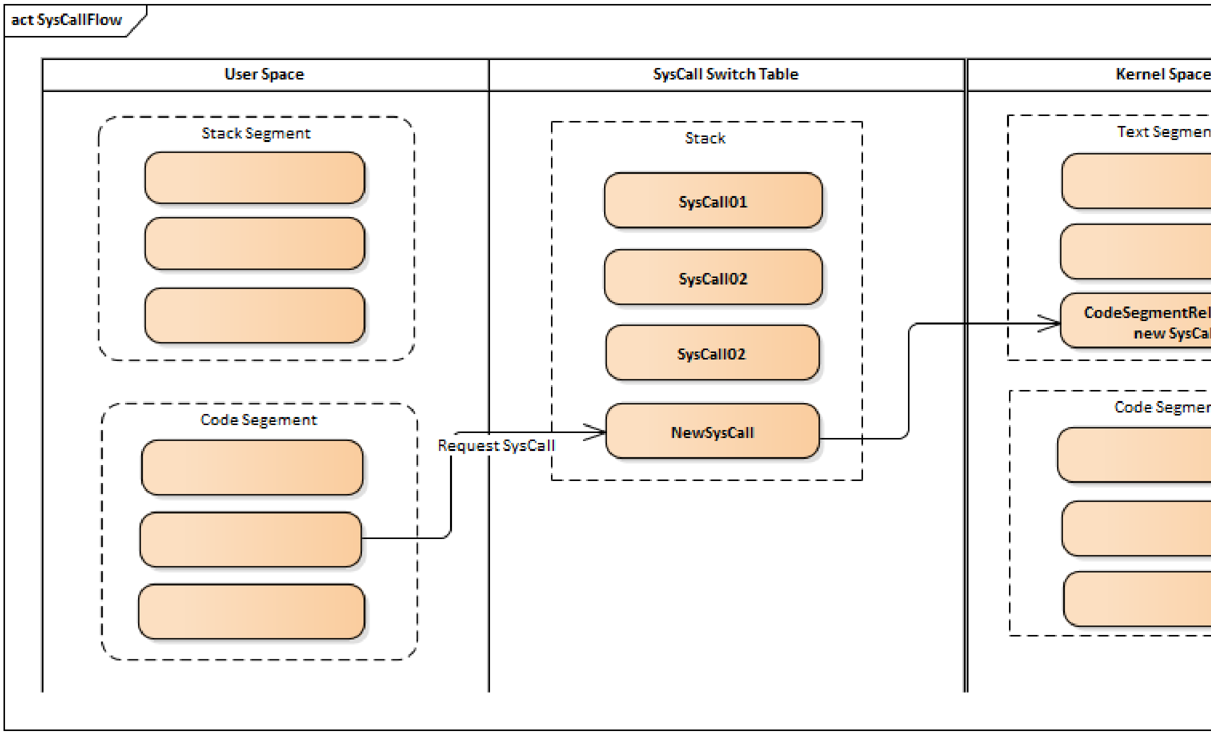
2. Define the syscall definition in "linux/arch/x86/include/asm/syscalls.h" (note: define it according to the target architecture x86_x64 or x86) Define the function with the standard naming conversion sys_ to identify it as sys call

   Listing 4: Define the syscall definiation "linux/arch/x86/include/asm/syscalls.h"

```
asmlinkage long sys_newSysCall(void);
```

3. Define the switch table for the call

4. Define the syscall in the syscall switch table "/arch/x86/kernel/syscall_table_32.S" The switch table which the addresses are called



5. Add the implementation in any file (no rules), you can create your own .C file for your own call Let's implement it in **”/kernel/sys.c”**

Listing 5: syscall implementation ”/kernel/sys.c”

```
asmlinkage long sys_newcall(void){
        printk(KERN_INFO ”\nnew call by process %s with pid %d\n”, current−>comm, current−>pid);
        return 0;
}
```

6. Build the kernel

2.2 Steps to call a system call from the user space

1. Copy system caller ID to eax (int) accumulator
2. Starting with right most argument and move each parameter onto each accumulator starting with ebx (max of 6 accumulator in intel)
3. Initiate a trap exception using processor specific instruction
4. read the exit value of the system call for eax.

The system call handling (as well the exception handling) is handled in **”/arch/i386/kernel/entry.S”** this file would be handled as follow

- Reads EAX
- lookup syscall switch table (the offset of the function)
- Allocate kernel stack (store the local data in the kernel stack)
- Update EIP register in PCB

These steps are converting the code from the user mode to kernel mode with more privileges

Listing 6: Syscall handling

```
#include <stdio.h>
main (){
        int a;
        __asm__(”movl $337, %eax”);
        __asm__(”int $0x80”);
        __asm__(”mov %eax, −4(%ebp)”); //copy the return value of the system call
}
```

Difference between a normal function call and a syscall

- Function call

Listing 7: Normal function call

```
main {
    int a = 100; //non−executable
    int b = 200;
    int c;
    int a+b; //executable
}

//Let's have a look at the assembly call
//1. Symbols
symbole_name type composition offset address
a   int 4  −12 (%ebp) //relative to the stack
b   int 4  −8 (%ebp)
c   int 4   −4 (%ebp)
                total 12
//2. Assembly
main:
    pushl %ebp  //pre−amble
    movl %esp, %ebp //pre−amble

    subl $12, %esp //the stack size = 12 as three variables
    movl $100, −12(%ebp) //initialize a by 100
    movl $100, −8(%ebp)
    movl −8(%ebp), %eax
    addl −12(%ebp), %eax
    movl %eax, −4(%ebp)

    leave //post−amble
    ret //post−amble
```

- Invoking a system call

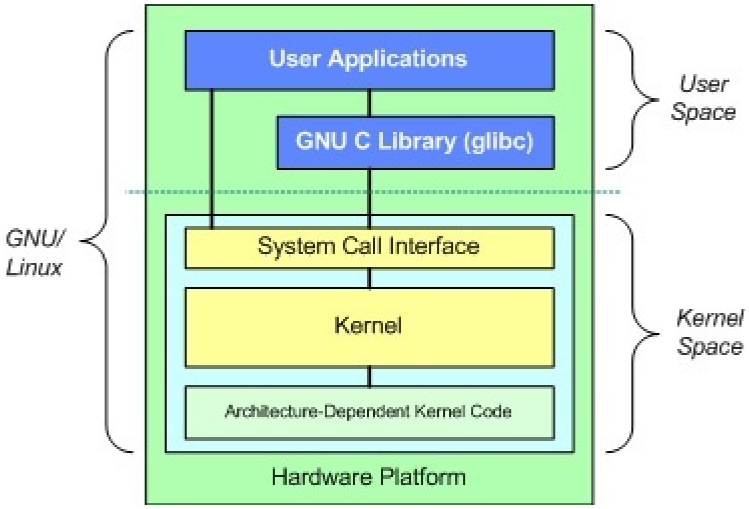
Listing 8: Invoking syscall

```
main(){
        int a;
        a = sys_newcalls();
}

main:
        pushl %ebp
        movl %esp, %ebx
        subl $4, %ebp

    // Copy system caller ID to eax (int) accumulator
        movl  $314, %eax
    // Trigger the context switch, and moving to the exception handler
        int $0x80

        movl −4(%ebp), %eax
```

You might read about the exception handling topic, here.

2.3 The system calls and the c Library

To make the application level code architecture independent, the use of the C Library is raised, as the C Library will be the translation unit(interface) between the kernel and the application, and for sure the C Library itself uses the system calls, and as an application developer you can use the C Library or directly use the Kernel *syscalls* to create applications.



And you should know that the *syscalls* are unique by the system.

2.4 Controlling and viewing information about the HW

As demonstrated above, You can use some applications to collect or even to control the HW devices.
These applications uses some C library functions, which used the kernel *syscalls* to control or display these information. Lets's see an example; *lshw, lspci, lsusb, lsblk, lscpu, lsdev*
for example if we used the *lspci* we will find that the terminal displays info about all the connect PCI devices

Listing 9: HW info

```
ubuntu@ip-172-31-46-156:~$ lspci
# bash output
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 01)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Unassigned class [ff80]: XenSource, Inc. Xen Platform Device (rev 01)
```

Note: that you can use lspci -v for the verbose mode (i.e. more information)

Even you can control the HW using the following command: *hdparm* or *setpci* commands

Listing 10: HW Control

You can use also other approach to control the HW units using the Linux *virtual file system* like echo to proc, dev, or sys file systems.

2.5 Reading the kernel messages!

printk is the kernel function to print messages, it's like C's printf() but for the kernel. the messages by *printk* is sent to a RAM buffer and sometimes to the system console. the kernel used a Logging Daemon to send the messages to a file or somewhere else. if you want to display these messages you can use the command *demesg*. it displays theRAM buffer to your terminal, and you need to note that the messages starts from very early of the boot process.

Listing 11: The Linux File System

```
ubuntu@ip-172-31-46-156:~$ dmesg
[ 0.000000] Linux version 4.15.0-1057-aws (buildd@lgw01-amd64-030) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu
[ 0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.15.0-1057-aws root=UUID=651cda91-e465-4685-
[ 0.000000] KERNEL supported cpus:
[ 0.000000] Intel GenuineIntel
[ 0.000000] AMD AuthenticAMD
[ 0.000000] Centaur CentaurHauls
[ 0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
[ 0.000000] x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
[ 0.000000] x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
```

You can also display some messages by accessing this file /var/log/messages or /var/log/syslog use: *tail -f /var/log/messages*

2.6 The virtual file systems

/proc and /sys files systems don't store their file system on the disk permanently, they generate it every time the Kernel runs or when you ask for it. it's not like the RAM file system, as the RAM file system stores their contents in the RAM.

1. /Proc - coming from process, and it's mounted at the boot time. it contains all the information of the running processes, and you can use the ***ps*** (process status) command to get the information you want from the *proc* file system. every process has a unique id PID, and there are files and directories per process. the process threads is represented in a process subdirectory called 'task'

Listing 12: The proc file system

```
ubuntu@ip-172-31-46-156:/proc$ ls
1 1853 2078 24 4 82 buddyinfo kcore scsi
10 1865 2083 2459 424 829 bus key-users self
1000 1875 2091 25 428 83 cgroups keys slabinfo
1012 1978 2097 2541 430 838 cmdline kmsg softirqs
1015 1979 21 2557 434 84 consoles kpagecgroup stat
1019 2 2113 26 435 845 cpuinfo kpagecount swaps
1034 20 2114 27 436 85 crypto kpageflags sys
1035 2001 2115 271 437 853 devices loadavg sysrq-trigger

ubuntu@ip-172-31-46-156:/proc$ echo $$
1979 # this is the id of the current terminal process
ubuntu@ip-172-31-46-156:/proc$ cd 1979
ubuntu@ip-172-31-46-156:/proc/1979$ ls
attr exe mounts projid_map status
autogroup fd mountstats root syscall
auxv fdinfo net sched task
cgroup gid_map ns schedstat timers
clear_refs io numa_maps sessionid timerslack_ns
cmdline limits oom_adj setgroups uid_map
comm loginuid oom_score smaps wchan
coredump_filter map_files oom_score_adj smaps_rollup
cpuset maps pagemap stack
cwd mem patch_state stat
environ mountinfo personality statm

ubuntu@ip-172-31-46-156:/proc/1979$ ls -l exe
lrwxrwxrwx 1 ubuntu ubuntu 0 Jan 11 17:04 exe -> /bin/bash

ubuntu@ip-172-31-46-156:/proc/1979$ ls task
1979
```

As you might noticed, They are normal text files.

as you can consider the system as a process under */proc*, then you can modify the system behavior in boot time or at runtime using the system tunable parameters (to tune the kernel). the */proc/sys* file system contains all of them. For example searching for ip_forward tunable variable. you will find this variable underneath ”./sys/net/ipv4/ip_forward”, and it’s value is 0 which means ip forwarding is not working.

Listing 13: Tunable variables

```
ubuntu@ip-172-31-46-156:~$ cd /proc
ubuntu@ip-172-31-46-156:/proc$ find . -name ip_forward 2>/dev/null
./sys/net/ipv4/ip_forward
ubuntu@ip-172-31-46-156:/proc$ cat ./sys/net/ipv4/ip_forward
0
```

```
ubuntu@ip-172-31-46-156:/proc$
```

Note: use the cat command to print a file, while echo to set into it.

You can use the command sysctrl to print all the tunable variables.

```
          ubuntu@ip-172-31-46-156:~$ sysctl -a
          abi.vsyscall32 = 1
          debug.exception-trace = 1
          debug.kprobes-optimization = 1
          dev.cdrom.autoclose = 1
          dev.cdrom.autoeject = 0
          ...

          ubuntu@ip-172-31-46-156:~$ sysctl -a | grep ip_forward 2>/dev/null
          net.ipv4.ip_forward = 0
          net.ipv4.ip_forward_use_pmtu = 0
```

You have to notice that you should use the *sysctl* command to deal with the tunable parameters. for example:

or you can change the Linux tunable parameters permanently by writing to the */etc/sysctl.conf* file.

Listing 14: The proc file system

```
ubuntu@ip-172-31-46-156:~$ sysctl -a | grep kernel.panic
kernel.panic = 0 # means the kernel will not reboot automatically if a node panics

ubuntu@ip-172-31-46-156:~$ sudo sysctl -w kernel.panic=10
kernel.panic = 10
# means the system will reboot after 10 seconds if panic happened
```

Note: The Linux tunable parameters can be used for Linux kernel optimization or to change the Linux behavior at runtime or during initialization.

2. /sys it's mounted at boot time, and it is for the 'kernel object' info. it contains info about the the connected HW (e.g. PCI info)

Listing 15: The *sysfs*

```
ubuntu@ip-172-31-46-156:/sys$ ls
block class devices fs kernel power
bus dev firmware hypervisor module

ubuntu@ip-172-31-46-156:/sys$ ls -l /dev/null
# c means character device file, and (1, 3) represents the minor and major numbers correspondingly.
crw-rw-rw- 1 root root 1, 3 Jan 11 14:02 /dev/null
ubuntu@ip-172-31-46-156:/sys$ ^C
ubuntu@ip-172-31-46-156:/sys$ ls -l /dev/zero
crw-rw-rw- 1 root root 1, 5 Jan 11 14:02 /dev/zero
```

3 The device files

Device files, are used for two kinds of devices; character or block devices. each device file have a major number, minor number, and a character to represent if it's block (b) or character (c). and you can interact with the a device by using its device files. the major numbers is used for all the same kind of devices, but the minor number is unique per a device. for example if you have a serial devices, then expect that they will all have the same major number.

The mounting point if the devfs is the /dev node.

3.1 Dealing with a device file

for example the character device files can implement read(), write(), and ioctl(), while enables then to deal with the device files. so a process opens a device file which returns a file descriptor then to use these implemented functions to access the device files.

Listing 16: Access a device file

```
ubuntu@ip-172-31-46-156:/proc$ echo hello > /dev/null
```

The kernel will arrange to call the write() function for the *null* device file

Note: in Linux, the */dev/null* is called the null device file which is absorbing any data written or redirected to it (the Linux black hole), you can use it to ignore info while displaying a certain needed info only. Also there is /dev/zero while a character device file considered an source of zeros, and /dev/random and /dev/urandom which an infinite source of random number (used as a secure source of random number for crypto-systems)

Listing 17: Access a device file

```
ubuntu@ip-172-31-46-156:~$ grep -r power /sys/ 2>/dev/null
/sys/module/pcie_aspm/parameters/policy:[default] performance powersave powersupersave

ubuntu@ip-172-31-46-156:~$
```

The following example throw all the stderr output to the null!, try it without using the */dev/null*.

In the following example, you can check your internet speed without occupying your disk!.

Listing 18: Access a device file

```
$ wget -O /dev/null http://releases.ubuntu.com/18.04.3/ubuntu-18.04.3-desktop-amd64.iso?_ga=2.53207274.31
--2020-01-13 22:22:25-- http://releases.ubuntu.com/18.04.3/ubuntu-18.04.3-desktop-amd64.iso?_ga=2.532
Resolving releases.ubuntu.com (releases.ubuntu.com)... 91.189.88.22, 91.189.88.166, 91.189.88.148, ...
Connecting to releases.ubuntu.com (releases.ubuntu.com)|91.189.88.22|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2082816000 (1.9G) [application/x-iso9660-image]
Saving to: '/dev/null'
```

```
/dev/null 1%[> ] 29.35M 2.34MB/s eta 17m 52s
```

3.2 The block devices

To see the block devices you have, change your directory to */sys/block*.

Listing 19: Access a device file

```
ubuntu@ip-172-31-46-156:~$ cd /sys/block
ubuntu@ip-172-31-46-156:/sys/block$ ls -l
total 0
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop0 -> ../devices/virtual/block/loop0
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop1 -> ../devices/virtual/block/loop1
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop2 -> ../devices/virtual/block/loop2
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop3 -> ../devices/virtual/block/loop3
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop4 -> ../devices/virtual/block/loop4
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop5 -> ../devices/virtual/block/loop5
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop6 -> ../devices/virtual/block/loop6
lrwxrwxrwx 1 root root 0 Jan 17 09:04 loop7 -> ../devices/virtual/block/loop7
lrwxrwxrwx 1 root root 0 Jan 17 09:04 xvda -> ../devices/vbd-768/block/xvda
```

You can use the fdisk command to check all the attached HDDs. use *sudo fdisk -a*

3.3 The Linux file system

Linux file system is single tree (unified file systems hierarchy), And you can install this tree distributed on multiple storage devices
So you have primary partition (contains root), and you can have one or more mounting points on disk. The file system hierarchy is as follow:

- / - the root node (it called slash/)
  - /root - home directory for root user fs(admin). You need permissions to access.
  - /boot - Contains all boot files needs boot the system
  - /home - contains all the users of the system, you can also reach your home using the sign from the command line grub - contains the bootloader, its binary, configs for boot, and kernel image e.g. vmlinuz-4.15.0-1051
  - /bin - contains all the command line applications
  - /lib - contains the shared libs
  - /etc - contains all configuration files for all the programs
  - /dev - its a placeholder for all devices in the system (Linux deal with any devices as files to read and write as buffer)

- /media - the mounted devices (camera, sdcard, ...)
- /mnt - the mounted devices (camera, sdcard, ...)
- /opt - Optional software
- /usr - Shared data or binaries among all users
- /var - variable files or log files ...
- proc - the proc file system is used to hold all the running processes
- sys - Interface for kernel

Listing 20: The Linux File System

```
ubuntu@ip-172-31-46-156:~$ cd /
ubuntu@ip-172-31-46-156:/$ ls
bin home lib64 opt sbin tmp vmlinuz.old
boot initrd.img lost+found proc snap usr
dev initrd.img.old media root srv var
etc lib mnt run sys vmlinuz
```

4 Linux Boot Process

GRUB is the Linux boot loader, the BIOS loads GRUB and GRUB loads the kernel image into memory.

GRUB loads the kernel and the *initfs*, and setup the kernel command line, then transfer the control to the kernel after passing the command line parameters to it. GRUB is built with support of file systems, it can find files, and can load the kernel by name, it can also find the kernel by its file name completion feature.

Note: use *man -k grub* to see more about GRUB features.

4.0.1 The passed command line parameters

GRUB passes the command line parameters to the kernel. so that you can put a script to handle during booting. You can access these command lines by using *dmesg* or by using */proc/cmdline*.

For a documentation about the kernel parameters, you will find in the Linux source tree in *documentation/kernel-parameters.txt*

4.0.2 The init file system

The initial file system or it's called the RAM file system (initrd) is used to provide drivers and support for mounting the system's real file system. The *(*initrd) has the init task which responsible for loading the device files which enables the Kernel to load the real file system. after init in *initrd* terminates, the kernel will start again the real init again this time from the real file system on disk. this init in Linux is called a ***systemd***. this init fs starts the Linux daemons after initializing the kernel. and the configuration of these services is under */etc/systemd/system*.

Listing 21: The Linux File System

```
ubuntu@ip−172−31−46−156:/boot$ which init
/sbin/init
ubuntu@ip−172−31−46−156:/boot$ ls −l /sbin/init
lrwxrwxrwx 1 root root 20 Nov 15 15:01 /sbin/init −> /lib/systemd/systemd

ubuntu@ip−172−31−46−156:/boot$ file /lib/systemd/systemd
/lib/systemd/systemd: ELF 64−bit LSB shared object, x86−64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
BuildID[sha1]=71fba1bd4906a59ccc85c824532db6637358c964, stripped
```

You can find the initramfs in */boot/init/initrd*, it's a CPIO zipped file, the Linux will extract it and mount it into RAM. to explore its internals and know about the services inside it. you can do the follow:

Listing 22: The Linux File System

```
ubuntu@ip−172−31−46−156:~$ ls /boot/
System.map−4.15.0−1051−aws initrd.img−4.15.0−1051−aws
System.map−4.15.0−1057−aws initrd.img−4.15.0−1057−aws
config−4.15.0−1051−aws vmlinuz−4.15.0−1051−aws
config−4.15.0−1057−aws vmlinuz−4.15.0−1057−aws
grub

ubuntu@ip−172−31−46−156:~$ mkdir /tmp/testramfs
ubuntu@ip−172−31−46−156:~$ cd /tmp/testramfs/
ubuntu@ip−172−31−46−156:/tmp/testramfs$ cp /boot/initrd.img−4.15.0−1057−aws /tmp/testramfs/i.gz
ubuntu@ip−172−31−46−156:/tmp/testramfs$ file i.gz
i.gz: gzip compressed data, last modified: Fri Jan 10 22:19:58 2020, from Unix

ubuntu@ip−172−31−46−156:/tmp/testramfs$ gunzip i.gz
ubuntu@ip−172−31−46−156:/tmp/testramfs$ file i
i: ASCII cpio archive (SVR4 with no CRC)

ubuntu@ip−172−31−46−156:/tmp/testramfs$ cpio −i −−no−absolute−filenames <i
107606 blocks

ubuntu@ip−172−31−46−156:/tmp/testramfs$ ls
# Those are the file system that will be loaded to the RAM
bin conf etc i init lib lib64 run sbin scripts usr var

ubuntu@ip−172−31−46−156:/tmp/testramfs$ ls −l init
−rwxr−xr−x 1 ubuntu ubuntu 6682 Jan 17 20:59 init # This is the systemd file
```

Note: Booting the system with rdinit=/bin/sh it will start a shell within initramfs. but If you start using init=/bin/bash it will start a bash after the initramfs is completed from the real file system on the disk. refer to Interrupting GRUB

4.1 Configuring GRUB

You can configure GRUB V1 by editing the *grub.conf* to add kernel entries In GRUB V2 there is a complete directory for that */etc/grub.d*, also you can find */etc/default/grub* for the default configuration. In order to have a new grub entry, you can add a file under */etc/grub.d*.

after adding any entries, then you need to call *update-grub* to update grub with the new info.

You can edit the **40_custom** file in */etc/grub.d* to easily add a new kernel entry.

Listing 23: The Linux File System

```
ubuntu@ip-172-31-46-156:~$ cd /etc/grub.d/
ubuntu@ip-172-31-46-156:/etc/grub.d$ ls
00_header 10_linux 30_os-prober 40_custom README
05_debian_theme 20_linux_xen 30_uefi-firmware 41_custom
ubuntu@ip-172-31-46-156:/etc/grub.d$ cat 40_custom
#!/bin/sh
exec tail -n +3 $0
# This file provides an easy way to add custom menu entries. Simply type the
# menu entries you want to add after this comment. Be careful not to change
# the 'exec tail' line above.

ubuntu@ip-172-31-46-156:/etc/grub.d$ echo "menuentry 'Custom Linux Boot Entry'
{ linux16 /vmlinuz-4.15.0-1051-aws root=/dev/mapper/centos-root ro rd.lvm.lv=centos/root
rd.lvm=centos/swap rhgb quiet LANG=en_US.UTF-8 initcall_debug initrd16 /initrd.img-4.15.0-1057-aws
}" >> 40_custom


ubuntu@ip-172-31-46-156:/boot$ sudo grub-mkconfig -o /etc/grub2.cfg
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/50-cloudimg-settings.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.15.0-1057-aws
Found initrd image: /boot/initrd.img-4.15.0-1057-aws
Found linux image: /boot/vmlinuz-4.15.0-1051-aws
Found initrd image: /boot/initrd.img-4.15.0-1051-aws
done
```

you can notice that we put **initcall_debug** as a part of our menuentry, so after rebooting, you can type *dmesg — initcall* and it will show all the log of init and order of executing and loading of the devices. then you can analyze and debug the init task.

4.2 Interrupting GRUB

Normally you can do that by pressing any key while booting. and then temporarily edit the GRUB configurations.

You can then press 'P' (GRUB 1) or Ctrl+'x' (GRUB 2) to resume booting.

You can also overriding the initrd;

1. reboot the system
2. after rebooting hit the down arrow to interrupt GRUB
3. then select any kernel entry, then type 'e' to edit its configurations.
4. the configuration will appear, just add at last *init=/bin/bash.* what do you notice? You will see that we have overridden the *initrd* and just jumped to the real fs on disk.

¡¡Photo here from reboot¿¿

5 Building the Linux kernel

You need first to get a kernel version from the kernel official website.

1. Get the Linux kernel source code: wget https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.3.15.tar.xz
2. Un-tar it: tar xf archive.tar.xz
3. You can use ***make menuconfig*** or ***make xconfig*** to configure the kernel. All the configurations will be stored in the *.config* file
4. Then you can use different targets to build the kernel; *make bzimage*, *make modules*, *make install*, *make modules_install*, or *make clear.*

Note: You can use *make help* to guide your about the targets of the makefile.

5.1 kernel patches

6 The loadable kernel modules (LKM)

They are .ko (kernel objects) files which **dynamically** adds more functionality to the kernel itself, and they run in the kernel space. So you can add a functionality to the kernel when you need without editing the kernel. and keeps the kernel with small size.

So where to find the kernel modules, simply in */lib/modules*

Listing 24: The Loadable Kernel Modules

```
ubuntu@ip-172-31-46-156:/lib/modules$ ls
4.15.0-1051-aws 4.15.0-1057-aws
# Two folder for each kernel image

ubuntu@ip-172-31-46-156:/lib/modules$ uname -r
4.15.0-1057-aws

ubuntu@ip-172-31-46-156:/lib/modules/4.15.0-1057-aws$ ls
build modules.alias.bin modules.dep.bin modules.symbols
initrd modules.builtin modules.devname modules.symbols.bin
kernel modules.builtin.bin modules.order vdso
modules.alias modules.dep modules.softdep

ubuntu@ip-172-31-46-156:/lib/modules/4.15.0-1057-aws$ cd kernel
ubuntu@ip-172-31-46-156:/lib/modules/4.15.0-1057-aws/kernel$ ls
arch block crypto drivers fs lib net spl virt zfs
# The kernel modules are grouped according to their function.

ubuntu@ip-172-31-46-156:/lib/modules/4.15.0-1057-aws/kernel$ find . -name '*.ko' | wc -l
949 # we have this number of .ko module files
```

and you will find also some configuration files at the same directory.

the modules can be at any directory, but the *modprobe* is designed to look only in */lib/modules*.

Note the LKM should complied to a certain kernel version not the other.

6.1 LKMs Commands

- The *lsmod* will list all the loaded module with its usage and its dependencies, if a modules depend on another modules, then the last one must be loaded before the new one otherwise an error will appear.

Listing 25: Lisiting the kernel modules

```
ubuntu@ip-172-31-46-156:/lib/modules/4.15.0-1057-aws/kernel$ lsmod
Module Size Used by
ufs 77824 0
msdos 20480 0
xfs 1204224 0
binfmt_misc 20480 1
serio_raw 16384 0
sch_fq_codel 20480 2
ib_iser 49152 0
rdma_cm 61440 1 ib_iser
iw_cm 45056 1 rdma_cm
ib_cm 53248 1 rdma_cm
ib_core 225280 4 rdma_cm,iw_cm,ib_iser,ib_cm

ubuntu@ip-172-31-46-156:/lib/modules/4.15.0-1057-aws/kernel$ lsmod | wc -l
```

```
44 # how many modules loaded
```

- The *rmmod* will remove a module, and take care when removing a module which already in use, It can panic the kernel.

- The *modinfo* will print t a meta-data about the module; name, author, or parameters, aliases, ...
- The *depmod* will create file for the dependencies of the modules (”modules.dep” file in */lib/nodules/KER_VER*).
- The *insmod*, and it returns after the module initialization done. (don't use and use *modprobe* instead)
- The *modprobe* load a module and looks for its dependencies which are generated from *depmod* Note: *modprobe* can do other operations on the modules.

6.2 Creating and Building a simple kernel module

A simple loadable kernel module:

Listing 26: A source of a simple LKM

```c
#include <linux/init.h> // Macros used to mark up functions e.g., __init __exit
#include <linux/module.h> // Core header for loading LKMs into the kernel
#include <linux/sched.h> // Contains types, macros, functions for the kernel

int simple_init(void){
    printk("Hello for the init function");
    return 0;
}

void simple_exit(void){
    printk("Hello from the exit function");
}

/** @brief A module must use the module_init() module_exit() macros from linux/init.h, which
 * identify the initialization function at insertion time and the cleanup function (as
 * listed above)
 */
module_init(simple_init);
module_exit(simple_exit);
```

Listing 27: A source of a simple LKM

```
ubuntu@ip-172-31-46-156:~$ mkdir simple_lkm
ubuntu@ip-172-31-46-156:~$ cd simple_lkm/
ubuntu@ip-172-31-46-156:~/simple_lkm$ vim simple_module.c
```

```
ubuntu@ip−172−31−46−156:~/simple_lkm$ echo ”obj−m := simple_module.o” > makefile
ubuntu@ip−172−31−46−156:~/simple_lkm$ ls
Makefile simple_module.c

ubuntu@ip−172−31−46−156:~/simple_lkm$ make −C /lib/modules/$(uname −r)/build M=$PWD modules
```

6.3 Developing Device Drivers by LKMs

7 Little bit about the application development

In Linux, as an application developer you can develop your application in ”C” in different ways:

- Using the Linux *syscalls*
- Interfacing the *syscalls* by the libc.
- Or you can use an application framework like QT to even develop a beautiful GUI applications

7.1 Meet Vim, GCC, and GDB

Vim is the command line text editor in Linux, it's very powerful tool to edit the text files without any GUI interface.
Why to use Vim ?
In embedded systems developments and debugging, you might not having the facility to edit the files using a full featured text editor over terminal.

Vim Modes:

- Text Mode
- Command Mode

Note: for multiple files projects you might need to use *make* or *cmake* to manage your building process.

7.1.1 GDB - The GNU Debugger

First you need to compile your code using debug symbols as follow, and note that you can use *-g or -ggdb* for a gdb format, also you can choice the level of the debugging info by specifying -g1, -g2, or -g3:
g++ -g example.cpp -o output.exe

Sometimes, people choice -O3 while compiling for optimization level3, if you use *-O3 -g* but the information while debugging will be poor.

In order to use GDB, you can run the following command, *gdb -q ./output.exe*, as you can see in the following example, you can find that gdb is working

Listing 28: Running the GDB

```
$ gdb −q ./output.exe
Reading symbols from ./output.exe...done.
(gdb)
```

You can make several operations using gdb, you need to set a break point, and then type run, gdb will stop at the break point:

Listing 29: GDB Basic Operations

```
$ gdb −q ./output.exe
Reading symbols from ./output.exe...done.
(gdb) break main # break keyword will set a break point in main, you can use also to set break at line number
Breakpoint 1 at 0x401574: file example.cpp, line 4.
(gdb) run # run the application
Starting program: /c/Users/aramadan/Desktop/TheDocumentationProject/linux−user/resources/src/gdb/output.ex
[New Thread 12904.0xb14]
[New Thread 12904.0x2cb8]

Thread 1 hit Breakpoint 1, main (argc=1, argv=0x1614e0) at example.cpp:4
4 for(i=0; i < 10; i++)
(gdb) continue # continue to the next point
Continuing.
Hello, world!

$ gdb −q ./output.exe
Traceback (most recent call last):
  File "<string>", line 3, in <module>
ModuleNotFoundError: No module named 'libstdcxx'
/etc/gdbinit:6: Error in sourced command file:
Error while executing Python code.
Reading symbols from ./output.exe...done.
(gdb) break main
Breakpoint 1 at 0x401574: file example.cpp, line 4.
(gdb) run
Starting program: /c/Users/aramadan/Desktop/TheDocumentationProject/linux−user/resources/src/gdb/output.ex
[New Thread 12904.0xb14]
[New Thread 12904.0x2cb8]

Thread 1 hit Breakpoint 1, main (argc=1, argv=0x1614e0) at example.cpp:4
4 for(i=0; i < 10; i++)

(gdb) continue
Continuing.
Hello, world!

Hello, world!

...
```

```
[Inferior 1 (process 12904) exited normally]
(gdb) quit
```

Now you can type some commands to GDB to explore the memory, and here is some examples:

1. *info registers* can be used to list all the register, you can also to get a value inside a certain register

Listing 30: Working with registers

```
$ gdb -q ./output.exe
Reading symbols from ./output.exe...done.
(gdb) break main
Breakpoint 1 at 0x401574: file example.cpp, line 4.
(gdb) run
Starting program: /c/Users/aramadan/Desktop/TheDocumentationProject/linux-user/resources/src/gdb/outp
[New Thread 10096.0x898]
[New Thread 10096.0x2c70]

Thread 1 hit Breakpoint 1, main (argc=1, argv=0xbf14e0) at example.cpp:4
4 for(i=0; i < 10; i++)
(gdb) info register
rax 0x1 1
rbx 0x0 0
rcx 0x1 1
rdx 0xbf14e0 12522720
rsi 0x5a 90
rdi 0xbf1520 12522784
rbp 0x65fe20 0x65fe20
rsp 0x65fdf0 0x65fdf0
r8 0xbf1e40 12525120
r9 0x13 19
r10 0x0 0
r11 0x65fc88 6683784
r12 0x0 0
r13 0x10 16
r14 0x0 0
r15 0x0 0
rip 0x401574 0x401574 <main(int, char**)+20>
eflags 0x202 [ IF ]
cs 0x33 51
ss 0x2b 43
ds 0x2b 43
es 0x2b 43
fs 0x53 83
gs 0x2b 43

(gdb) info register rip # get a certain register value
rip 0x401574 0x401574 <main(int, char**)+20>
```

2. *"x" short for examine* is used to have direct access to the memory, it's used to display the memory. you can examine in different formats

   - o Display in octal.
   - x Display in hexadecimal.
   - u Display in unsigned, standard base-10 decimal.
   - t Display in binary.

   The *"x"* is very powerful as it can explore the memory in a lot of ways, let's have some examples:

   Listing 31: Display a memory locations

   ```
   (gdb) info register rip
   rip 0x401574 0x401574 <main(int, char**)+20>
   (gdb) x/x 0x401574
   0x401574 <main(int, char**)+20>: 0x00fc45c7
   (gdb) x/o 0x401574
   0x401574 <main(int, char**)+20>: 077042707
   (gdb) x/t $rip
   0x401574 <main(int, char**)+20>: 00000000111111000100010111000111
   (gdb) x/16x $rip
   0x401574 <main(int, char**)+20>: 0x00fc45c7 0x83000000 0x7f09fc7d
   0x0d8d4812
   0x401584 <main(int, char**)+36>: 0x00002a78 0x0015e3e8 0xfc458300
   0xb8e8eb01
   0x401594 <main(int, char**)+52>: 0x00000000 0x30c48348 0x9090c35d
   0x28ec8348
   0x4015a4 <__do_global_dtors+4>: 0x65058b48 0x4800001a 0x8548008b
   0xff1d74c0
   (gdb) x/12x $rip
   0x401574 <main(int, char**)+20>: 0x00fc45c7 0x83000000 0x7f09fc7d
   0x0d8d4812
   0x401584 <main(int, char**)+36>: 0x00002a78 0x0015e3e8 0xfc458300
   0xb8e8eb01
   0x401594 <main(int, char**)+52>: 0x00000000 0x30c48348 0x9090c35d
   0x28ec8348
   ```

   as you can see, we displayed the memory of the next instruction stored in *rip* register, even you can reference the register to it the address stored on it using *$rip*, and for all times the outputs is the same but in different formats.

   Even you can display the memory in different sizes, as the default examine size if *word*, then you have other ways to how the memory is treated and this can be useful in certain locations:

   Listing 32: Change the size of printed data

```
(gdb) x/12xh $rip
0x401574 <main(int, char**)+20>: 0x45c7 0x00fc 0x0000 0x8300
0xfc7d 0x7f09 0x4812 0x0d8d
0x401584 <main(int, char**)+36>: 0x2a78 0x0000 0xe3e8 0x0015
(gdb) x/12xb $rip
0x401574 <main(int, char**)+20>: 0xc7 0x45 0xfc 0x00 0x00 0x00
0x00 0x83
0x40157c <main(int, char**)+28>: 0x7d 0xfc 0x09 0x7f
```

- A single byte
- h A halfword, which is two bytes in size
- w A word, which is four bytes in size
- g A giant, which is eight bytes in size

As the gdb is used also to disassemble the code, then you can use the "x" command to display the disassembled instructions, hence you can use "x" instruction to display the number of disassembled instructions as follow:

Listing 33: Display a memory locations

```
(gdb) x/4i $rip
=> 0x401574 <main(int, char**)+20>: movl $0x0,-0x4(%rbp)
   0x40157b <main(int, char**)+27>: cmpl $0x9,-0x4(%rbp)
   0x40157f <main(int, char**)+31>: jg 0x401593 <main(int, char**)+51>
   0x401581 <main(int, char**)+33>: lea 0x2a78(%rip),%rcx # 0x404000
(gdb)
```

One hint: in GDB you can use a full text commands or you might need to write a abstracted commands, see the example below, instead of writing *"info register rip"*, *"i r rip"*, and they will give the same result.

Listing 34: Write your GDB instructions in an easy way

```
(gdb) info register rip
rip 0x401574 0x401574 <main(int, char**)+20>
(gdb) i r rip
rip 0x401574 0x401574 <main(int, char**)+20>
```

7.1.2 Creating GDB Scripts

A good thing when you use GDB is that you can setup a script to run as a debugging scenario, and let it get the debugging info you needed for you.

By default during startup, gdb executes the file *.gdbinit*. This is where you write your gdb code. In case you want to have many scripts that test different things, you can tell gdb to look at other scripts besides the default one by adding the *–command=¡filename¿* argument when running gdb.

by writing ***gdb -q −command=backtrace-script***, you will get trace of all frames in stack of our program:

Listing 35: backtrace script output

```
Num Type Disp Enb Address What
1 breakpoint keep y 0x0000000000000667 in main(int, char**) at example.cpp:14
2 breakpoint keep y 0x000000000000064d in func1() at example.cpp:8
        bt
        continue

Breakpoint 1, main (argc=1, argv=0x7fffffffe508) at example.cpp:14
14 for(i=0; i < 10; i++)
```

and the script is:

Listing 36: backtrace script

```
set pagination off
set logging file gdb.txt
set logging on
file output.exe
b main
b func1
commands
bt
continue
end
info breakpoints
r
set logging off
quit
```

for more info, you can refer to the gdb quick reference from here.

7.2 Concurrent Programming

You can achieve concurrent programming in Linux using two ways:

- Kernel Level threads Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel.

- User Threads using threads library (pThreads), or C++ threads using C++ thread library The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads.

7.2.1 Linux Threads

You can create a kernel level threads in two ways; Using fork(), or clone() system calls, but before let's have a look on how threads works in the different OSs:
In windows:
- it will create a PCB, Process Control Block (in the kernel space) related to that PID and points to the main process of that program, when the processor starts a new thread - it's allocates a new dynamic memory and puts this code into it and creates a new thread object which points into it (with the same PID) - The key point now is whatever the CPU time acquired by this PCB (represent the main program), that has to be shared on these threads, and the piece of code which will divide this time among the threads called -¿ user level scheduler and if we need more time to be allocated for a certain thread, we need to change it's priority, note that the kernel scheduler is not aware of the contexts or threads in the user space, the object that holds an info about the thread is called thread object - For each thread -¿ ThreadObject, + Code + Stack - but the issue of that model is, that the user space scheduler is not performing a full context switching but when the time of a thread ends, the user space scheduler just will move the control o the other thread, it's conditional jump to the other thread

In Unix based thread:
- A new address space in the user space will be allocated for every thread, - A new PCB address space in the kernel space will be allocated to point on every thread, every thread is a new process (with differnet PID) identified by the kernel - The difference now is all the info about the threads will be allocated in the PCB related to that thread in the KERNEL SPACE with the control of the kernel - This kind of threading is called kernel supported threading

-In the matter of performance windows is good (because switching between thread is just a conditional jump and no context switching), in the matter of fault tolerance, linux is better -Comparing the programmers friendly for each one, the easy to debug and the easy to deal with is the user threading module in windows is more friendly because a global varaibles is not accessible anymore in linux after creating a threads to share data, the same is not here in unix to exchange data between thread, the developer should write extra code for sharing between them this called interprocess communication IPC! - In the matter of distrusting the thread between cores, Unix based are better because in user level threading, you have all the threads within the same PCB then they share the same core, but for the Unix based it's possible to have thread1 in a core, thread2 in another core ...

But Linux came with the best of two approaches, LWP Light-Weight Process, Linux got a concept of clone in place of fork, clone will create a new address space as fork exactly, but it will allow a the processes o share resources with parents and now they can access the globle data in data segement or file descriptors for the parent, which means no need for IPC anymore.

Using fork():
fork -¿ sys_fork -¿ do_fork - all the resources in the parent PCB are copied into

the child PCB - After the child process terminates, the parent process should read its exit, if not the child enters a state called zoombie state or defunct state - every process enters this state !, at this moment the PCB for this process is still exists, and all the resources for it are locked up, when all processes are in zoombie, the system is frozen coz all resources locked up - There are two approaches to notify the parent about exit 1. Sync - suspend the parent till the child terminates 2. Async - Child sends a signal to the parent

What is happening with fork() ?
- parent's page tables(PCB, and address space, stack, global, and local data) been copied to child to create a unique task structure for the child! - But this is not right, actually what happens, that the Linux makes a Copy-On-write ONLY, means that the resources shared with parent are read-only, and when acquire to write, the linux will copy this portion in memory and write into it

Listing 37: Consumer Producer example

```
#define CHILD 0
main(){

        int fd;
        char buf[2];
        pid_t pid;
        int childstatus;
        fd = open("./test",0_RDONLY);
        pid= fork();
        if (pid == CHILD){
                read(fd,buf,2); // here the child is delivered the file with open state
                printf("in child %c\n",buf[0]); //output: a
                printf("in child %c\n",buf[1]); //output: b
                close(fd); //here the Copy-On-Write been triggered, copy the fd and close it
        }else{
                //parent
                wait(&childstatus); //suspend the parent till child completes
                read(fd,buf,2); //here will read from the original file descriptor, which the index shifted by 2
                printf("in parent %c\n",buf[0]); //output: c
                printf("in parent %c\n",buf[1]); //output: d
                close(fd);
        }
}
```

Clone system call:
It is used to create multiple threads that can concurrently run in the same shared memory space!, clone is the same fork(), but you control the resources you share with child

You can create threads using clone as following: pid_for_child = clone(funn_pointer_for_the_child, stack+stack_size, flags, any_args_passed_to_child)

7.2.2 POSIX Threads (pthreads)

On the other side, you can use the POSIX thread to create a user-level threads. and this example as a simple thread:

Listing 38: Simple pThreads example

```
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

void *PrintHello(void *threadid) {
   long tid;
   tid = (long)threadid;
   cout << "Hello World! Thread ID, " << tid << endl;
   pthread_exit(NULL);
}

int main () {
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);

      if (rc) {
         cout << "Error:unable to create thread," << rc << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

The difference between threads and fork, fork creates another process which has its own PDT (Process Descriptor table) but Pthreads sharing the same PDT Note: pthreads APIs use clone syscall

You can use also use a synchronization methods like mutex and conditional variables as follow:

Listing 39: Consumer Producer example

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>

#define MAX 1
```

```
int buffer[MAX];
int fill = 0;
int use = 0;
int loops = 0;

sem_t empty;
sem_t full;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}

int get() {
    int b = buffer[use];
    use = (use + 1) % MAX;
    return b;
}



void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        int b = get();
        sem_post(&empty);
        printf(”%d\n”, b);
    }
}

int main(int argc, char *argv[])
{
    if(argc < 2 ){
        printf(”Needs 2nd arg for loop count variable.\n”);
        return 1;
    }

    loops = atoi(argv[1]);

    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0); // ... and 0 are full

    pthread_t pThread, cThread;
    pthread_create(&pThread, 0, producer, 0);
    pthread_create(&cThread, 0, consumer, 0);
    pthread_join(pThread, NULL);
    pthread_join(cThread, NULL);
    return 0;
}
```

7.2.3 C++ Threads (C++11 and C++14)

Here is a simple example of creating a simple thread

Listing 40: A source of a simple LKM

```
#include <iostream>
#include <thread>

void thread_function()
{
    std::cout << ”thread function\n”;
}

int main()
{
    std::thread t(&thread_function); // t starts running
    std::cout << ”main thread\n”;
    t.join(); // main thread waits for the thread t to finish
    return 0;
}
```

7.2.4 Socket Programming

7.2.5 Linux Application Development

7.2.6 Meet Gnome GDK+ and KDE QT

8 Installing Ubuntu distribution

While installing, how to partition your hard disk

9 Bash Scripting

10 some useful commands

1. Check the kernel version; use the uname command (unix name)

```
    ubuntu@ip-172-31-46-156:~$ uname -r
    4.15.0-1057-aws
    ubuntu@ip-172-31-46-156:~$
```

2. check the memory usage, you can use the procfs

```
ubuntu@ip-172-31-46-156:~$ head /proc/meminfo
MemTotal: 1007300 kB
MemFree: 351080 kB
MemAvailable: 734812 kB
Buffers: 32956 kB
Cached: 438592 kB
SwapCached: 0 kB
Active: 361348 kB
Inactive: 160464 kB
Active(anon): 50860 kB
Inactive(anon): 164 kB
ubuntu@ip-172-31-46-156:~$
```

3. use the strace command (*syscall* trace) to check the usage of an application to the system calls Example: make a report of usage of the syscalls for date command

Listing 41: The Linux File System

```
ubuntu@ip-172-31-46-156:~$ strace -c date
Mon Jan 13 20:47:21 UTC 2020
% time seconds usecs/call calls errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00 0.000000 0 5 read
  0.00 0.000000 0 1 write
  0.00 0.000000 0 21 close
  0.00 0.000000 0 21 fstat
  0.00 0.000000 0 1 lseek
  0.00 0.000000 0 19 mmap
  0.00 0.000000 0 4 mprotect
  0.00 0.000000 0 1 munmap
  0.00 0.000000 0 3 brk
  0.00 0.000000 0 3 3 access
  0.00 0.000000 0 1 execve
  0.00 0.000000 0 1 arch_prctl
  0.00 0.000000 0 1 clock_gettime
  0.00 0.000000 0 19 openat
------ ----------- ----------- --------- --------- ----------------
100.00 0.000000 101 3 total
ubuntu@ip-172-31-46-156:~$
```

the date command prints the date and uses for that a system calls, we use strace with the option -c (count) to count the usage of the *syscalls*.

4. Search for all the device files with the same major number. for example, /dev/null has the major number 1, and minor number 3 as follow, then we can

use grep to get all the device files "ˆc" then to match again all the device files with major number 1.

Listing 42: Lisiting all the device files with the same major number

```
ubuntu@ip-172-31-46-156:~$ ls -l /dev/null
crw-rw-rw- 1 root root 1, 3 Jan 17 08:28 /dev/null

ubuntu@ip-172-31-46-156:~$ ls -l /dev | grep ^c | grep " 1,"

crw-rw-rw- 1 root root 1, 7 Jan 17 08:28 full
crw-r--r-- 1 root root 1, 11 Jan 17 08:28 kmsg
crw-r----- 1 root kmem 1, 1 Jan 17 08:28 mem
crw-rw-rw- 1 root root 1, 3 Jan 17 08:28 null
crw-r----- 1 root kmem 1, 4 Jan 17 08:28 port
crw-rw-rw- 1 root root 1, 8 Jan 17 08:28 random
crw-rw-rw- 1 root root 1, 9 Jan 17 08:28 urandom
crw-rw-rw- 1 root root 1, 5 Jan 17 08:28 zero
```

5. The pstree command
it prints a tree for all the relationship of the processes.

Listing 43: The first process in pstree command

```
ubuntu@ip-172-31-46-156:/boot$ pstree | head
systemd-+-accounts-daemon---2*[{accounts-daemon}]
        |-acpid
        |-2*[agetty]
        |-amazon-ssm-agen---7*[{amazon-ssm-agen}]
        |-atd
        |-cron
        |-dbus-daemon
        |-lvmetad
        |-lxcfs---2*[{lxcfs}]
        |-networkd-dispat---{networkd-dispat}
```

6. The systemctl - working with the *systemd* services The command used to check if a systemd daemon (or service) is running

Listing 44: Lisiting all the device files with the same major number

```
ubuntu@ip-172-31-46-156:~$ systemctl status sshd
 ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enab
   Active: active (running) since Sat 2020-02-29 13:24:14 UTC; 12min ago
 Main PID: 961 (sshd)
    Tasks: 1 (limit: 1152)
   CGroup: /system.slice/ssh.service
           961 /usr/sbin/sshd -D

ubuntu@ip-172-31-46-156:~$ ps -ef | grep sshd
root 961 1 0 13:24 ? 00:00:00 /usr/sbin/sshd -D
root 8871 961 0 13:32 ? 00:00:00 sshd: ubuntu [priv]
ubuntu 9006 8871 0 13:32 ? 00:00:00 sshd: ubuntu@pts/0
ubuntu 9029 9012 0 13:33 pts/0 00:00:00 grep −−color=auto sshd
```

You can use also use the *ps* command indirectly to see if the sshd service is running as one of the process as shown.

10.1 But what is systemd ?

7. Using cat command to append into files, just use ¡¡EOF¿¿ with cat. then write what ever you want and when you type EOF it will exit!.

Listing 45: Cat to append on text files

```
aramadan@CAI1−L11666 MSYS ˜
$ touch testfile

aramadan@CAI1−L11666 MSYS ˜
$ cat <<EOF>> testfile
> Hello
> World!
> EOF

aramadan@CAI1−L11666 MSYS ˜
$ cat <<EOF>> testfile
> again
> EOF

aramadan@CAI1−L11666 MSYS ˜
$ cat testfile
Hello
World!
again
```

8. History command for shell, it get's use all the commands you run on your bash and you can configure it to keep as much history as you can

Listing 46: History Command

```
aramadan@CAI1−L11666 MSYS ˜
$ history
    1 cd /c/Users/aramadan/Desktop/clang_AST
    2 clang++ −std=c++11 −stdlib=libc++ main.cpp −fno−rtti −IC:\msys64\mingw64\include −LC:\msys64\m
    3 −lclangFrontend −lclangDriver −lclangSerialization −lclangCodeGen −lclangParse −lclangSema −lclangStatic
    4 erCheckers −lclangStaticAnalyzerCore −lclangAnalysis −lclangARCMigrate −lclangRewrite −lclangRewriteFro
...
    180 cat <<EOF>> testfile
    again
    EOF

    181 cat testfile
    182 history
```

9. working with files and directory using *chmod*

Listing 47: File Permissions using chmod

```
aramadan@CAI1-L11666 MINGW64 ~
$ ls -l
total 7
-rw-r--r-- 1 aramadan VNET+Group(513) 2610 Feb 29 16:24 aws-ubuntu
-rw-r--r-- 1 aramadan VNET+Group(513) 574 Feb 29 16:24 aws-ubuntu.pub
-rw-r--r-- 1 aramadan VNET+Group(513) 11 Feb 29 14:21 file
-rw-r--r-- 1 aramadan VNET+Group(513) 20 Feb 29 21:49 testfile
|[-][-][-][-][------] [-------------]
| | | | | | |
| | | | | | +------------> Group
| | | | | +--------------------> Owner
| | | | +-------------------------> Indicates # of files inside directory
| | | +----------------------------> Others Permissions
| | +-------------------------------> Group Permissions
| +----------------------------------> Owner Permissions
+-------------------------------------> File Type (7 Type in Linux)

aramadan@CAI1-L11666 MINGW64 ~
$ chmod u-w testfile

aramadan@CAI1-L11666 MINGW64 ~
$ ls -l
total 7
-rw-r--r-- 1 aramadan VNET+Group(513) 2610 Feb 29 16:24 aws-ubuntu
-rw-r--r-- 1 aramadan VNET+Group(513) 574 Feb 29 16:24 aws-ubuntu.pub
-rw-r--r-- 1 aramadan VNET+Group(513) 11 Feb 29 14:21 file
-r--r--r-- 1 aramadan VNET+Group(513) 20 Feb 29 21:49 testfile

aramadan@CAI1-L11666 MINGW64 ~
$ chmod g+w testfile

aramadan@CAI1-L11666 MINGW64 ~
$ chmod o+w testfile
```

You can use also *umask* command which allows you to view or to set the file mode creation mask, which determines the permissions bits for newly created files or directories. It is used by mkdir, touch, tee and other commands that create new files and directories.

Listing 48: File Permissions using umask

```

aramadan@CAI1-L11666 MINGW64 ~
$ umask
0022
```

```
aramadan@CAI1−L11666 MINGW64 ˜
$ umask −S
u=rwx,g=rx,o=rx

##Not complete
```

and you can change the ownership using *chown*:

Listing 49: ownership Control

```
# change ownership to user: root, and group hacker
aramadan@CAI1−L11666 MINGW64 ˜
$ chown root:hacker testfile

# Only change the group
aramadan@CAI1−L11666 MINGW64 ˜
$ chgrp hacker testfile
```

Understanding the ACL (Access Control Lists)

Listing 50: getfacl and setfacl

```
# setfacl −m ”u:user:permissions” <file/dir>

# getfacl <file/dir>

# not complete
```